HANDBOOK

Programming with AI

Lunario

version 2.0 · complete · 2026

Contents

PART I · FOUNDATIONS

1 Mental model and working cycle 2
- 1.1 Who is in charge here 2
- 1.2 The working cycle 4
- 1.3 What AI does well and what it does badly 6
- 1.4 Three skills 9
- 1.5 The cost of a mistake 11
- 1.6 The illusion of finished code 14

2 Reading code 17
- 2.1 Why read code if the AI writes it 17
- 2.2 Anatomy of a file 19
- 2.3 Anatomy of a project 22
- 2.4 Variables, types, data structures 25
- 2.5 Functions 28
- 2.6 Conditions and loops 31
- 2.7 Classes and objects 33
- 2.8 Exceptions and error handling 36
- 2.9 Decorators and “magic” 38
- 2.10 Asynchrony 41
- 2.11 Dependencies 44
- 2.12 Reading a traceback 47
- 2.13 Reading a diff 51
- 2.14 Red flags in generated code 54

3 Stating the task 57
- 3.1 Why the AI does the wrong thing 57
- 3.2 Anatomy of a prompt 60
- 3.3 Levels of detail 63
- 3.4 Template — a new function 65
- 3.5 Template — fix a bug 68
- 3.6 Template — refactoring 70
- 3.7 Template — explain this code 73
- 3.8 Template — review this 75
- 3.9 Template — write tests 78
- 3.10 Technique — plan first, code second 80
- 3.11 Technique — let the AI ask the questions 82
- 3.12 Technique — input → output examples 85
- 3.13 Technique — limit the blast radius 87
- 3.14 The project context file 90
- 3.15 How to write a mini-spec 92
- 3.16 Prompt antipatterns 95

4 Checking and debugging 99
- 4.1 The rule of trust 99
- 4.2 Running it and reading the logs 101
- 4.3 Preparing an error to be fixed 104
- 4.4 Narrowing the search 107
- 4.5 Debugging with print 109
- 4.6 Tests — read them and run them 112
- 4.7 The acceptance checklist 114
- 4.8 Hallucinations 117

4.9 The "fixed one, broke another" loop 120
4.10 When to stop and roll back 123

5 Process, tools, safety 126
5.1 Git as insurance 126
5.2 Chat or terminal agent 128
5.3 Session context 131
5.4 When to start a new session 133
5.5 Choosing a model for the task 136
5.6 The environment 138
5.7 Secrets 140
5.8 Basic deployment 143

PART II · STACKS

6 Python and application logic 148
6.1 Anatomy of a Python project 148
6.2 Idioms the AI writes constantly 151
6.3 Typing, seriously 155
6.4 Asynchrony in practice 158
6.5 Stating tasks in Python 161
6.6 AI mistakes in Python, and a checklist 163

7 Web backend 167
7.1 How a backend is built 167
7.2 HTTP in plain terms 170
7.3 APIs, schemas and validation 173
7.4 Reading backend code 176
7.5 The database 179
7.6 Queries that drag 182
7.7 Migrations 185
7.8 Access, AI mistakes, and a checklist 188

8 Frontend 191
8.1 How a browser assembles a page 191
8.2 Reading HTML 193
8.3 Reading CSS 196
8.4 Developer tools 200
8.5 AI mistakes in the frontend, and a checklist 203

9 JavaScript and TypeScript 207
9.1 Reading JavaScript 207
9.2 Asynchrony in JavaScript 210
9.3 What TypeScript adds 213
9.4 How to read other people's types 216
9.5 Types as a check on the AI's work 219
9.6 AI mistakes in JS and TS, and a checklist 223

10 React in TypeScript 226
10.1 The anatomy of a React project 226
10.2 Components and JSX 229
10.3 Props and state 232
10.4 Effects 235
10.5 Stating interface tasks 238
10.6 AI mistakes in React, and a checklist 241

11 Flutter and Dart 244
11.1 The anatomy of a Flutter project 244

11.2 Dart in half an hour ... 247
11.3 Widgets ... 250
11.4 State ... 253
11.5 Stating mobile interface tasks ... 257
11.6 AI mistakes in Flutter, and a checklist ... 260

12 Building and shipping ... 263
12.1 What a build is and why it breaks ... 263
12.2 Android and iOS ... 269
12.3 Shipping and updates ... 274
12.4 A checklist before shipping ... 278

PART III · CROSS-CUTTING TOPICS

13 Architecture in plain terms ... 283
13.1 How to break an idea into parts ... 283
13.2 Folder structure ... 287
13.3 Where to keep the data ... 291
13.4 Configs and environments ... 295
13.5 How to describe the architecture to an AI ... 300
13.6 Questions before starting a project ... 304

14 Workshop ... 307
14.1 Case — a Telegram bot with an external API ... 307
14.2 Case — a web API with a database ... 312
14.3 Case — a React site for that API ... 315
14.4 Case — a mobile application ... 319
14.5 Case — an evening's script ... 322

Appendices ... 326
A Request templates ... 326
B A code review checklist ... 329
C Common Python errors ... 330
D Common JS, TypeScript and Dart errors ... 333
E Git — rescue scenarios ... 335
F The terminal — the necessary minimum ... 339
G Glossary ... 341

Index ... 346

PART I

Foundations

Three skills that depend on neither the language, nor the library, nor which model you happen to use: stating the task, reading someone else's code, and checking the result. This part is the core of the book. If you read only this, you will get more out of it than out of any of the later parts on its own.

1 Mental model and working cycle

Before getting into techniques, the main thing has to be settled — who in this pair is responsible for what, and in what order the work happens. Everything else in the book rests on this chapter.

1.1 Who is in charge here

the roles of human and AI, and what happens when they blur

WHAT IT IS

Working with an AI has a structure worth saying out loud, because by default it blurs.

You are the architect and the reviewer. The AI is the implementer.

You decide what the result should be and whether what came out fits. The AI proposes how to do it, and does it. The decision stays with you — not because the AI is stupid, but because you are the one who will answer for the result.

WHAT THIS MEANS IN PRACTICE

The architect decides what to build

Which problem is being solved at all. What parts the project has. Where the data lives. What matters more — speed or simplicity. These are decisions that cannot be delegated: the AI has neither your constraints, nor your plans, nor any knowledge of why the whole thing was started.

The reviewer decides what to accept

Code does not become yours because it appeared in your file. It becomes yours when you have read it and agreed. Until that moment, what sits in the project is someone else's decision that you never checked.

The implementer builds and proposes

The AI writes code well, suggests options, explains other people's decisions and notices what you missed. It can — and should — argue

and propose something better (article 3.11). But the last word is not its own.

> **REMEMBER**
> You are the one who will answer for this code in front of users, in front of the server, and in front of yourself a month from now. The fact that an AI wrote it changes nothing about that.

WHAT HAPPENS WHEN THE ROLES BLUR

Two extremes, and both are common.

You become the implementer

It looks like this: got the code, didn't understand it, pasted it, ran it. Didn't work — asked for a fix. Still didn't understand. Repeat.

This is exactly the no-reading loop from article 1.2. Formally the work is moving, but the decisions are being made by someone who does not answer for them. A month later the project holds code nobody has read and a bug nobody can find.

The sign: you cannot explain what a file in your own project does.

The AI becomes the architect

This one looks different: the task is stated broadly — "build me a reminder bot" — and the AI makes every decision for you. Which database, which structure, which libraries, how data is stored. All of it gets locked into the project, and changing it later is expensive.

The problem is not that the decisions are bad. Usually they are fine — average for the industry. The problem is that they are not yours, and you find out about them a month later, when something needs to change.

The sign: the project contains a library and you don't know what it is for.

WHAT TO TELL THE AI

Roles are not set by settings; they are set by how the task is stated. The architect says what to build:

BETTER

```
Storage will be SQLite, not JSON — there will be a lot of
records and we need queries by date. No ORM, we write the
queries by hand, there aren't many.
```

The implementer gets what it is good at:

BETTER

```
Propose a table structure for subscriptions. It needs
user_id, city and creation date. Tell me which indexes
are worth adding and why.
```

And one separate move — asking for your role back, when you feel decisions are being made without you:

PROMPT

```
Which decisions did you make on my behalf in this code?
List them, with the alternatives that existed.
```

WHAT TO LOOK FOR

You haven't argued with the AI in a while. If every proposal is accepted, odds are they are not being read. Normal work includes a few "no, let's do it differently" per session.

You don't know why the project is built that way. Any part you cannot say "I chose this because…" about belongs to someone else.

Tasks are phrased as "make it good". That is how a customer talks, not an architect. An architect names the constraints.

Changes are accepted without reading the diff. The most reliable sign that the reviewer's chair is empty (article 2.13).

CHEAT SHEET

- You are the architect and the reviewer, the AI is the implementer.
- The architect decides what to build; the reviewer, what to accept.
- Code becomes yours when you have read it and agreed.
- Can't explain a file in your own project — you have become the implementer.
- A library whose purpose you don't know — the AI has become the architect.
- Ask: "which decisions did you make on my behalf?"

1.2 The working cycle

task → description → generation → reading → checking → iteration

WHAT IT IS

The working cycle is the order of steps in which a task turns into working code. Six steps, repeated until the result is good enough.

The cycle is worth knowing by heart, because almost every problem with an AI is a skipped step rather than a bad model. Each chapter of

this book takes one of the steps apart in detail, and we will keep coming back to this list.

The six steps

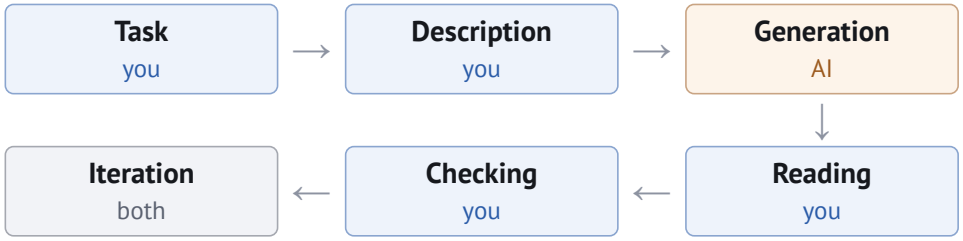


Fig. 1.1 · The working cycle. Blue is what you do, amber is the AI. Four of the six steps are yours, and generation is the shortest of them.

#	Step	What comes out
1	Task	an understanding of what the result should be
2	Description	a prompt or a mini-spec
3	Generation	code
4	Reading	an understanding of what it wrote
5	Checking	a run, a test, an answer: works or not
6	Iteration	the next prompt, or a finished feature

A common misconception is that working with an AI means step 2 and step 3. In reality it is steps 1, 2, 4 and 5 — and generation is the shortest and easiest part.

WHAT TO LOOK FOR

The main mistake looks like this: got the code → didn't understand it → ran it → doesn't work → "doesn't work, fix it" → got the code → didn't understand it → ran it…

This is the **no-reading loop**. It feels like work, because something keeps happening, but no lap adds any understanding. Five laps in, you have a project you don't know and a bug nobody can find.

The only way out of the loop is step 4. You don't have to read every line — it's enough to understand what the code does and where. But the step cannot be skipped.

> **REMEMBER**
> If you cannot explain in two sentences what the code does, it should not stay in the project, even if it works.

WHAT TO TELL THE AI

Each step has its own kind of prompt. These are not separate tricks — they are one cycle, broken into turns.

PROMPT

```
Step 2, description:
Add a /weather command to bot.py. It takes a city name,
requests the weather from api.openweathermap.org and replies
with one line: temperature and conditions. Take the key from
the OWM_API_KEY environment variable. If the city is not
found, reply "City not found".
```

PROMPT

```
Step 4, reading:
Explain line by line what the function you added does.
Separately, tell me what happens if the API returns an error.
```

PROMPT

```
Step 5, checking:
Which three cases should I check by hand to be sure
the command works correctly?
```

The third prompt is underrated. The AI knows perfectly well where its own code is fragile — you only have to ask.

CHEAT SHEET

- Six steps: task → description → generation → reading → checking → iteration.
- Four of the six are yours. Generation is the easiest part.
- The no-reading loop is the main way to lose control of a project.
- Can't explain the code in two sentences — don't keep it.

1.3 What AI does well and what it does badly

and why its tone won't tell you which

WHAT IT IS

An AI has strengths and weaknesses, and they are fairly predictable. Knowing them matters more than it seems: half of all disappointment comes from pointing a task in the wrong direction.

But the main thing in this article is something else. **The tone of an answer has nothing to do with how right the AI is.** It writes a boilerplate function it has seen a thousand times, and invents a library method that does not exist, with exactly the same confidence.

WHAT IT LOOKS LIKE IN CODE

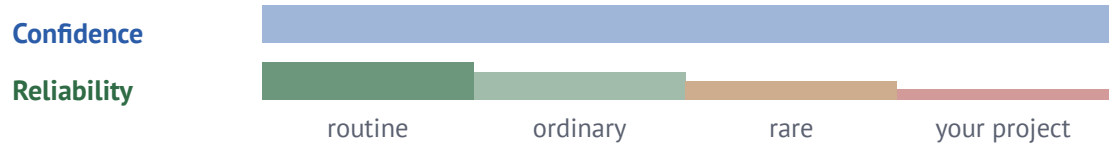


Fig. 1.2 · The AI's confidence does not change; its reliability does. The further a task is from routine, the wider the gap between how an answer sounds and what it is actually worth.

Where the AI is strong

What	Why
Routine code: CRUD, parsing, formats	there was a great deal of it in training
Plumbing and chores: configs, arguments, logging	boring for a human, easy for a machine
Translating between languages	the structure stays, the syntax changes
Explaining someone else's code	a comprehension task, not an invention task
A first draft	faster than a blank page, even if you rewrite it
Finding your own typos	when you point at where to look

What the whole list has in common: the task is common, there is one right answer, and checking it is easy.

Where the AI is weak

What	Why
Your project and your data	it has not seen them (article 3.1, reason 2)
Recent library versions	training holds more old code than new
Rare libraries	few examples — more invention
Decisions with trade-offs	it doesn't know what matters more to you

Anything requiring memory of earlier decisions	a new session has no context
Judging its own work	it defends its own code (article 3.8)

> **CAREFUL**
> Notice what is missing from the list of weaknesses: “hard problems”. Difficulty is not the deciding factor. An AI handles hard routine code beautifully and gets simple but rare things wrong. What decides is how common the task is, not how hard.

Why confidence doesn’t help

An AI does not signal uncertainty — not by tone, not by hedging. An answer about a library it knows cold and an answer about a library it has seen twice sound exactly alike.

Hence the practical consequence: **you decide how much to check, based on the nature of the task, not on how convincing the answer sounds.**

WHAT TO TELL THE AI

Confidence can be pulled out with a direct question — and the AI answers it more honestly than its tone implies:

PROMPT

```
How confident are you in this code? Separately, tell me which
parts you would check first and why.
```

For an unfamiliar library it is worth asking outright:

PROMPT

```
Are you sure session.fetch_all() exists in aiohttp 3.9?
If you are not sure, say so and I will check the docs.
```

This phrasing works: by explicitly allowing “I’m not sure”, you get a far more honest answer than from “does this method exist?”.

WHAT TO LOOK FOR

The more specific the task, the more carefully you read. Routine code can be skimmed; code for a rare library is read line by line.

A confident tone on a rare topic is a reason to tense up, not to relax.

“This is common practice” is not an argument. Common where? Your project may work differently, and the AI does not know that.

A trade-off handed to the AI. “Which is better, SQLite or Postgres?” without your conditions gives you an average answer, not yours.

CHEAT SHEET

- What decides is how common the task is, not how hard.
- Strong at: routine code, chores, translation, explanation, first drafts.
- Weak at: your project, recent versions, rare libraries, trade-offs.
- The tone of an answer is unrelated to how right the AI is.
- You choose how much to check, based on the nature of the task.
- Allow “I’m not sure” and you will hear an honest answer.

1.4 Three skills

understanding, stating, checking

WHAT IT IS

The whole book is built around three skills. This is not a neat diagram but the actual shape of the contents: chapters 2, 3 and 4 *are* the three skills, one chapter each.

Skill	What it means	Chapter
Understanding	reading code, knowing the terms, seeing a project’s structure	2
Stating	framing a task so that what you had in mind comes out	3
Checking	making sure it worked, catching errors, rolling back	4

The other chapters are applications of these three: process and tools (5), architecture (13), the practicum (14), the stack chapters.

WHY THEY ARE NEEDED TOGETHER

The skills are not independent. Each leans on the others, and a gap in one brings down the whole thing.

Without understanding, checking does not work. You cannot be sure code does the right thing without reading it. A run checks exactly one case — the one you ran.

Without stating, understanding is wasted. You can see the solution is wrong but cannot explain which one you need — and the next attempt comes back the same.

Without checking, the first two are useless. Read it, framed it, got the code — and never ran it. Everything before that was for nothing.

That is why the working cycle in article 1.2 looks like a cycle: the three skills are applied in turn on every lap.

WHERE TO START

If you develop them one at a time, the order is this:

Stating first. It pays off fastest: a good prompt saves an hour the first time you use it, and it can be learned in an evening. That is exactly why chapter 3 was written first.

Understanding second. The largest skill and the slowest to acquire, but without it you hit a ceiling: you can improve prompts forever, yet control will not appear until you read the result.

Checking third. It comes quickly once the first two are there. Partly it is just a set of habits: run it, look at the diff, run the tests.

> **TIP**
> If it isn't clear which skill is lagging, look at where you lose time. Redoing things three times — stating. Afraid to touch your own code — understanding. Catching bugs at your users rather than at your desk — checking.

WHAT TO TELL THE AI

The skills can be practised inside ordinary work, by adding one sentence to ordinary tasks.

For understanding:

PROMPT

```
Before I accept this code: explain line by line what it does.
I want to understand it, not just paste it.
```

For stating — feedback on your own prompts:

PROMPT

```
I stated the task like this: <prompt>. What was unclear in
that wording? What did you have to fill in yourself?
```

For checking:

PROMPT

```
Which three cases should I check by hand to be sure
this works? Include at least one failing case.
```

The second prompt is especially useful. It turns every failed attempt into a lesson: instead of irritation you get a concrete list of what was missing.

WHAT TO LOOK FOR

Leaning too far into stating. It looks like collecting tricks: ever newer ways to ask, while the result still goes unread. The ceiling arrives fast.

Leaning too far into understanding. Picking apart every line, including the ones you never meant to change. Reading is a means, not an end; enough is enough (article 2.1).

Checking postponed "until later". The most expensive imbalance: errors pile up and are all found at once, exactly when they are hardest to find.

CHEAT SHEET

- Three skills: understand · state · check. Chapters 2, 3, 4.
- Without understanding there is no checking; without checking the first two are useless.
- Start with stating — it pays off fastest.
- Understanding is the slowest skill and the most important.
- Redoing it three times — stating. Afraid to touch the code — understanding.
- Ask: "what did you have to fill in from my wording?"

1.5 The cost of a mistake

where you can trust, and where you cannot

WHAT IT IS

Checking everything equally carefully is impossible, and unnecessary. The question is what to decide by: where to skim and where to read line by line.

The common answer — "by how hard the task is" — is wrong. The right criterion is different: **what the mistake costs if it happens.** And that cost is made of two things: how easy it is to roll back, and how quickly you would notice.

HOW TO WORK IT OUT

Reversibility
How simple it is to put things back.

Easy to roll back	Hard to roll back
a code change under git	a migration that altered the data
a new function nobody calls yet	messages already sent to users
a local experiment	deleted files

a change covered by tests	a published release people have downloaded

Git makes almost anything involving code reversible — which is why it gets an article of its own (5.1). What is irreversible is what has left the repository: data, mail, money, publications.

Visibility

How quickly the mistake announces itself.

You'll notice at once	You might not notice for months
it won't start	wrong numbers in a report
it crashes with an error	records lost during an import
the button does nothing	the wrong time zone
tests go red	a leak that only shows under load

The right column is the dangerous one. A mistake that crashes the program is honest: it reports itself immediately. A mistake that quietly corrupts data surfaces once everything is already corrupted.

Together

	Notice at once	Might not notice
Easy to roll back	skim: run it and look	carefully: read the logic
Hard to roll back	carefully: read the diff	line by line + plan + tests

The bottom-right corner is migrations, mailings, money and other people's data. There you need all of it: a plan (article 3.10), a mini-spec (3.15), tests (3.9) and a read of the diff (2.13). The top-left is what you can simply run.

> **REMEMBER**
> The dangerous thing is not the complex one, it is the invisible one. Code that crashes you will fix today. Code that quietly rounds the wrong way you will be hunting six months from now, after a user complains.

WHAT TO TELL THE AI

The assessment can be handed to the AI, and it does it reasonably well:

PROMPT

```
Before you start: what in this task is irreversible,
and what could break without being noticed? Briefly.
```

For anything that changes data there is a separate question:

PROMPT

```
Does this change modify existing data? If so, how do I roll
it back if it turns out to be wrong?
```

If there is no answer to the second question, the task cannot be started: the way back comes first, the change second.

PROMPT

```
Make the migration reversible: keep the old file instead of
deleting it. Rename rather than delete.
```

WHAT TO LOOK FOR

Irreversible work done without a rehearsal. Migrations, bulk edits, mailings — all of these are first run against a copy. A no-write mode (`--dry-run`) that only prints what it was about to do costs one sentence to ask for and saves you regularly.

Deleting instead of renaming. `rm` and `DROP` are irreversible, renaming is not. It is the cheapest way to make an operation reversible.

No way to notice. If a change has no test, no log and no visible effect, you will be the last to learn that it was wrong.

Trust out of habit. Five good changes in a row do not make the sixth safe. The level of checking is set by the task, not by the previous run of luck.

CHEAT SHEET

- Checking is tuned to the cost of a mistake, not to difficulty.
- Cost = reversibility × visibility.
- The invisible is the dangerous one: it surfaces once everything is spoiled.
- Bottom-right corner (irreversible + invisible) — plan, spec, tests, diff.
- No way to roll back — don’t start; invent the way back first.
- Rename instead of deleting.

1.6 The illusion of finished code

why “it works” does not mean “it’s right

WHAT IT IS

Code from an AI looks finished. Even indentation, sensible names, type annotations, tidy comments. It looks like code somebody thought about.

That impression is deceptive, and it is deceptive for three reasons. They are worth taking apart at the start of the book, because everything that follows is built on not falling for them.

THREE ILLUSIONS

Illusion one: it looks tidy

How neat code is says nothing about whether it is right. A person who writes sloppily usually writes sloppily everywhere, so the look of the code told you something. With an AI that link is gone: it formats a correct solution and a wrong one with exactly the same care.

Worse, tidiness works against you — it lulls. Code with crooked indentation gets read carefully; code that looks like a textbook gets scrolled past.

Illusion two: it ran

A run checks exactly one case — the one you ran. Usually that is the happy path: the city was found, the data arrived, the user typed what was expected.

```
temp = data.get("main", {}).get("temp", 0)
```

This line will always run. On correct data it gives the right answer; on a server error it gives “It is currently 0 degrees”. The program works. The program lies.

That is exactly why this book pays so much attention to the unhappy cases: empty data, dropped connections, garbage input. That is where the errors live that a run does not find.

Illusion three: the AI sounds sure

The tone of an answer is unrelated to how right the AI is (article 1.3). It does not say “I’m not sure”, does not hedge, and does not mark the places where it was guessing. An explanation of a method that does not exist sounds every bit as convincing as a description of one it knows cold.

> **REMEMBER**
> Three signs you cannot trust: it is beautifully formatted, it ran, it sounds confident. None of them speaks to correctness — all three speak only to how it looks.

WHAT TO USE INSTEAD

The book proposes replacing the feeling of "seems fine" with four concrete questions. They are the same questions as in article 2.1, and they close all three illusions:

1. **What does this function take and what does it return?**
2. **Does it change anything outside itself?**
3. **What happens if the data is not what was expected?**
4. **Where is it called from?**

The third question is the antidote to the second illusion. It forces you to think about the cases you did not run.

WHAT TO TELL THE AI

The illusion of confidence is broken by a direct question:

PROMPT
```
Name three places in this code where it will break
on unexpected data. If there are none, explain why not.
```

The illusion of the run — by a question about what went unchecked:

PROMPT
```
I ran it and got the right answer. Which cases did
my run not check?
```

And the illusion of tidiness — by asking for the trade-offs:

PROMPT
```
Which decisions in this code are debatable? What did you
choose without thinking, because it is the usual way?
```

The last question often pulls out the most interesting material: decisions the AI made out of habit rather than because they fit your case.

WHAT TO LOOK FOR

You accepted the code without asking a single question. Normal work includes a few clarifications on every non-trivial change.

Checking came down to running it. Ran it, answer looked right, accepted. That checks one scenario out of ten.

Comments instead of code. `# TODO: handle the empty list` is a hole, dressed up to look like it was considered (article 2.14).

The feeling that it all went too smoothly. If the task was non-trivial and the solution arrived without a single caveat or question, part of the difficulty has probably just gone unnoticed.

CHEAT SHEET

- Tidy formatting says nothing about correctness and dulls your attention.
- A run checks one case — usually the happy one.
- A confident tone is unrelated to whether the AI is right.
- Instead of a feeling, four questions: takes · returns · changes · called from.
- Ask: “which cases did my run not check?”
- Too smooth on a hard task — the difficulty went unnoticed.

2 Reading code

The central skill of the whole book. Until you can read what the AI wrote, you can neither check it nor argue with it. Reading does not mean being able to write the same thing from scratch.

2.1 Why read code if the AI writes it

reading and being able to write are different skills

WHAT IT IS

The central misunderstanding of this chapter sounds like this: "if the AI writes the code, reading it is optional." It isn't — and here is why.

Reading and writing are different skills, and reading is far easier. To write a function from scratch you have to hold the syntax, the library and the order of operations in your head. To read a finished one, it is enough to understand what each line does. The second is learned several times faster and gives you almost all of the benefit.

WHY IT MATTERS IN PRACTICE

Without reading, three things from the working cycle stop working.

Checking. You cannot be sure the code does what is needed if you don't understand what it does. A run only checks the case you ran.

Arguing. When the AI proposes something you don't like, you need substantive grounds. "I feel like something's off here" it will bat away; "this function overwrites the input list, and that list is used later" it will not.

Ownership. A month from now somebody — most likely you — will open this file. If nobody read the code while it was being written, you will be working it out from scratch, and at the worst possible moment: when something has broken.

> **REMEMBER**
> You are answerable for code you did not read exactly as much as for code you wrote yourself. The AI's authorship changes nothing: it is you who will answer to the users, to the server, and to yourself a month from now.

HOW MUCH READING IS ENOUGH

You do not need to understand every character. You need to be able to answer four questions:

Question	Why
What does this function take and return?	to know how to use it
Does it change anything outside itself?	to predict the consequences
What happens if the data is not what was expected?	to find holes before users do
Where is it called from?	to know what breaks when you change it

If all four have answers, the code has been read well enough. Working out exactly how the algorithm inside is implemented is usually unnecessary.

WHAT TO TELL THE AI

Reading does not have to be done alone. An AI explains its own code well, if you ask properly (article 3.7):

> **PROMPT**
> ```
> Explain line by line what this function does. Separately:
> what it takes, what it returns, and what it changes outside
> itself.
> ```

The reverse is useful too — read it yourself and check with the AI:

> **PROMPT**
> ```
> Here is how I read it: the function takes a list of orders,
> sums them up and writes the result to the log. If the sum is
> zero it writes nothing. Did I miss anything?
> ```

The second way teaches faster: you think first, then compare. Mistakes in your understanding surface immediately, and they stick.

WHAT TO LOOK FOR

“Looks right” is not reading. Code written by an AI almost always looks right: even indentation, clear names, tidy structure. Outward neatness says nothing about what it does.

Long code does not have to be read in full. Start with the function signatures and the entry point. If those make the connections clear, you can leave the insides alone until you need them.

An unfamiliar construct is not a reason to give up. It is perfectly normal not to recognise something: a decorator, a generator, `async with`. Ask what it is — this chapter has an article for each of them.

CHEAT SHEET

- Reading and writing are different skills. Reading is learned faster.
- Without reading, checking, arguing and ownership all stop working.
- Enough is: what it takes · what it returns · what it changes outside · where it is called.
- “Looks right” is not the same as “has been read”.
- Read it yourself, check with the AI — that is the faster way to learn.

2.2 Anatomy of a file

imports, constants, definitions, entry point

WHAT IT IS

Almost any code file is laid out the same way: at the top what it needs to work, in the middle the definitions, at the bottom the run. Knowing these four zones, you can find your way around an unfamiliar file in half a minute without reading it through.

The order is not accidental: it matches the order in which the code executes when the file is imported.

WHAT IT LOOKS LIKE IN CODE

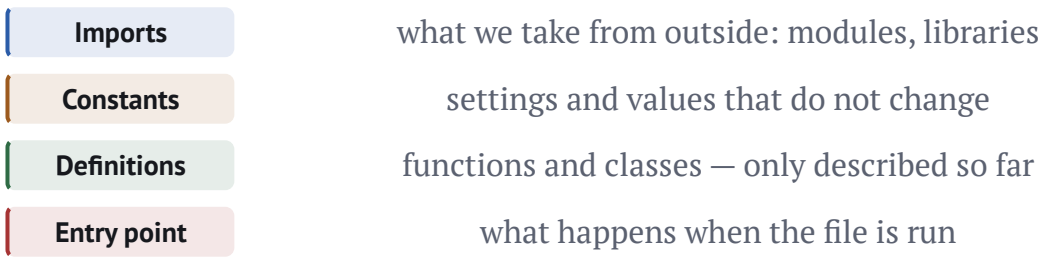


Fig. 2.1 · The four zones of a file. Top to bottom, in the same order the code executes when it runs.

Here is a whole file — all four zones in order:

```
"""Reminder bot. The whole thing in one file."""
import os
import sqlite3

DB_PATH = os.environ.get("DB_PATH", "reminders.db")
MAX_TEXT = 200


def get_conn() -> sqlite3.Connection:
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    return conn


def add_reminder(user_id: int, text: str) -> int:
    if len(text) > MAX_TEXT:
        raise ValueError(f"text is longer than {MAX_TEXT} characters")
    with get_conn() as conn:
        cur = conn.execute(
            "INSERT INTO reminders (user_id, text) VALUES (?, ?)",
            (user_id, text),
        )
        return cur.lastrowid


def main() -> None:
    ...


if __name__ == "__main__":
    main()
```

What to read in each zone

Imports answer the question “what does this file depend on”. `import sqlite3` — it works with a database. See `import requests` and the file goes out to the network. This is the first thing worth reading: the imports tell you what the file is in the business of doing.

Constants are settings lifted to the top so they are easy to find. `DB_PATH = os.environ.get("DB_PATH", "reminders.db")` reads as: take the value from the environment variable, and if it isn’t there, use `reminders.db`.

Definitions are the longest zone and the most harmless. The key point: `def` only **describes** a function, it does not run it. The file can be imported and not a single line inside a `def` will execute.

The entry point is the `if __name__ == "__main__":` line. It reads as: "if this file was run directly rather than imported, do what follows."

> **TIP**
> Start reading an unfamiliar file from the end. The entry point shows what actually happens on a run, and from there you can see which functions matter and which are helpers.

WHAT TO TELL THE AI

PROMPT

```
Break this file down by zones: what is in the imports, which
constants there are, which functions are defined and what runs
on startup. Briefly, one line per item.
```

For an unfamiliar project a different question is more useful:

PROMPT

```
Which function in this file should I start reading to
understand what it does? Don't explain the whole file,
just say where to start and why.
```

WHAT TO LOOK FOR

Code outside functions. Anything written at the top level that is not an import, a constant or a definition will execute when the file is imported. The line `conn = sqlite3.connect(DB_PATH)` outside a function means the database opens simply because something imported the file — and that is almost always a mistake.

No entry point in a file meant to be run. If `if __name__ == "__main__":` is missing but the file is supposed to be runnable, somewhere near the bottom there will be a bare call — and it fires on import too.

Constants that aren't constant. A name in capitals promises the value does not change. If the AI wrote `MAX_TEXT = 200` and then `MAX_TEXT = get_limit()` somewhere, it misleads everyone, including the next AI.

Secrets in constants. `TOKEN = "123456:ABC-DEF"` right there in the file means the token goes into git and stays in the history forever. The right way is an environment variable (article 5.7).

CHEAT SHEET

- Four zones: imports · constants · definitions · entry point.
- Read the imports first: they tell you what the file is about.
- `def` only describes a function, it does not execute it.
- `if __name__ == "__main__":` — "if the file was run directly".
- Read an unfamiliar file from the end, starting at the entry point.
- Code outside functions runs on import — almost always a mistake.

2.3 Anatomy of a project

where things live and what you can leave closed

WHAT IT IS

A project is dozens of files, but not all of them need reading. Most are either housekeeping or generated by tools. This article teaches you to tell one from the other, and to find where execution begins.

You need the skill every time you open someone else's project — or your own, forgotten one — and every time you have to tell the AI which file to work in.

WHAT IT LOOKS LIKE IN CODE

```
weather-bot/
├── .git/                history, don't touch by hand
├── .venv/               installed libraries
├── .env                 secrets: tokens, passwords
├── .gitignore           what never goes into git
├── requirements.txt     list of dependencies
├── README.md            where to start
├── bot.py               entry point
├── config.py            reading settings
├── db.py                database connection
├── storage.py           database queries
├── handlers/            command handlers
│   ├── __init__.py
│   ├── start.py
│   └── weather.py
├── migrations/          database structure changes
└── tests/               tests
```

What to read and what not to

Read	Don't read
the entry point (`bot.py`, `main.py`, `app.py`)	`.venv/`, `node_modules/`, `vendor/`
`README.md`	`.git/`

the logic folders (`handlers/`, `services/`)	lock files: `poetry.lock`, `package-lock.json`
`requirements.txt` — what the project uses	generated output: `dist/`, `build/`, `__pycache__/`

The rule is simple: **if a file was created by a tool rather than a person, you don't need to read it.** A `.venv` folder may hold hundreds of thousands of lines; not one of them is yours.

How to find the entry point

Three ways, fastest first:

1. **By name.** `main.py`, `bot.py`, `app.py`, `manage.py`, `index.js`, `main.dart` — nine times out of ten that's it.
2. **By README.** It usually gives the run command, and the command names the file.
3. **By search.** `if __name__ == "__main__"` for Python, `"scripts"` in `package.json` for JavaScript.

Names that mean something

Some folder names are stable and turn up across many projects:

Name	What's inside
`handlers/`, `routes/`, `views/`	what happens in response to a user action
`models/`, `schemas/`	the shape of the data: which fields, which types
`services/`, `core/`	the main logic, not tied to any interface
`utils/`, `helpers/`	small helper functions
`migrations/`	database structure changes, applied in order
`tests/`	tests
`static/`, `assets/`	images, styles, fonts

`utils/` is a special case: it is where everything that found no home gets dumped. If it has grown large, that is a sign the structure needs revisiting.

> **TIP**
> An `__init__.py` in a Python folder is usually empty and exists only so the folder counts as a package. If it isn't empty, take a look: it may hold imports that run whenever the package is touched.

WHAT TO TELL THE AI

PROMPT

```
Show the project file tree without .venv, .git and __pycache__.
For each top-level folder, one line saying what is in it.
```

When you need to know where to begin:

PROMPT

```
I am opening this project for the first time. Name three files
I should read to understand how it is built, in order.
```

And the most practical one — before stating a task:

PROMPT

```
I need to add a /stats command. Which files would have to
change? Don't write anything yet.
```

The answer to the last one doubles as a check on whether the AI understood the project structure correctly.

WHAT TO LOOK FOR

A `.env` file in git. Checked in one line: is `.env` in `.gitignore`? If not, the tokens are already in the repository history.

Logic in the entry point. If `bot.py` is five hundred lines, there is no separation, and the project will be hard to change. That is bad for the AI too: it cannot work on one part without disturbing the rest.

Two folders for one job. `utils/` and `helpers/`, or `services/` and `core/`, in the same project is a sign code was filed away without looking. Ask the AI how they differ: if it cannot explain, they should be merged.

An empty `tests/`. The folder exists, the tests do not. Which means nobody will know what a change broke (article 3.9).

CHEAT SHEET

- Read: the entry point, the README, the logic folders, the dependency list.
- Don’t read: anything created by a tool rather than a person.
- Find the entry point by name, by README, or by `if __name__`.
- Stable names: `handlers/`, `models/`, `services/`, `migrations/`.
- Before a task, ask the AI which files it will touch.
- `.env` must be in `.gitignore`.

2.4 Variables, types, data structures

four containers that almost everything is built from

WHAT IT IS

Almost all data in a program sits in one of four structures: a dictionary, a list, a tuple or a set. Knowing how they differ makes reading code much faster — the shape of the brackets alone tells you what is going on.

We will also cover type annotations (`text: str`) separately: they appear constantly in AI-written code and read more easily than they look.

WHAT IT LOOKS LIKE IN CODE

```
user = {"id": 7, "name": "Ola", "cities": ["Tønsberg", "Oslo"]}

print(user["name"])
print(user.get("email"))
print(user.get("email", "not set"))

cities = user["cities"]
cities.append("Bergen")
print(user["cities"])          # the original changed too

point = (59.27, 10.41)
lat, lon = point
print(lat, lon)

seen = {"Oslo", "Bergen", "Oslo"}
print(sorted(seen), len(seen))
```

Output:

```
Ola
None
not set
['Tønsberg', 'Oslo', 'Bergen']
59.27 10.41
['Bergen', 'Oslo'] 2
```

The four structures

Kind	Brackets	What for	Sign in code
Dictionary	`{"key": value}`	data with named fields	access by `x["name"]`
List	`[1, 2, 3]`	an ordered collection	access by index `x[0]`, `.append()`
Tuple	`(1, 2)`	a set of values nobody changes	unpacking `a, b = x`

Set	`{1, 2, 3}`	unique values	a check like `if x in seen`

Curly braces mean both a dictionary and a set — easy to tell apart: if there are colons inside, it's a dictionary.

Two lines worth noticing

```
print(user.get("email"))           # None
print(user.get("email", "not set"))
```

`user["email"]` raises `KeyError` when the key is missing. `user.get("email")` quietly returns `None`. The difference matters and comes up constantly in AI-written code: `.get()` is a favourite way to "fix" an error by hiding it (article 2.14).

```
cities = user["cities"]
cities.append("Bergen")
print(user["cities"])   # ['Tønsberg', 'Oslo', 'Bergen']
```

`cities` is not a copy, it is the same thing. Lists and dictionaries are passed by reference: change one and both change. This is the source of a whole class of bugs that look like magic.

Type annotations

```
def add_reminder(user_id: int, text: str) -> int:
```

Read left to right: `user_id` is an integer, `text` is a string, the function returns an integer. This is a hint for the reader, not a check: Python does not enforce annotations, and nothing stops you passing a string where a number was promised.

The ones you will meet most often:

Written	Means
`str`, `int`, `float`, `bool`	string, integer, float, true/false
`list[str]`	a list of strings
`dict[str, int]`	a dictionary: string keys, integer values
`str | None`	a string **or** nothing
`-> None`	the function returns nothing

`str | None` is especially useful when reading: it tells you outright that the result has to be checked.

WHAT TO TELL THE AI

PROMPT
```
Which data structures are used in this file and what is in
each? Briefly, one line per variable.
```

A useful question when data comes from outside:

PROMPT
```
Show me an example of the data that will reach this function.
One happy case and one with missing fields.
```

When you need to know whether something changes unexpectedly:

PROMPT
```
Does this function modify the list it was given, or build
a new one?
```

WHAT TO LOOK FOR

`.get()` where an error is wanted. If a field is required, its absence is an error, not a reason to substitute `None`. A silent `None` surfaces later and further from the cause.

Modifying what was passed in. A function received a list and appended to it — the caller's list changed too. If that wasn't intended, hard bugs follow.

A dictionary instead of a proper structure. `data["result"]["items"][0]["price"]` is a sign nobody parsed the data. Every link in that chain can fail, and the traceback will only show the last one.

Annotations that don't match the code. `-> str` while the function sometimes returns `None`. Annotations check nothing, so they can drift away from the code unnoticed.

CHEAT SHEET

- Four structures: dictionary · list · tuple · set.
- Curly braces with colons — a dictionary; without — a set.
- `x["k"]` fails on a missing key, `x.get("k")` returns `None`.
- Lists and dictionaries are passed by reference: change one, both change.
- `str | None` in an annotation — the result must be checked.
- Annotations check nothing; they are for the reader.

2.5 Functions

in, out, and what happens on the side

WHAT IT IS

To read a function you do not have to take its insides apart. It is enough to answer three questions: what comes in, what goes out, and what changes outside.

The third question is the most important and the most often skipped. A function can return nothing at all and still turn half the program upside down.

WHAT IT LOOKS LIKE IN CODE



Fig. 2.2 · A function is read along three directions. Side effects are what happens in addition — and what people usually forget to check.

```python
# Pure function: same input -> same output, nothing else
def format_price(amount: float, currency: str = "NOK") -> str:
    return f"{amount:.2f} {currency}"


# Side effect: changes what it was given
def add_tax(items: list[dict], rate: float) -> None:
    for item in items:
        item["price"] = round(item["price"] * (1 + rate), 2)


# Side effect: writes to the outside world
def log_order(order_id: int) -> None:
    with open("orders.log", "a", encoding="utf-8") as f:
        f.write(f"order {order_id}\n")
```

A pure function

`format_price` takes a number and a string and returns a string. Call it a hundred times with the same arguments and you get the same thing a hundred times, and nothing else happens.

Functions like this are the easiest to check: compare the input with the output and you are done.

A function with a side effect

`add_tax` returns `None`, but it does do work — it changes the dictionaries in the list it was given. Here is what follows from that:

```
cart = [{"name": "coffee", "price": 100.0}]
add_tax(cart, 0.25)
print(cart)     # [{'name': 'coffee', 'price': 125.0}]
add_tax(cart, 0.25)
print(cart)     # [{'name': 'coffee', 'price': 156.25}]
```

The second call charged tax on the tax. The function is not protected against being called twice, and it will not tell you so — not by its name, not by its signature.

> **CAREFUL**
> `-> None` in a signature is a reason to tense up, not to relax. A function that returns nothing must be doing something outside itself — otherwise why call it?

Where to look for side effects

Run your eye down the body of the function and look for four things:

Sign	What is happening
assigning into something that arrived as an argument	changes the caller's data
`open()`, writing to a file	changes the disk
`conn.execute()`, `INSERT`, `UPDATE`	changes the database
`requests`, `await session.post()`	changes something on another server
`global`, assigning to a module variable	changes the program's state

If none of these is there and a `return` is, the function is pure and life is simpler.

Default arguments

```
def format_price(amount: float, currency: str = "NOK") -> str:
```

`currency` may be omitted, and then it is `"NOK"`. Easy to read, but there is a trap the AI reproduces regularly:

```
def add_item(item, cart=[]):    # don't do this
    cart.append(item)
    return cart
```

A list in a default value is created **once** — and stays shared across every call. The second `add_item("bread")` returns a list of two items. The right way is `cart=None`, and `if cart is None: cart = []` inside.

WHAT TO TELL THE AI

PROMPT

```
For this function answer three questions:
what it takes, what it returns, what it changes outside itself.
One line per question.
```

PROMPT

```
What happens if I call this function twice in a row
with the same arguments?
```

The second question catches exactly the class of bug in the tax example.

WHAT TO LOOK FOR

A function that both returns and changes. It does two jobs at once, which makes it hard to reuse and easy to call at the wrong moment. Usually worth splitting.

The name promises the wrong thing. `get_user` should only get. If a database record is created inside `get_user`, the name is lying, and the next reader will trip over it.

A mutable default value. `=[]` or `={}` in the arguments is almost always a bug.

Too many arguments. More than five is a sign the function is doing several jobs. Check whether it should be split.

CHEAT SHEET

- Three questions: what it takes · what it returns · what it changes outside.
- Pure function: same input, same output, and nothing else.
- `-> None` means all the work is in the side effects.
- Look for side effects in: writes to arguments, files, the database, the network, `global`.
- `=[]` and `={}` as default arguments are a bug.
- Ask: “what happens if I call it twice in a row?”

2.6 Conditions and loops

reading the flow of control

WHAT IT IS

Control flow is the order in which lines execute. Top to bottom by default, but conditions and loops change that: some lines are skipped, some are repeated.

To read code like this you need to be able to answer one question: **on what data does this line execute?** The rest follows.

WHAT IT LOOKS LIKE IN CODE

```
orders = [
    {"id": 1, "status": "paid", "total": 250},
    {"id": 2, "status": "pending", "total": 0},
    {"id": 3, "status": "paid", "total": 90},
]

total = 0
for order in orders:
    if order["status"] != "paid":
        continue
    if order["total"] <= 0:
        print(f"order {order['id']}: paid, but the amount is zero")
        continue
    total += order["total"]
print("total:", total)
```

Output:

```
total: 340
```

How to read it

Indentation is scope. Everything shifted right of a `for` or an `if` belongs to it. In Python indentation is not formatting but syntax: move a line and you change the meaning of the program.

`continue` means "skip the rest and take the next item". The two of them in the example are filters: unpaid orders and zero-value orders never reach the addition.

`break` means "leave the loop entirely". Confusing the two is a common reading error: `continue` skips one step, `break` ends the iteration.

A loop's `else` is a rare construct, but it does turn up. It runs if the loop finished without a `break`.

List comprehensions

The same meaning fits on one line:

```
paid = [o["total"] for o in orders if o["status"] == "paid"]
print(sum(paid))     # 340
```

It reads, in meaning, right to left: *take* `o` *from* `orders` *where the status is* `paid`*, and collect their* `o["total"]`. The AI writes these readily — they are short, but with two or three conditions they become unreadable.

> **TIP**
> If you keep stumbling over a comprehension, ask for it to be rewritten as a plain loop: "rewrite this list comprehension as a for loop so the order of operations is visible." The meaning stays the same and it reads more easily.

The ternary expression

```
status = "paid" if order["total"] > 0 else "pending"
```

Reads as: `status` equals `"paid"` if the condition holds, otherwise `"pending"`. The same construct in JavaScript and Dart is written differently:

```
const status = order.total > 0 ? "paid" : "pending";
```

The `?` before the first option, the `:` before the second — same thing, other syntax.

WHAT TO TELL THE AI

> **PROMPT**
> On what data does line 14 execute? Give a concrete example
> of input that gets us there.

> **PROMPT**
> Can this loop run zero times? On what data?

The second question matters more than it looks. A loop over an empty list simply does nothing — silently, with no error. If the code below expects the variable to have accumulated something, it gets zero or emptiness.

WHAT TO LOOK FOR

Deep nesting. Three or more levels of `if` inside `for` inside `if` is hard to read and easy to break. A sign that some conditions could become an early `continue` or move into a function of their own.

A loop that might not run. An empty list, an empty API response — and the whole block is skipped. Check that the code below accounts for that.

Modifying a list while iterating it. Removing items from the same list the `for` is walking gives you skipped items. It looks harmless and works incorrectly.

A condition that is always true or always false. This happens after edits: `if status != "paid" or status != "pending"` is true for every value. Substituting any value is enough to spot it.

CHEAT SHEET

- Indentation in Python is syntax, not formatting.
- `continue` skips a step, `break` leaves the loop.
- A comprehension reads like a loop: take, filter, collect.
- The main question: on what data does this line execute?
- The second question: can the loop run zero times?
- Don't modify a list while you are looping over it.

2.7 Classes and objects

what they are and when they are really needed

WHAT IT IS

A class is a description of how an object is built: what data it holds and what it can do. An object is a concrete instance made from that description.

To read code it is enough to understand four things: `class`, `__init__`, `self` and the dot. Everything else is built on top of those.

WHAT IT LOOKS LIKE IN CODE

```
class Cart:
    def __init__(self, user_id: int) -> None:
        self.user_id = user_id
        self.items: list[dict] = []

    def add(self, name: str, price: float) -> None:
        self.items.append({"name": name, "price": price})

    def total(self) -> float:
        return sum(item["price"] for item in self.items)


cart = Cart(user_id=7)
cart.add("coffee", 100.0)
cart.add("bread", 45.0)
print(cart.total())      # 145.0
print(cart.user_id)      # 7
```

The four things to know

`class Cart:` is a description. Like `def`, on its own it runs nothing.

`__init__` runs at the moment the object is created. The line `cart = Cart(user_id=7)` calls exactly that. This is where the starting data is set.

`self` is the object itself. It is the first argument of every method, but you don't pass it when calling: `cart.add("coffee", 100.0)` has two arguments although the definition has three. Python supplies `self` for you.

The dot accesses the object's data or methods: `cart.items`, `cart.total()`. With parentheses it is a method call; without, it is reading data.

What makes a class different from a pile of functions

A class keeps state between calls. `cart.items` lives in the object: add an item and it stays there. With plain functions you would have to pass that list back and forth by hand.

Hence the rule for reading: **find where the state changes.** Everything starting with `self.` and standing to the left of an `=` is a change to the object. These are the side effects of article 2.5, just inside a class.

Dataclass — a class for data only

```python
from dataclasses import dataclass

@dataclass
class Item:
    name: str
    price: float
    qty: int = 1


item = Item("coffee", 100.0)
print(item)              # Item(name='coffee', price=100.0, qty=1)
```

`@dataclass` writes `__init__` and a readable printout for you. It appears often, and that is a good sign: a class with no methods, only data, reads easily.

> **TIP**
> When the AI proposes a class, ask: "could the same thing be done with functions?" Often the answer is "yes, and it would be simpler". A class earns its place when there is state that lives between calls; without state it only adds layers.

Inheritance

```
class PremiumCart(Cart):
    def total(self) -> float:
        return super().total() * 0.9
```

`PremiumCart(Cart)` means "the same as `Cart`, but with changes". `super().total()` calls the parent's version.

For reading, inheritance is awkward because the code spreads across two places: to understand what `PremiumCart` does you have to open `Cart` too. Which is why more than two levels deep is almost always evil.

WHAT TO TELL THE AI

PROMPT
```
What does this object keep between calls, and which methods
change that state?
```

PROMPT
```
Is this class justified, or would two functions be simpler?
Answer straight.
```

When a class is inherited:

PROMPT
```
Which methods does PremiumCart override and which does it
take from the parent? A list.
```

WHAT TO LOOK FOR

A class with no state. If `__init__` saves nothing and every method could be a plain function, the class is superfluous. The AI creates these readily — they look "proper".

`self.` assigned outside `__init__`. A field that appears in the middle of another method is easy to miss while reading: the object suddenly acquires data it did not have before.

Deep inheritance. Three levels or more and understanding one line means opening four files.

A mutable value in a class field. Same as with default arguments (article 2.5): a list declared at class level rather than in `__init__` is shared by every object.

CHEAT SHEET

- Four things: `class` · `__init__` · `self` · the dot.
- `__init__` runs when the object is created.
- You don't pass `self` when calling; Python supplies it.
- A class earns its place through state between calls; without it, functions are enough.
- Look for `self.x = ...` — those are changes to the object.
- Inheritance deeper than two levels almost always hurts reading.

2.8 Exceptions and error handling

where an error is raised and where it is caught

WHAT IT IS

An exception is a way of saying that work cannot continue. A `raise` line raises an error, a `try/except` block catches it. Between the two, execution jumps over all the code in between.

This mechanism is read through two questions: **where the error is raised** and **where it is caught**. There may be a dozen functions between them — which is exactly what a traceback shows (article 2.12).

WHAT IT LOOKS LIKE IN CODE

```
def parse_port(raw: str) -> int:
    try:
        port = int(raw)
    except ValueError:
        raise ValueError(f"port must be a number, got: {raw!r}") from
None
    if not 1 <= port <= 65535:
        raise ValueError(f"port out of range: {port}")
    return port


for raw in ["8080", "abc", "99999"]:
    try:
        print(raw, "->", parse_port(raw))
    except ValueError as e:
        print(raw, "-> error:", e)
```

Output:

```
8080 -> 8080
abc -> error: port must be a number, got: 'abc'
99999 -> error: port out of range: 99999
```

How to read it

`try:` — "attempt this". If something inside fails, execution moves to `except` immediately and the rest of the `try` is skipped.

`except ValueError:` catches only errors of that kind. Others fly past and go further up.

`raise` raises an error. Here the function turns the opaque `invalid literal for int()` into a clear message that includes the value that didn't fit.

`as e` puts the error object into a variable so the text can be read.

`finally:` runs either way, error or not. It is for closing files and connections.

The hierarchy of errors

Errors form a tree: `ValueError` and `KeyError` are special cases of `Exception`. So `except Exception` catches almost everything, while `except ValueError` catches only its own.

Kind	When it happens
`ValueError`	the value is the wrong shape: `int("abc")`
`KeyError`	no such key in the dictionary
`TypeError`	the wrong type: a string added to a number
`AttributeError`	no such field or method (often on `None`)
`FileNotFoundError`	the file is not there
`TimeoutError`	the answer never came

These six cover most of what you will meet while reading.

> **CAREFUL**
> `except Exception: pass` is the most harmful line in Python. It catches everything and says nothing: the program carries on with wrong data and the cause surfaces nowhere. When you see it in AI-written code, ask which error it actually expected here.

Catch it where you know what to do

The rule for checking where `try` belongs: catch an error where there is a sensible answer to "and now what?".

In the example, `parse_port` does not catch its own range error — it has no idea what to do with a bad port. The calling code does: print a message and move on to the next value.

Which is why a `try` wrapped around every line is a sign the author never thought about who would react to the error, or how.

WHAT TO TELL THE AI

PROMPT

```
Which exceptions can this function raise, and where are they
caught? If nowhere, say so.
```

PROMPT

```
There is an except Exception here. Which specific errors
did you expect to catch? Narrow it down to those.
```

The second prompt is a working way to fix over-broad `except` clauses: the AI can usually name the specific kinds, if asked.

WHAT TO LOOK FOR

`except Exception: pass` — the error is swallowed whole. The code doesn't crash and therefore looks like it works.

`except` with no type. A bare `except:` catches even a Ctrl+C interrupt.

An error caught but not recorded. If the `except` has neither a log nor a re-`raise`, the information is lost forever.

A message without the data. `raise ValueError("invalid value")` — which one? A good message always contains the thing that didn't fit.

A `try` wrapped around twenty lines. It is unclear which of them can fail. Narrowing the `try` to one or two lines is the cheapest way to make the code readable.

CHEAT SHEET

- `raise` raises, `try/except` catches, execution jumps between them.
- `except ValueError` catches only its own, `except Exception` almost everything.
- Six common errors: Value · Key · Type · Attribute · FileNotFound · Timeout.
- Catch where you know what to do next.
- `except Exception: pass` — the error is swallowed and the code lies.
- An error message should always name the value that didn't fit.

2.9 Decorators and "magic"

why code does not execute top to bottom

WHAT IT IS

A decorator is a line with an `@` above a function. It means: this function is wrapped in another one, and on a call the wrapper runs first.

You need to know this for one simple reason: **because of decorators, code does not execute in the order it is written.** For anyone reading code, that is the main source of bewilderment.

WHAT IT LOOKS LIKE IN CODE

```
import functools
import time


def timed(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        t = time.perf_counter()
        result = fn(*args, **kwargs)
        print(f"{fn.__name__}: {time.perf_counter() - t:.3f} s")
        return result
    return wrapper


@timed
def slow_sum(n: int) -> int:
    return sum(range(n))


print(slow_sum(1_000_000))
```

Output:

```
slow_sum: 0.011 s
499999500000
```

The timing message printed **before** the result, although in the code the `print` comes after. That is how the wrapper works: the call `slow_sum(...)` actually calls `wrapper`, which runs the real function, measures the time, prints, and only then returns the result.

How to read it

`@name` **above a function** — the function has been wrapped. To know what happens on a call you have to look at `name`, not only at the function itself.

`*args, **kwargs` means “any arguments at all”. The wrapper does not know what the original function takes, so it takes everything and passes it on.

`functools.wraps` is housekeeping: it keeps the original function’s name and docstring on the wrapper. Without it, `slow_sum.__name__` would return `wrapper` and tracebacks would stop making sense.

The decorators you will meet most

Decorator	What it does
`@router.message(...)` (aiogram)	“call this function on such a message”
`@app.get("/path")` (FastAPI)	“call it on such an HTTP request”
`@dataclass`	writes `__init__` and the printout for you
`@property`	a method read like a field, without parentheses
`@staticmethod`	a method that does not need `self`
`@functools.cache`	remembers the result; same argument, no recomputation

The first two matter most in practice. They **register** the function: you never call it anywhere, yet it gets called when the right event arrives.

```python
@router.message(Command("weather"))
async def cmd_weather(message: Message) -> None:
    ...
```

Searching the project for a call to `cmd_weather` finds nothing. That is normal: the framework calls it, and the `@` line is the application for that.

> **REMEMBER**
> If a function is never called anywhere yet it works, look at the line above it. Almost always there is a decorator there that registered it.

Other languages

The same thing exists everywhere, under other names:

```javascript
// JavaScript: the wrapper is explicit, no @ syntax
const timedSum = withTiming(slowSum);
```

```dart
// Dart: annotations @override, @immutable — more often hints
// than wrappers
```

In React, hooks play a similar role: you never call `useEffect` directly, React calls it itself at the right moment.

WHAT TO TELL THE AI

> **PROMPT**
> What does the @timed decorator do to slow_sum?
> Explain the order in which the lines execute on a call.

When it isn’t clear who calls a function:

> **PROMPT**

```
This function is never called in the code. Who calls it
and when?
```

WHAT TO LOOK FOR

The order of decorators. If there are several, they apply bottom-up — the one closest to the function first. Swapping them changes the behaviour.

A decorator silently swallowing errors. A wrapper with a `try/except` inside means errors from the real function may never reach you (article 2.8).

A wrapper that changes the arguments or the result. Read what exactly is passed into `fn(...)` and what comes back: a wrapper can substitute both.

No `functools.wraps`. A small thing, but because of it tracebacks show `wrapper` instead of function names, and debugging gets harder.

CHEAT SHEET

- `@name` above a function — it is wrapped; look at the wrapper.
- Because of decorators, code does not execute top to bottom.
- `*args, **kwargs` — “I take anything and pass it on”.
- `@router.message`, `@app.get` register a function: the framework calls it.
- A function never called yet working — look for the decorator above it.
- Several decorators apply bottom-up.

2.10 Asynchrony

async, await, and why a bot needs them

WHAT IT IS

Asynchrony is a way of not standing idle while the program waits for something. An API request takes half a second, and ordinary code just stands there for all of it. Asynchronous code gets other things done meanwhile.

For a bot this is essential: while one user waits for the weather, the rest must still get answers. To read the code it is enough to understand two words: `async` and `await`.

WHAT IT LOOKS LIKE IN CODE

```
import asyncio
import time


async def fetch_city(name: str) -> str:
    await asyncio.sleep(0.3)          # as if it were a network request
    return f"{name}: +4"


async def main() -> None:
    t = time.perf_counter()
    for city in ["Oslo", "Bergen", "Tromsø"]:
        print(await fetch_city(city))
    print(f"one at a time: {time.perf_counter() - t:.1f} s")

    t = time.perf_counter()
    results = await asyncio.gather(
        *(fetch_city(c) for c in ["Oslo", "Bergen", "Tromsø"])
    )
    print(f"all at once: {time.perf_counter() - t:.1f} s")


asyncio.run(main())
```

Output:

```
one at a time: 0.9 s
all at once: 0.3 s
```

Three requests of 0.3 seconds each: in sequence, 0.9; together, 0.3. That is the whole benefit of asynchrony, measured.

Three words

`async def` — the function is asynchronous. The main consequence: calling it **does not run it** it returns a promise object. To get the result you need `await`.

`await` — “wait for the result, and while we wait, go do other things”. It can only appear inside an `async def`.

`asyncio.run(main())` — the launch. The point where the asynchronous world begins.

Where it pays and where it doesn’t

Asynchrony helps where the program **waits**: network, disk, database. Where the program **computes**, there is nothing to gain — the processor is busy and there is nothing to switch to.

Helps	Doesn’t help

API requests	multiplying matrices
database work	image processing
reading and writing files	parsing a large text
waiting for messages from Telegram	any heavy computation

The most common mistake: a forgotten `await`

```
data = fetch_city("Oslo")        # forgot await
print(data)
```

This prints something like `<coroutine object fetch_city at 0x...>` plus a `coroutine was never awaited` warning. The function never ran at all.

> **CAREFUL**
> `<coroutine object ...>` in the output or the logs is always a forgotten `await`. It is the first thing to check when asynchronous code "does nothing".

A blocking call in asynchronous code

The second common trouble is an ordinary synchronous library inside an `async def`:

```
async def fetch_city(name: str) -> str:
    r = requests.get(url)        # blocks everything!
    return r.text
```

`requests` is synchronous: for the duration of the request the whole program stops, and the bot stops answering anyone. The asynchronous replacements are `aiohttp`, or `httpx` in async mode. Same with databases: `sqlite3` blocks, `aiosqlite` does not.

The sign while reading: inside an `async def`, a call without `await` to something that goes to the network, the disk or the database.

WHAT TO TELL THE AI

> **PROMPT**
> Are there any blocking calls in this asynchronous code?
> Name the lines and what to replace them with.

> **PROMPT**
> Do these three requests run one after another or together?
> If one after another, show me how to do them together.

> **PROMPT**
> Rewrite this so it doesn't block the event loop.
> You may change libraries, but say which ones you are adding.

WHAT TO LOOK FOR

A forgotten `await`. The most common one. Found by `coroutine` in the output.

A synchronous library inside an `async def`. `requests`, `time.sleep`, a plain `open` on a big file — all of them stop the program entirely. `time.sleep(1)` has to become `await asyncio.sleep(1)`.

Sequential `await`s where one batch would do. A loop with an `await` inside runs the requests one after another. If they are independent, combine them with `asyncio.gather` — as in the example, three times faster.

Asynchrony where it isn't needed. If the program never waits for anything, `async` only adds complexity. The AI sometimes makes everything asynchronous.

CHEAT SHEET

- `async def` does not run when called — it needs `await`.
- `await` only works inside an `async def`.
- It pays where the program waits: network, disk, database. Not for computation.
- `<coroutine object ...>` — a forgotten `await`.
- `requests` and `time.sleep` inside an `async def` block everything.
- Combine independent requests with `asyncio.gather`.

2.11 Dependencies

what is imported, from where, and in which version

WHAT IT IS

Dependencies are other people's code that your project uses. Reading them matters for two reasons: the imports tell you what a file is in the business of doing, and the versions explain why AI-written code might not run.

The second reason is exactly reason 3 from article 3.1, "it remembers something else". Half of all unexplained errors in generated code come down to a version.

WHAT IT LOOKS LIKE IN CODE

```
import os                           # standard library
import sqlite3

import aiohttp                      # external, from requirements
from aiogram import Bot, Router
from aiogram.filters import Command

from config import settings         # ours, from this project
from storage import add_reminder
```

Three groups separated by blank lines: standard library, external packages, own code. This is the conventional order, and it shows at a glance what comes from where.

How to tell them apart

Sign	Where it comes from
`os`, `sys`, `json`, `re`, `datetime`, `pathlib`	standard library, nothing to install
listed in `requirements.txt`	an external package
matches a file or folder name in the project	your own code

If you are not sure, look in `requirements.txt`. What isn't there and doesn't look like a project file is most likely from the standard library.

Two forms of import

```
import sqlite3              # take the whole module
conn = sqlite3.connect(...)

from aiogram import Bot     # take one name out of the module
bot = Bot(token=...)
```

The only difference is how you refer to things afterwards. The form `from ... import *` is bad: it is unclear which names appeared in the file, and they can shadow existing ones.

Where the versions are written

```
aiogram==3.13.1
aiohttp>=3.9,<4.0
python-dotenv
```

- `==3.13.1` — exactly this version.
- `>=3.9,<4.0` — anything in that range.
- nothing specified — anything, including a release that comes out tomorrow and is incompatible.

The last case is the source of “it worked yesterday” errors. Which is why working projects pin their versions.

To find out what is actually installed:

```
pip freeze | grep aiogram
```

> **REMEMBER**
> Before asking for code that uses a library, look up its version and name it in the prompt. It is one command and one of the highest-return habits in this whole book.

Why the version decides so much

Across major versions, libraries break compatibility. The examples you will meet most often:

Library	What changed
aiogram 2 → 3	different handler registration, different imports
pydantic 1 → 2	different method names, different validation
React (classes → hooks)	an entirely different way of writing components
Flutter	changes its API across major versions

There is more code for the old versions on the internet, so the AI reaches for it by default.

WHAT TO TELL THE AI

PROMPT

```
Here is my requirements.txt. Write code only for these versions.

aiogram==3.13.1
aiohttp==3.9.5
```

When the code won’t run:

PROMPT

```
The code fails: ImportError: cannot import name 'Dispatcher'
from 'aiogram'. I have aiogram 3.13.1. Which version was this
written for, and how do I rewrite it for mine?
```

Before adding a new library:

PROMPT

```
Can this be solved with the standard library, without a new
dependency? If not, explain why.
```

That last question is worth asking every time. Every dependency is something to keep updated, something that can break, and something that makes the project bigger.

WHAT TO LOOK FOR

An import that isn't in `requirements.txt`. The code will not run on your machine even though it "works" for the AI. Checked instantly: try running it.

An invented library. The AI sometimes imports packages that do not exist, or methods a package does not have. The check is `pip index versions <name>` or simply a search in the documentation (article 4.8).

A new dependency for one function. A whole package to format a date is almost always too much.

An import inside a function. Sometimes justified (a slow library, circular imports), but more often it is a scar from edits. If you spot one, ask why.

Versions that have drifted apart. `requirements.txt` says one thing, `pip freeze` shows another. The environment was not built from that file, and your machine and the server may differ.

CHEAT SHEET

- Three groups of imports: standard · external · your own.
- Not sure where something comes from — check `requirements.txt`.
- `==` pins the version; nothing specified means anything, and that is risky.
- Always name the library version in the prompt.
- Half of the strange errors in AI code are a version problem.
- Before a new dependency, ask whether you can do without it.

2.12 Reading a traceback

finding the guilty line in thirty seconds

WHAT IT IS

A traceback is what Python prints when a program crashes. It looks intimidating, reads in half a minute, and almost always names the guilty line outright.

The skill pays twice: you understand immediately what happened, and you can hand the AI not "it doesn't work" but the exact cause. The

second matters more — it decides whether it gets fixed on the first try or the fifth.

WHAT IT LOOKS LIKE IN CODE

Here is a program that requests the weather and shows the temperature:

```
import json


def fetch_weather(city: str) -> dict:
    # The server's reply when the city is not found
    raw = '{"cod": "404", "message": "city not found"}'
    return json.loads(raw)


def format_report(data: dict) -> str:
    temp = data["main"]["temp"]
    return f"It is {temp} degrees right now"


def handle_command(city: str) -> str:
    data = fetch_weather(city)
    return format_report(data)


if __name__ == "__main__":
    print(handle_command("Tonsberg"))
```

Run it, and you get:

```
Traceback (most recent call last):
  File "/tmp/tbdemo2/weather2.py", line 21, in <module>
    print(handle_command("Tonsberg"))
          ~~~~~~~~~~~~~~~^^^^^^^^^^^^
  File "/tmp/tbdemo2/weather2.py", line 17, in handle_command
    return format_report(data)
  File "/tmp/tbdemo2/weather2.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'
```

How to read it

Rule one: read bottom to top. The last line is the point; everything else is how we got here.

```
KeyError: 'main'
```

`KeyError` means: something reached for a dictionary key that isn't there. The key is named outright — `'main'`.

Rule two: the second-to-last block is the crash site.

```
  File "/tmp/tbdemo2/weather2.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
```

File, line 11, function `format_report`. The marks under the line are Python's own hint (since 3.11): the `^^^^` points at exactly the part of the expression that failed. Here you can see the problem is `["main"]`, not `["temp"]`. That matters: without the marks you would be guessing which of the two lookups was to blame.

Rule three: the blocks top to bottom are the call path. The program started at line 21, went into `handle_command` (line 17), from there into `format_report` (line 11), and crashed. That path answers the question "where did the data come from" — and that is almost always the real cause.

Rule four: look for your own files. In a real traceback half the lines come from libraries: `site-packages`, `aiogram`, `asyncio`. You can scroll past them. Look at the last line that mentions *your* file: the error is almost always there, even when the crash happened inside a library.

What we learned

Line 11 is not at fault — it honestly does what it was asked. Line 6 is at fault: the server replied "city not found" and the code decided it had received the weather. **The crash site and the cause are different places.** The traceback shows the first; the call path helps you find the second.

WHAT TO TELL THE AI

The most common mistake is retelling the error in your own words.

DON'T DO THIS

```
doesn't work, throws some key error
```

The AI has to guess: which key, in which file, on which line. It will start rewriting everything in sight.

The right way is to hand over the whole traceback, adding what it does not contain: what you did and what you expected.

BETTER

```
Crashes when requesting the weather for a city that doesn't
exist. Expected the reply "City not found", got an exception.

Traceback (most recent call last):
```

```
  File "weather.py", line 21, in <module>
    print(handle_command("Tonsberg"))
  File "weather.py", line 17, in handle_command
    return format_report(data)
  File "weather.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'

Fix it so that the server error is handled in fetch_weather,
rather than surfacing in format_report.
```

That last sentence matters. Without it the AI will most likely wrap line 11 in a `try/except` or substitute `data.get("main", {})`. It stops crashing, the user gets "It is None degrees right now", and the real error goes into hiding.

> **CAREFUL**
> "Make the traceback go away" and "fix the error" are different tasks. By default the AI does the first, because that is literally what it was asked. Say where the error should be handled.

WHAT TO LOOK FOR

The AI muffles the error instead of handling it. The signs in generated code: `except Exception: pass`, `except: return None`, `.get(key, {})` instead of a check. The code stops crashing and starts lying.

The AI fixes the crash site, not the cause. If it only changed the line the traceback pointed at, ask: "where did the wrong data reach this function from?"

A truncated traceback. Copy only the last line and the AI never sees the call path, so it fixes blind. Copy the whole thing, starting at the word `Traceback`.

An asynchronous traceback is longer and messier: it holds `asyncio` frames and library internals. Same rule — find the last line with your own file in it.

CHEAT SHEET

- Read bottom to top: the last line says what happened.
- The second-to-last block is the file and line of the crash; `^^^^` marks the guilty part of the expression.
- Scroll past other people's files, find the last one of yours.
- Crash site ≠ cause. Look for the cause further up the call path.
- Give the AI the whole traceback plus what you did and what you expected.

2.13 Reading a diff

what exactly the AI changed

WHAT IT IS

A diff is a list of changes: what was removed, what was added. When the AI edits code, the diff is what you read — not the whole file. It shows exactly what changed and nothing else.

This is the main tool of acceptance. All of your control over a project rests on the habit of looking at the diff before taking a change.

WHAT IT LOOKS LIKE IN CODE

We asked for the `KeyError` on a missing city to be fixed. Here is what came back:

```
--- a/weather.py
+++ b/weather.py
@@ -2,6 +2,8 @@ import logging

 log = logging.getLogger(__name__)

+DEFAULT_TEMP = 0
+

 def fetch_weather(city: str) -> dict:
     data = call_api(city)
@@ -9,5 +11,5 @@ def fetch_weather(city: str) -> dict:


 def format_report(data: dict) -> str:
-    temp = data["main"]["temp"]
+    temp = data.get("main", {}).get("temp", DEFAULT_TEMP)
     return f"It is {temp} degrees right now"
```

How to read it

A `-` at the start of a line means the line was removed. `+` means added. A line with no sign is context: it did not change and is shown so you can see where you are.

`@@ -9,5 +11,5 @@` is a position marker: there were five lines starting at nine, now five starting at eleven. The exact numbers are rarely needed, but they show whether the file has shifted.

`--- a/` **and** `+++ b/` say which file. When several files change, there are several such blocks.

A modified line shows up as a removal plus an addition. A `-`/`+` pair next to each other is one line that was rewritten.

What we see in this diff

One line changed. `data["main"]["temp"]` became `data.get("main", {}).get("temp", DEFAULT_TEMP)`.

It will stop crashing. But now, when the reply is "city not found", the user gets **"It is 0 degrees right now"**. The error is not fixed — it is hidden, and hidden in a way that looks like a real answer.

This is exactly the case from article 3.5: "make the error go away" and "fix the cause" are different tasks. And the only way to see it is by reading the diff: a run will show that the bot no longer crashes and leave you feeling everything is fine.

> **REMEMBER**
> A diff is read to find not what broke, but what changed quietly. A run checks one case; the diff shows all of them.

How to look at a diff

```
git diff                  # changed but not yet staged
git diff --staged         # already staged
git diff --stat           # just the files and the line counts
git diff -- weather.py    # one file only
```

Start with `--stat`. One line of output where you expected one — good. Seven files where you expected one — a signal to stop (article 3.13).

```
 weather.py | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

The order of reading

1. `--stat` — how many files and lines. Does it match what you expected?
2. **Removed lines** (`-`). What disappeared? Removal is more dangerous than addition: added code is at least visible, while a deleted check is hard to notice.
3. **Added lines** (`+`). Do they do what was asked?

4. **What did not change although it should have.**

The second point is the one people skip most, and it matters more than the third.

WHAT TO TELL THE AI

PROMPT
```
Show me a diff of your changes, not the whole file.
```

PROMPT
```
In this diff: does the behaviour change for cases that used
to work? If so, which ones?
```

PROMPT
```
You removed the line with the length check. Why?
```

The second prompt is a decent self-check after a large edit, but remember: about its own code the AI is lenient (article 3.8). Serious diffs you read yourself.

WHAT TO LOOK FOR

A diff bigger than expected. A two-line change with a fifty-line diff means something else was done too.

Reformatting mixed in with the change. Reordered imports, changed quotes, rewrapped lines — the real change drowns in the noise. This has to be forbidden up front (article 3.13).

Deleted checks. `if not data: return`, `if len(text) > MAX` — lines like these are the first to be lost in a rewrite and look optional.

A quiet change of behaviour. As in the example: there was an exception, now there is a zero. Formally one line, in substance a different program.

Changes to files nobody asked about. Look at `--stat` first.

CHEAT SHEET

- `-` removed, `+` added, no sign means context.
- A `-`/`+` pair next to each other is a rewritten line.
- Start with `--stat`: how many files and lines.
- Read the removed lines before the added ones.
- Look not for a break, but for a quiet change of behaviour.
- A diff bigger than expected — stop and work out why.

2.14 Red flags in generated code

ten signs worth a closer look

WHAT IT IS

The closing article of the chapter: a list of what turns up most often in AI-written code and most often turns out to be a problem. A red flag is not a verdict, it is a reason to stop and read more carefully.

All ten rest on the earlier articles, so they read quickly — this is a summary, not new material.

WHAT TO LOOK FOR

1. Swallowed errors

```
try:
    data = fetch_weather(city)
except Exception:
    pass
```

The program doesn't crash and therefore looks like it works. In reality wrong data travels on, and the cause surfaces nowhere. The same family: `except: return None` and `.get(key, {})` where the field is required (articles 2.8, 2.13).

2. Defaults instead of errors

```
temp = data.get("main", {}).get("temp", 0)
```

Zero degrees looks like a real answer. If a field is required, its absence is an error, not an excuse to substitute a number.

3. Invented libraries and methods

An import of a package that doesn't exist, or a call to a method the library doesn't have. The code looks confident and fails on the first run. Cured by checking the version (article 2.11) and a quick look at the documentation.

4. A blocking call in asynchronous code

```
async def handler(message):
    r = requests.get(url)       # stops everything
```

The bot stops answering everyone for the duration of one request (article 2.10).

5. Secrets in the code

```
TOKEN = "7123456789:AAE-abcdef..."
```

It goes into git and stays in the history forever. Tokens come from environment variables, nowhere else.

6. Code outside functions

The line `conn = sqlite3.connect(DB_PATH)` at the top level of a file runs on import — not when you expect, and not only once (article 2.2).

7. A mutable default value

```
def add_item(item, cart=[]):
```

The list is created once and stays shared across every call (article 2.5).

8. The function does more than was asked

Logging, caching, retries and validation have appeared inside, none of which was discussed. Not necessarily harmful, but it is code you did not order and did not check.

9. Quietly modifying what was passed in

```
def add_tax(items: list[dict], rate: float) -> None:
    for item in items:
        item["price"] = ...
```

Outside the function the data changed too. A second call charges tax on the tax (article 2.5).

10. Comments instead of code

```
# Access rights need to be checked here
# TODO: handle the empty list case
return result
```

The comment describes something the code does not do. It looks like a considered decision and is in fact a hole, politely labelled.

WHAT TO TELL THE AI

One prompt to check with, after the code arrives and before you accept it:

PROMPT

```
Check your own code against this list:
- swallowed exceptions and defaults instead of errors
- calls to methods that may not exist in this library version
- blocking calls inside async
- secrets in the code
- code outside functions
- mutable default values in arguments
- modifying data passed in as an argument

Which of these are present? If none, say so.
```

A prompt like this works better than a general “review this”: it names what to look for (article 3.8).

REMEMBER
A red flag is a reason to read more closely, not a reason to rewrite. Every one of the ten is sometimes justified; the question is whether it is there on purpose.

CHEAT SHEET

- Swallowed errors: `except Exception: pass`, `.get()` instead of a check.
- A default instead of an error is an answer that lies.
- Invented libraries and methods — check the version and the docs.
- `requests` and `time.sleep` inside an `async def`.
- Tokens in the code go into git history forever.
- Code outside functions runs on import.
- `=[]` in arguments, modifying what was passed, doing extra work.
- `# TODO` in the code is a hole, not a solution.

3 Stating the task

This is where most of the book's practical value sits. An AI almost never goes wrong "on its own" — it does exactly what it heard. Which means the only way to steer the result is through the prompt.

3.1 Why the AI does the wrong thing

four reasons, and all four are cured by the prompt

WHAT IT IS

An AI almost never goes wrong "on its own". It does exactly what it heard — it just heard something other than what you meant. There are exactly four reasons, and each has a sign that gives it away.

The article exists so that you don't fix blind. When you get the wrong result, identify the reason first and only then rewrite the prompt: different reasons need different cures, and curing the wrong one adds a second mistake to the first.

WHAT IT LOOKS LIKE IN CODE

One task throughout: add a command to the bot that shows the weather. Below are four ways to get the wrong thing.

Reason 1. You didn't say

The most common. You left out a detail and the AI supplied its own.

DON'T DO THIS

```
Add a weather command
```

In your head: a bot on aiogram, the key in an environment variable, a one-line reply. The AI heard none of that and substituted the industry average: synchronous `requests`, the key as a string right there in the code, a five-line reply with emoji.

The sign: the code works, but the solution is not yours. No complaints about the quality; the complaint is about the choices.

The cure: say what was in your head. The four parts of a prompt — article 3.2.

Reason 2. It cannot see

The AI does not know your project unless you show it. In a chat it sees no files at all. An agent in the terminal does see them, but only the ones it has managed to read.

DON'T DO THIS

```
Do it the same way as in the neighbouring handler
```

The sign: the AI invents function names, imports modules that don't exist, writes in a style the project doesn't use.

The cure: show it. Paste the file, name the path, tell it to read. If you find yourself explaining the same thing every time, start a project context file (article 3.14).

Reason 3. It remembers something else

The model was trained on whatever code was most plentiful on the internet. And the most plentiful is code for old versions: `aiogram 2` rather than `aiogram 3`, React class components rather than hooks, the old `Navigator` in Flutter.

DON'T DO THIS

```
Write a handler on aiogram
```

The sign: the code looks confident and does not run. The errors are `ImportError`, `AttributeError`, "no such parameter". Or the library itself complains that the approach is deprecated.

The cure: name the version. Not `aiogram` but `aiogram 3`. Not `React` but `React 18, function components, hooks`.

> **CAREFUL**
> This is the one reason of the four that cannot be cured by adding detail. You can describe the task perfectly and still get code for an old version — if you don't name the version.

Reason 4. It heard you literally

The AI did exactly what was asked. The trouble is that what was asked was the wrong thing.

DON'T DO THIS

```
Get rid of this error
```

The error will disappear. Most likely it will be wrapped in a `try/except` and suppressed — formally, the request was fulfilled. The difference between "get rid of the error" and "fix the cause" is obvious to you and not to the AI.

The sign: the task is closed, the symptom is gone, the problem remains. Or it left together with behaviour you needed.

The cure: state the goal, not the action. Not "get rid of the error" but "make the server error be handled in `fetch_weather`".

WHAT TO TELL THE AI

A summary table — the reason can be identified from it in seconds.

What you see	Reason	What to do
Working code, but not your solution	you didn't say	name the specifics
Invented names, foreign style	it cannot see	show the files
Won't run, `ImportError`	remembers something else	name the version
Symptom gone, problem stayed	heard you literally	name the goal, not the action

When the reason isn't clear, ask the AI itself — it answers this honestly:

PROMPT

```
Before you change anything: which assumptions did you rely on
when writing this code? Which of them did I never tell you?
```

WHAT TO LOOK FOR

Don't fix reason 1 with the cure for reason 3. The typical scene: you got the wrong result, you start adding more and more detail to the prompt, and the code still doesn't run — because the problem was the version, not the detail.

One task, one reason at a time. When you rewrite the prompt, change one thing. Otherwise you won't know what helped, and next time you'll be guessing again.

"It's stupid" is not a reason. That feeling comes from reason 4, when the AI did literally what was asked. Check what you asked for, not what you meant.

CHEAT SHEET

- Four reasons: you didn't say · it can't see · it remembers something else · it heard you literally.
- Identify the reason first, rewrite the prompt second.
- Always name the version: no amount of detail cures that one.
- State the goal, not the action.
- Not sure why — ask which assumptions it relied on.

3.2 Anatomy of a prompt

context, goal, constraints, definition of done

WHAT IT IS

A good prompt has four parts: **context**, **goal**, **constraints** and a **definition of done**. This is not a form to fill in every time — it is a list of what the AI has to learn from somewhere. If you don't say it, it will guess, and the guess will be the industry average rather than yours.

This article is the foundation of all of chapter 3. What follows are ready-made templates for particular tasks, and every one of them is built from these four parts.

The four parts

Part	Answers	What happens if you skip it
Context	where, and in which project	the AI writes code "from scratch", not fitted to yours
Goal	what the result should be	you get something similar, but not it
Constraints	what must not be done	it rewrites extra things, adds dependencies
Definition of done	how to know it is finished	"finished" arrives earlier than it should

WHAT IT LOOKS LIKE IN CODE

Take a live task: a Telegram bot needs a limit on how often commands can be sent.

DON'T DO THIS

```
Add spam protection to the bot
```

The prompt looks clear, and it contains none of the four parts. What the AI will decide for you: which framework, what limit, where to keep the counters, what to reply to the user, whether to count commands or

messages, whether to touch the existing handlers. Six decisions, none of them yours.

Now the same thing with the four parts:

```
BETTER
Context: bot on aiogram 3, file handlers/commands.py.
Handlers are functions with the @router.message decorator.
Data is stored in SQLite via aiosqlite, module db.py.

Goal: rate-limit commands — no more than 5 commands per minute
from one user.

Constraints:
- no new dependencies, only what is already in requirements.txt
- don't change the existing handlers, do it as separate middleware
- keep the state in memory, don't write to the database

Done when: the sixth command within a minute gets the reply
"Too fast, wait a moment" and a warning goes to the log.
All other commands work as before.
```

The second prompt is ten times longer and saves an hour of rework.

How to know there is enough context

Context is what the AI cannot see. If it works inside the project (an agent in the terminal), it reads the files itself and context shrinks to "where to look". In a chat with no access to the code, context has to come from your words or from a pasted file.

The minimum that is needed almost always:

- the language, the framework and the **version** — `aiogram 3`, not just `aiogram`
- the file or module to work in;
- how similar problems are already solved in the project.

The third point is underrated. It matters more than the first two: it is what makes the AI write in the project's style rather than its own.

> **TIP**
> If you find yourself explaining the context from scratch every time, its place is a project context file. That is article 3.14.

The definition of done

The most-skipped part and the most useful. A definition of done is a checkable condition, not a wish.

Not a definition	A definition

“so that it works well”	“the sixth command within a minute is refused”
“reliable”	“on a dropped connection it retries three times, then logs”
“fast”	“a reply in under a second with 100 users”
“nice-looking”	“buttons in two rows, labels no longer than 20 characters”

It does two things. First, the AI knows when to stop. Second, you get a way to check the result without reading the code — you just check the condition.

WHAT TO TELL THE AI

The four parts do not have to be a list. In short tasks they fit in a paragraph:

PROMPT

```
In utils/text.py add a function that splits a long message
into chunks of 4096 characters at paragraph boundaries, without
breaking words. No external libraries. Done when a 10,000-character
text yields 3 chunks and joining the chunks equals the original.
```

All four are there: context (`utils/text.py`), goal (a splitting function), constraint (no libraries), definition of done (10,000 characters → 3 chunks, rejoining matches).

For a large task the same four parts unfold into a mini-spec — article 3.15.

> **REMEMBER**
> A missing part does not go away — the AI fills it in. The only question is whether you will be happy with its version.

WHAT TO LOOK FOR

Too much context is harmful too. Dump the whole project and the important part drowns. Give what relates to the task and point at the rest in one line.

Phrase constraints as “don’t”, not “only”. “Use only the standard library” is sometimes read as a preference. “Don’t add new dependencies” works more reliably.

A definition you cannot check is not a definition. If you don’t know how you will check the result, the task is not fully stated. That is a

signal to go back to step 1 of the working cycle, not to keep writing the prompt.

The framework version decides more than it seems. `aiogram 2` and `aiogram 3` are incompatible, React changed its approach after hooks, Flutter changes its API across major versions. Don't name the version and you get code for whichever one was most plentiful in training.

CHEAT SHEET

- Four parts: context, goal, constraints, definition of done.
- Context is what the AI can't see. Always: framework, version, file.
- Write constraints as "don't do", not "only do".
- A definition of done must be checkable: a condition, not an adjective.
- If you don't know how you'll check the result, the task isn't stated yet.
- A missing part gets filled in by the AI.

3.3 Levels of detail

when one line is enough and when you need a mini-spec

WHAT IT IS

A detailed prompt is not always better than a short one. On a small change a mini-spec costs ten minutes instead of one; on a large task one short line turns into an hour of rework.

The level is chosen not by the size of the task but by the **cost of being wrong**: what it costs to get the wrong thing. This article gives a rule that picks the level in a couple of seconds.

WHAT IT LOOKS LIKE IN CODE

Three levels, short to detailed.

Level 1. One line

For changes you can see with your eyes and roll back easily.

PROMPT

```
Rename get_data to fetch_weather throughout the file.
```

PROMPT

```
Add logging at the start of every handler.
```

Checking takes seconds: look at the diff, accept or roll back. There is nothing to spell out.

Level 2. A paragraph

The workhorse. An ordinary feature that has to fit into existing code.

PROMPT

```
In handlers/weather.py add a 10-minute cache of weather replies
so we don't hit the API on every request. Keep it in memory,
a dict keyed by city name. Don't add new dependencies.
```

All four parts of a prompt are already here, just compressed into three lines. This is enough in most cases.

Level 3. A mini-spec

For tasks that touch several files, change the shape of the data, or are hard to roll back.

PROMPT

```
Task: move subscription storage from a JSON file to SQLite.

Context: subscriptions currently live in data/subs.json and are read
by load_subs() in storage.py. There are five calls to load_subs and
save_subs in the project, all in handlers/.

What is needed:
- a subscriptions table: user_id, city, created_at
- the same load_subs/save_subs functions, signatures unchanged,
  so the handlers don't have to be touched
- migration: on first start, move the data out of the JSON and
  rename the file to subs.json.bak

Constraints:
- plain sqlite3 only, no ORM
- don't change the handlers at all

Done when: the bot starts on an empty database, subscriptions from
the old JSON are visible after a restart, and all five calls work
as before.
```

Long — and it pays for itself. A task like this, started from one line, gets redone three times.

WHAT TO TELL THE AI

The rule for choosing: **look not at the size of the task but at what happens if it comes out wrong.**

Cost of being wrong	Level	What the task looks like
Seconds, visible at a glance	one line	rename, formatting, a single call
Minutes, easy to roll back	paragraph	a new function, a change in one file
Half an hour or more	mini-spec	several files, data structure, migrations

Two questions that settle any doubt:

1. **Can I check the result without reading the code?** If not, you need a higher level — and first of all a definition of done.
2. **Is it easy to roll back?** One commit rolls back easily. A database migration does not.

> **TIP**
> Torn between a paragraph and a mini-spec — ask for a plan. Let the AI lay out what it intends to do first (article 3.10). A ten-line plan costs a minute and shows how closely you and the AI understood the task the same way.

WHAT TO LOOK FOR

A mini-spec on a small change does harm. The more text there is, the more the AI "helps": rewriting neighbouring functions, adding handling for cases nobody asked about. A short task gets a short prompt.

Length is not the same as detail. Three lines with the framework version, the file name and a definition of done beat twenty lines of musing about how nice it would be.

The level goes up as you work. You started with a line and got the wrong thing — don't append to that same turn; restate it one level higher. Rewriting is cheaper than patching.

CHEAT SHEET

- The level is set by the cost of being wrong, not the size of the task.
- One line — visible at a glance and easy to roll back.
- A paragraph — an ordinary feature, the normal mode.
- A mini-spec — several files, data structures, migrations.
- Can't check the result without reading the code — go one level up.
- Got the wrong thing — restate it, don't append.

3.4 Template — a new function

the most frequent request in daily work

WHAT IT IS

New functionality is what people ask an AI for most often. This request has a stable shape: name the place, describe the behaviour, draw the boundaries, set the check.

The template below is not a form to fill in but a reminder. In time you say it without thinking, but early on it is worth looking at.

WHAT IT LOOKS LIKE IN CODE

The template

PROMPT

```
Where: <file or module; how the neighbouring code is built>
What: <behaviour — what goes in, what comes out>
Boundaries: <what not to do; what not to touch>
Done when: <a checkable condition>
```

Bad

DON'T DO THIS

```
I need a function for working with user subscriptions
```

"Working with subscriptions" means adding, deleting, checking and listing. The AI will choose for you, and will most likely write a two-hundred-line class with all four operations, half of which you don't need.

Good

BETTER

```
Where: storage.py, next to load_subs(). Data is in SQLite,
connection via get_conn() from db.py.

What: a function add_sub(user_id: int, city: str) -> bool.
Adds a subscription. Returns False if that subscription already
exists, True if it was added.

Boundaries: only this function, don't touch the others.
Don't add new dependencies.

Done when: a repeat call with the same arguments returns False
and no second row appears in the table.
```

The result is predictable: one function, a clear signature, checkable behaviour.

WHAT TO TELL THE AI

What is required

The signature. The name, the arguments with their types, what it returns. This is the cheapest detail and the most useful: it removes half of all ambiguity. If you don't know what to call it, describe it in words and the AI will suggest something.

The behaviour in the unhappy case. What to do when the data isn't there, the network dropped, the argument is empty. Don't say and the AI chooses for you — and most often it chooses to return `None` quietly.

The boundaries. The sentence "don't touch the other functions" saves more time than anything else. Without it the AI happily "improves" the neighbouring code too, and the diff grows from ten lines to two hundred.

What is optional

There is no need to spell out the algorithm when it is obvious — that is the AI's job. There is no need to ask for a comment on every line: they clutter the code and are the first thing to go stale.

> **TIP**
> If the function will be used in several places, add: "show an example call". One line of example immediately shows whether it is pleasant to use — and an awkward signature is better reworked now than after ten call sites.

When there are several functions

Don't ask for five functions in one prompt. Ask for the first, read it, accept it — and then get the rest with one sentence: "do the same for `delete_sub` and `list_subs`". The AI picks up the style of the code you have already accepted, and checking is easier.

WHAT TO LOOK FOR

The function does more than was asked. The sign: logging, caching, retries and validation have appeared inside, none of which was discussed. Not harmful in itself, but it is code you did not order and did not check.

Silent error handling. `return None` instead of an exception, `except: pass`, defaults where there should be an error. The code doesn't crash and therefore looks fine — and later you go hunting for where the `None` came from.

A signature convenient for the AI, not for you. Seven arguments, five of them optional, a dictionary instead of explicit parameters. If the call reads badly, rework it right away.

The function doesn't fit the project. Its own database connection instead of `get_conn()`, its own log format, its own naming style. Most

often this is reason 2 from article 3.1: the AI simply never saw how the neighbours are built.

CHEAT SHEET

- Four lines: where · what · boundaries · done when.
- The signature is the cheapest and most useful detail.
- Always say what to do in the unhappy case.
- "Don't touch anything else" saves the most time.
- Several functions — one at a time, then "do the same".
- Check whether the function does more than was asked.

3.5 Template — fix a bug

what you did, what you expected, what you got

WHAT IT IS

A fix request differs from a new-function request in one way: here you are not giving an assignment but **evidence**. The AI did not watch it break, and all it has is your description.

From which the main rule follows: do not retell the error in your own words. Hand over what you saw, verbatim, and add what the traceback does not contain — what you did and what you expected.

WHAT IT LOOKS LIKE IN CODE

The template

PROMPT

```
What I did: <the action that led to the error>
What I expected: <the behaviour you consider correct>
What I got: <output, traceback, screenshot — verbatim>

<the whole traceback>

Where to fix: <if you know which layer should handle it>
```

Bad

DON'T DO THIS

```
the bot crashes on weather, fix it
```

The AI knows neither the city, nor the error, nor the line. It will start rewriting the whole handler, changing things that worked along the way.

Good

BETTER

```
What I did: sent the bot /weather Tonsberg
What I expected: the reply "City not found" — the API has no such city
What I got: an exception, the bot did not reply at all

Traceback (most recent call last):
  File "handlers/weather.py", line 34, in cmd_weather
    await message.answer(format_report(data))
  File "handlers/weather.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'

Where to fix: the API error should be caught in fetch_weather,
not in format_report.
```

WHAT TO TELL THE AI

Three questions without which nothing gets fixed

What you did. The reproduction steps. If it doesn't happen every time, say so too: "fails about one time in five" changes the direction of the search completely.

What you expected. Without it the AI cannot tell a bug from intended behaviour. In the example above, that line is what says "City not found" is a normal reply rather than an error.

What you got. Verbatim. The whole traceback, starting at the word `Traceback`. How to read one — article 2.12.

Where to fix — optional, but valuable

If you don't say, the AI fixes wherever the traceback pointed. And the crash site and the cause are different places. Line 11 honestly does what it was asked; the guilty line is the one that decided a server reply was the weather.

> **REMEMBER**
> "Make the error go away" and "fix the cause" are different tasks. By default the AI does the first, because that is literally what it was asked.

When there is no traceback

Sometimes nothing crashes, it just works incorrectly. Then the data becomes the evidence:

PROMPT

```
What I did: /stats for October
What I expected: 142 records, as in the database
What I got: 138
```

```
Here is the query I count with: <code>
Here is what the database says: <output of SELECT COUNT(*)>
```

Two numbers that don't match are evidence every bit as good as a traceback.

WHAT TO LOOK FOR

The AI muffles the error instead of handling it. The signs in the diff: a `try/except` around the line that failed, `.get(key, {})` instead of a check, a default where there should be an error. It stops crashing and starts lying.

The crash site is fixed, not the cause. If the change only touched the line the traceback pointed at, ask: "where did the wrong data reach this function from?"

Extra things rewritten along the way. A two-line fix should not produce a fifty-line diff. Add to the prompt: "change only what the fix requires".

Fixed but not verified. A diff that looks right does not mean the bug has gone. Reproduce the very case you started with.

CHEAT SHEET

- Three lines: what I did · what I expected · what I got.
- Hand over the traceback verbatim and in full.
- Say which layer to fix in, or the crash site gets fixed instead.
- "Make the error go away" ≠ "fix the cause".
- No traceback — two numbers that don't match are the evidence.
- After the fix, reproduce the original case.

3.6 Template — refactoring

changing the shape without touching the behaviour

WHAT IT IS

Refactoring is changing code without changing what it does. Splitting a long function, lifting out a repeat, renaming things so a human can read them. From the outside, after a refactoring, everything works exactly as before.

Hence the main difficulty of the request: **you have to say explicitly that the behaviour must not change**. Otherwise the AI takes the task as "make it better" and adjusts the logic along the way — which is not refactoring, and now everything has to be re-checked.

WHAT IT LOOKS LIKE IN CODE

The template

```
PROMPT
What's wrong: <why the code needs reworking>
What to do: <which change of shape>
Behaviour stays: <what must work exactly as before>
How I'll check: <what proves nothing broke>
```

Bad

```
DON'T DO THIS
Refactor handlers/weather.py, make it nicer
```

"Nicer" is an invitation to rewrite the file from scratch. You get a different architecture, different names, edge cases fixed (or broken) in passing, and a diff nobody can read.

Good

```
BETTER
What's wrong: the cmd_weather function is 80 lines and mixes
the API request, parsing the reply and building the text.

What to do: split it into three functions — fetch, parse, format.
Leave cmd_weather as the entry point; it only calls them in turn.

Behaviour stays: the reply texts, the log format and the error
handling remain exactly the same, down to the character.

How I'll check: I'll run the same three commands as before and
compare the bot's replies word for word.
```

WHAT TO TELL THE AI

The sentence not to start without

> *The behaviour stays the same.*

Write it outright, even when it seems obvious from the word "refactoring". For an AI that word is vague: in the training data it covers both renaming a variable and rewriting a module from scratch.

Refactoring and a feature are separate passes

The temptation is strong: while we're in this file anyway, let's add a cache. Don't — and here is why. If something breaks afterwards, you won't know whether the refactoring broke it or the new feature did. Kept apart they are diagnosed instantly; mixed together, in hours.

PROMPT

```
For now only split it into functions, with no change in behaviour.
We'll add the cache as a separate next step.
```

REMEMBER

A refactoring is always its own commit. That is not etiquette, it is how you avoid losing an evening to finding out what exactly broke.

What to check with

Tests are best: run them before, run them after, the result matches. If there are no tests, a manual check will do, but it has to be described in advance — otherwise after the change you find there is nothing to compare against.

A trick that costs a minute: before the refactoring, save the program's output for a few inputs to a file. Afterwards, compare.

```
python bot.py --dry-run > before.txt
# ... refactoring ...
python bot.py --dry-run > after.txt
diff before.txt after.txt
```

An empty `diff` means the refactoring went cleanly.

WHAT TO LOOK FOR

A diff bigger than it should be. Splitting a function into three means moving lines, not rewriting them. If almost every line in the diff is changed, the code was rewritten from scratch and the behaviour has almost certainly drifted.

The texts changed. The most common uninvited improvement: the AI "tidies up" the messages to the user. "City not found" becomes "Unfortunately, the specified city could not be found 😔". A small thing, and already a change of behaviour.

New dependencies or abstractions appeared. A base class, a factory, a decorator — where all you asked for was splitting a function. A sign the AI was solving its own problem rather than yours.

Edge handling disappeared. The empty-list check, `if not data`, an early `return` — in a rewrite these lines are the first to be lost, because they look optional.

CHEAT SHEET

- Four lines: what's wrong · what to do · behaviour stays · how I'll check.
- Write "the behaviour stays" explicitly; the word "refactoring" does not guarantee it.
- Refactoring and a feature are separate passes and separate commits.
- Save the output before and after, compare with `diff`.
- A big diff when splitting a function means the code was rewritten, not moved.

3.7 Template — explain this code

the key request for anyone learning to read

WHAT IT IS

The most underrated request. It creates nothing, so it feels optional — and in fact it is what turns someone else's code into yours and closes step 4 of the working cycle.

The important part is asking **before** you accept the code, not after it breaks. Two minutes of explanation saves an hour of debugging a week later.

WHAT IT LOOKS LIKE IN CODE

Three levels of question

PROMPT

```
What does this do, in two sentences?
```

For quick orientation: you opened someone else's file and need to know whether it matters at all.

PROMPT

```
Explain line by line what this function does.
For each line: what happens and why it is needed.
```

For code you are about to accept into the project.

PROMPT

```
Why is it done this way here? What were the alternatives
and why was this one chosen?
```

For code that looks odd. Often it turns out the oddity is justified — or that there is no justification and the code can be simplified.

The question that works best of all

BETTER

```
What happens if the API returns an empty list?
And if the connection drops halfway through?
```

"What happens if" tests code for soundness better than asking for an explanation. When explaining, the AI retells the intention; when answering "what if", it has to follow the real path of execution — and it often finds the hole itself.

WHAT TO TELL THE AI

Ask for an explanation at your level

An explanation is useless if it contains as many unfamiliar words as the code did.

PROMPT

```
Explain it assuming I know Python at the level of functions and
dictionaries, but have never used async and don't know what a
decorator is.
```

This is not embarrassing and not "for beginners" — it saves time. The AI adjusts the explanation and you avoid five follow-up questions.

Ask for an example rather than a definition

An abstract explanation is forgotten; a concrete one is not.

PROMPT

```
Show me an example: what data arrives in this function and what
it returns. Two or three cases, including a failing one.
```

> **TIP**
> A good way to check that you understood: retell the explanation in your own words and ask to be corrected. "Here is how I read it: the function takes X, and if Y it does Z. Right?" Mistakes in your understanding surface immediately.

Explanation as acceptance

Before accepting generated code, one question is worth asking:

PROMPT

```
Name three places in this code where it will break
on unexpected input.
```

If the AI cannot name a single one, it probably did not check; if it names three, you have a ready-made list to test.

WHAT TO LOOK FOR

The AI explains the intention, not the code. It reads the function names and retells what they *ought* to do. If a function is called `validate_user` but validates nothing, the explanation will still say "validates the user". The defence: ask not "what does it do" but "what happens with this input".

An explanation that sounds confident and is wrong. Especially for third-party libraries and rare language constructs. If a decision depends on the explanation, check the documentation.

Too smooth. If the explanation contains not one "this could break here", it is probably a retelling rather than an analysis. Ask about the weak spots outright.

CHEAT SHEET

- Ask before you accept the code, not after it breaks.
- Three levels: in two sentences · line by line · why this way.
- "What happens if…" works better than "explain".
- Name your own level and you get an explanation without the padding.
- Retell it in your own words and ask to be corrected.
- The AI tends to retell the intention from the names rather than read the code.

3.8 Template — review this

and why you shouldn't ask in the same chat

WHAT IT IS

A review is a request to find problems in code. It has a peculiarity the others don't: **an AI is bad at criticising code it has just written itself.** In the same conversation it tends to confirm its decisions rather than challenge them.

So a review almost always starts by moving the code into a clean session.

WHAT IT LOOKS LIKE IN CODE

Bad

DON'T DO THIS

```
Review this code
```

With no indication of what to look for, you get a list of generalities: "add error handling", "use type annotations", "extract the constants". All true and all useless — the same list could be written without looking at the code.

Good

BETTER

```
Here is a function from my project. I didn't write it;
I need an analysis.
```

```
Look in this order:
1. Cases where it returns a wrong result silently, with no error
2. Unhandled edge cases: empty data, None, a dropped connection
3. Anything that breaks if it is called from two places at once

Don't comment on style or naming.
For each finding: what breaks and on what data.
If there are no findings, say so — don't invent any.

<code>
```

Three differences: the order of importance is set, the noise is cut off, and finding nothing is explicitly allowed.

The sentence that changes the result

> *Here is a function from my project. I didn't write it.*

It takes away the AI's role as the author defending their work. Easy to verify: ask for a review of your own code in the same chat and in a new one — the difference is immediate.

WHAT TO TELL THE AI

Set an order, not just a list

Without priorities a review turns into twenty items where a typo in a comment sits next to data loss. Ask for them sorted by the weight of the consequences:

PROMPT

```
Sort the findings from the most dangerous to the most trivial.
For each one, say what happens if it isn't fixed.
```

Allow it to find nothing

An AI tends to produce findings because that is what it was asked for. If there are none, it will invent some. One sentence removes that:

PROMPT

```
If there are no serious problems, say so plainly.
Don't make up remarks.
```

CAREFUL

Treat the findings as hypotheses, not verdicts. Some of them will be wrong — the AI does not see the whole project and may call something a problem that is handled elsewhere. Check every remark against the code.

Three questions instead of a review

When the whole code does not need going through, short, targeted questions work:

PROMPT
```
Is there a place here where data can be lost without an error?
```

PROMPT
```
Which of this breaks if the function is called twice in a row?
```

PROMPT
```
Which line would you rewrite first, and why?
```

The last one is especially good: it forces a choice rather than a list of everything.

WHAT TO LOOK FOR

A review made of generalities. "Add logging", "think about performance" — remarks that fit any code at all. A sign the prompt never said what to look for.

Invented problems. The AI cites library behaviour that doesn't exist, or an edge case impossible given the project's logic. Check against the code and the docs.

The main thing missed. No findings does not mean the code is good — only that the AI found nothing. It does not cancel your own check.

Reviewing its own code in the same chat. The most common mistake. A new session costs thirty seconds and changes the quality of the analysis more than any wording does.

CHEAT SHEET

- Review in a clean session, with "I didn't write this code".
- Say what to look for and in what order of importance.
- Cut the noise: "don't comment on style".
- Allow it to find nothing, or it will invent something.
- Findings are hypotheses; check each one against the code.
- "Which line would you rewrite first" is a short substitute for a full review.

3.9 Template — write tests

writing them is optional, reading them is not

WHAT IT IS

A test is code that checks other code. You will not have to write tests yourself — the AI writes them readily. But you **will** have to read them, and that pays off: a good set of tests is a list of what your program promises to do.

For anyone working with an AI, tests have a second use that matters more than the first: they stop you breaking what already worked. Ask for a change, run the tests, and you see at once whether the AI disturbed the neighbours.

WHAT IT LOOKS LIKE IN CODE

First the list of cases, then the code

The key move: don't ask for tests straight away. Ask for a list of what to check.

BETTER

```
Here is the split_message function. List the cases worth
covering with tests. No code yet — just the list.
Mark which cases you consider important and which are
just-in-case.
```

The list reads in half a minute, and it shows immediately whether the two of you understood the task the same way:

AI REPLY

```
Important:
1. Text shorter than the limit — returned as one chunk
2. Text longer than the limit — split at paragraph boundaries
3. Joining the chunks equals the original text
4. A word is not broken in the middle

Just in case:
5. An empty string
6. Text exactly at the limit, character for character
7. A single paragraph longer than the limit
```

Case 7 is the one people usually forget. With the list in front of you, you decide what is needed and what is not, and only then ask for the code.

PROMPT

```
Write pytest tests for cases 1–4 and 7. Skip the rest.
```

The template

PROMPT

```
Code: <what we are testing>
With: <pytest / unittest / whatever the project already uses>
Cases: <the list agreed in advance>
Without: <mocks where avoidable; the real network; files on disk>
```

WHAT TO TELL THE AI

A test checks behaviour, not construction

A bad test breaks on any refactoring, even when the program still works correctly. That happens when the test peers inside: checking private fields, the order of calls, intermediate variables.

```
PROMPT
The tests should check only what the function returns and its
visible effects. Don't peer into the internals: after a
refactoring the tests should still pass.
```

Ask for tests that fail first

A way to be sure a test checks anything at all:

```
PROMPT
Before fixing the bug, write a test that reproduces it.
It must fail against the current code.
```

A failing test is proof the bug was understood correctly. If it is green straight away, it is checking the wrong thing.

> **REMEMBER**
> A test that has never failed guarantees nothing. Before trusting a test, make sure it knows how to go red.

Running them is not optional

Generated tests have to be run. It sounds obvious and gets skipped constantly — tests look convincing and create a false sense of coverage.

```
pytest -q
```

WHAT TO LOOK FOR

A test that always passes. `assert result is not None` where the function cannot return `None`. Looks like a check, checks nothing.

Mocks instead of checking. The AI replaces half the system with stubs, and the test verifies that the stub returned what was put into it. Formally green, substantially empty.

Tests that repeat the implementation. If the test reproduces the function's algorithm line by line, it isn't checking correctness — it is checking that the code equals itself.

Only the happy cases covered. Empty data, `None`, a dropped connection, an input that is too long — that is where the bugs hide. If the set contains not one failing case, the set is incomplete.

CHEAT SHEET

- First the list of cases, then the test code.
- A test checks behaviour, not internal construction.
- A test for a bug must fail before the fix.
- A test that has never failed guarantees nothing.
- Generated tests must be run.
- Not one unhappy case in the set — the set is incomplete.

3.10 Technique — plan first, code second

ten lines of text instead of two hundred lines of rework

WHAT IT IS

The technique is simple: before asking for code, ask for a plan. Let the AI write down what it intends to do — and you check whether that matches what you had in mind.

The benefit is in **where** the mismatch shows up. A wrong plan is visible in half a minute; wrong code, an hour later, once it is already woven into the project and growing call sites.

WHAT IT LOOKS LIKE IN CODE

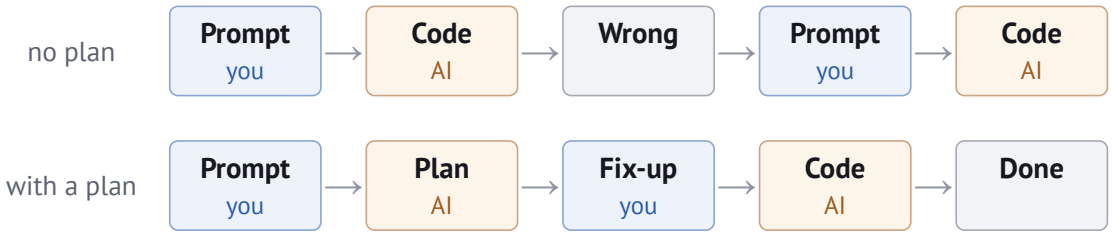


Fig. 3.1 · A mismatch is cheaper to catch in the plan. The path with a plan is one step longer and shorter by all the rework.

How to ask

BETTER

```
Don't write code. First lay out a plan:
- which files you will touch and what changes in each
- which functions you will add, with signatures
- what could break in the existing code

I'll wait for the plan, then tell you what to adjust.
```

The sentence “don’t write code” is needed literally: without it the AI writes both the plan and the code at once, and the point of the tech-

nique is lost — nobody reads a plan with a finished solution sitting underneath it.

What to check in the plan

Not everything, four things:

What we look at	Warning sign
The list of files	more files than you expected
The signatures	arguments you had not anticipated
"What could break"	empty, or "nothing"
The order of steps	it starts by reworking something that works

An empty "what could break" section is the most reliable sign that the AI did not think about consequences but listed its wishes.

WHAT TO TELL THE AI

Adjusting a plan is an ordinary conversation; there is no need to rewrite the whole prompt:

PROMPT

```
The plan is broadly right, but:
1. don't touch storage.py, leave the subscriptions as they are
2. two functions instead of a new class
3. start with the migration, not with the handlers

Rewrite the plan with that in mind.
```

When the plan is fine, one sentence:

PROMPT

```
Plan accepted, work to it. If along the way you find the plan
doesn't work, stop and tell me rather than inventing another
solution on the fly.
```

The second sentence matters. Without it the AI, on hitting an obstacle, will quietly turn aside, and you get code that does not match the agreed plan.

> **TIP**
> Save the accepted plan in a task file — in `docs/tasks/`, say. A week later, coming back to the code, you will read not only what was done but why it was done that way.

When the technique isn't needed

For one-line changes (article 3.3) a plan is wasted time. The technique pays off where more than one file is involved or where rolling back is expensive.

WHAT TO LOOK FOR

A suspiciously smooth plan. Every step simple, no risks, all of it fits in three bullets. Usually that means the difficulty was not found, not that there is none. Ask: "what is the riskiest thing in this plan?"

The code doesn't match the plan. It happens, especially in long sessions. When accepting the work, compare: the same files? the same signatures?

A plan as a way of stalling. If plans are requested three times in a row and reworked each time, the problem is not the plan — the task is not clearly stated. Go back to step 1 of the working cycle.

CHEAT SHEET

- "Don't write code" is a required sentence; otherwise you get both at once.
- In the plan, check: files, signatures, what breaks, the order of steps.
- An empty "what could break" means consequences were not considered.
- "If you get stuck, stop and say so rather than improvising."
- Keep the accepted plan next to the task.
- For one-line changes the technique is unnecessary.

3.11 Technique — let the AI ask the questions

a way to see what you left out

WHAT IT IS

Before the AI starts work, ask it to ask you questions. The technique addresses reason 1 from article 3.1 — "you didn't say" — and addresses it better than any template, because it finds the gaps you did not know were there.

The one-line form:

PROMPT

```
Before writing any code, ask me 3 questions whose answers
would most change the solution.
```

The limit of "3" matters. Without it there will be ten questions, half of them trivial, and you will get bored of answering by the fourth.

WHAT IT LOOKS LIKE IN CODE

The task looks clear:

```
PROMPT
Add a /stats command to the bot — a user's statistics for the month.
Before writing any code, ask me 3 questions whose answers
would most change the solution.
```

The reply:

```
AI REPLY
1. "For the month" — the last 30 days or the current calendar
   month? The database query depends on it.
2. Statistics for one's own data only, or can an admin view
   someone else's? That changes the permission check.
3. What should be shown if there is no data for the period at
   all — zeroes or a "no data" message?
```

All three are genuine forks, and your prompt closed none of them. Without the questions you would have got three decisions that were not yours and found out about them when the command behaved oddly.

A stronger variant

```
BETTER
Before writing any code: list the decisions you will have to
make on my behalf if I don't answer. For each one, say what
you would choose by default.
```

This variant is better value: it requires no answers. The AI shows a list of its decisions with the defaults, you read it and step in only where you disagree.

```
AI REPLY
I'll decide for you:
- period: the last 30 days (simpler and more usual for bots)
- access: own data only
- empty period: a "no data" message, not zeroes
- format: a list of lines, no table
```

One sentence — "all correct except the period, I need the calendar month" — and the task is settled.

WHAT TO TELL THE AI

When the technique is especially useful

- **A task in your own words, with no detail.** When you have not yet thought through how it should work.
- **An unfamiliar domain.** Integrating with an API you don't know, a format you are seeing for the first time.
- **Expensive rework.** Migrations, changes to the shape of the data, anything hard to roll back later.

When it isn't needed

If the task is already described using the template from article 3.4, there will be no questions left — and an AI that has been asked to ask will ask something formal just to comply. On clear tasks the technique only adds a step.

> **TIP**
> It pairs well with the plan (article 3.10): three questions first, then a plan that takes the answers into account, then the code. On a large task those two steps take five minutes and save an evening.

Answers can be short

There is no need to answer at length; marking them is enough:

PROMPT

```
1 — calendar month
2 — own data only
3 — "no data"
```

WHAT TO LOOK FOR

Questions for the sake of questions. "What is your naming style?", "Do you want comments?" — formal questions that change nothing. A sign the task was clear already and the technique is superfluous here.

The AI asked and then answered itself. Sometimes it poses questions and, without waiting, carries on writing code with its own assumptions. Not fatal, but the decisions were made by it — read which ones.

More than three questions. If the limit does not hold and there are a dozen, the task is stated too broadly. Break it into parts.

CHEAT SHEET

- "Ask 3 questions whose answers would most change the solution."
- The numeric limit is required, or there will be ten.
- Stronger: "which decisions will you make for me, and what are your defaults?"
- One line per answer is enough.
- On a clear task the technique is superfluous — the questions will be formal.
- A dozen questions means the task is too broad; split it.

3.12 Technique — input ⟶ output examples

two examples instead of a paragraph of description

WHAT IT IS

Instead of describing in words what a function should do, show a pair of "what came in ⟶ what should come out". Two examples remove more ambiguity than a paragraph of explanation, and they are checked instantly.

The technique is strongest where words are slow and imprecise: parsing text, formats, data transformations, edge cases.

WHAT IT LOOKS LIKE IN CODE

In words

```
DON'T DO THIS
Write a function that normalises a phone number
into international format
```

What does "normalise" mean? Which input formats? What about spaces and brackets? What about a number with no country code? The AI will close all of those forks itself.

With examples

```
BETTER
Function normalize_phone(raw: str) -> str | None.
Examples:

"+47 912 34 567"   -> "+4791234567"
"47-912-34-567"    -> "+4791234567"
"912 34 567"       -> "+4791234567"   (no code — assume Norwegian)
"+7 999 123-45-67" -> "+79991234567"
"not a phone"      -> None
""                 -> None

The default country code comes in as an argument, default_cc="47".
```

Six lines close everything: the output format, what to do with separators, the behaviour without a country code, handling garbage. In words the same would have taken half a page, and something would still have been left unsaid.

The examples double as tests

```
PROMPT
Turn these examples into a parameterised pytest test.
```

One prompt and the check is ready. The examples you explained the task with become the thing that verifies it. This is the cheapest way to get the tests from article 3.9.

WHAT TO TELL THE AI

How many examples are needed

Three to five. One shows the happy case and says nothing about boundaries. Ten are too many and start contradicting each other.

Be sure to include:

- **the ordinary case** — the thing it is all for;
- **a boundary case** — an empty string, zero, a single record, the maximum length;
- **a failing case** — garbage input, and what should happen then.

The third is skipped most often and decides the most: it is what tells the AI whether to return `None`, raise an exception, or substitute a default.

The notation

An arrow and alignment read better than prose:

```
input              -> output
```

If the output is complex, show it as it is, with its structure:

PROMPT

```
Input:  {"cod": "404", "message": "city not found"}
Output: None

Input:  {"main": {"temp": 4.2}, "weather": [{"main": "Rain"}]}
Output: ("Rain", 4.2)
```

> TIP
> When the data format is unfamiliar, ask for the reverse: let the AI invent five example inputs and say what it thinks should come out. You correct two of them — and the task ends up described more precisely than if you had written the examples yourself.

Where the technique doesn't fit

Anywhere the result cannot be written as a string: interfaces, architecture, "make it look good". For those, the plan (3.10) and the questions (3.11) are what work.

WHAT TO LOOK FOR

Code fitted to the examples. The most insidious failure: the function contains `if raw == "+47 912 34 567": return ...`. It looks like it works,

passes all your examples and falls apart on the seventh input. Read the code, not only the result.

Examples that contradict each other. This happens when there are many and they were written in haste. The AI will quietly pick one reading. If you spot a contradiction in the answer, fix the examples, not the code.

Edge cases decided for you. If you didn't say what to do with an empty string, the code will contain a decision you never made. Check which one.

CHEAT SHEET

- Two or three "input → output" examples instead of a paragraph.
- Required: the ordinary case, a boundary case, a failing case.
- The failing case decides what happens on garbage input.
- Examples turn into a test with one prompt.
- Check that the code solves the task rather than fitting the examples.
- For tasks with no string-shaped result the technique doesn't apply.

3.13 Technique — limit the blast radius

the less we touch, the easier it is to check

WHAT IT IS

By default an AI assumes it may change anything within reach. You asked for a function to be fixed — and along the way variables in the neighbouring one have been renamed, imports reordered, and error handling "improved" where nobody asked.

The technique is to name the boundaries in advance. The benefit is not tidiness but checkability: **a ten-line diff you will read, a two-hundred-line diff you will not.** And an unread diff is code you do not control.

WHAT IT LOOKS LIKE IN CODE



Fig. 3.2 · The narrower the zone, the smaller the diff — and the likelier you are to actually read it.

Sentences that work

PROMPT
```
Change only the format_report function. Don't touch the rest
of the file.
```

PROMPT
```
Changes only in handlers/weather.py. If something else needs
changing, say so, but don't do it.
```

PROMPT
```
Don't touch imports, formatting or the names of existing
variables. Only the logic inside the named function.
```

The third removes the most common source of noise: AIs love reordering imports and tidying up style, and those changes take up more of the diff than the real edit does.

The other half of the technique

Limiting the zone is not enough — you have to give a way out if the boundaries turn out to be wrong:

BETTER
```
Change only fetch_weather. If the task cannot be solved without
touching anything else, stop and tell me what else needs
changing and why. Don't do it yourself.
```

Without that sentence the AI, on hitting the limit, will either break it quietly or produce a worse solution just to stay inside it. With it, it asks — and you decide.

WHAT TO TELL THE AI

Boundaries come in three sizes

Zone	When	How to say it
Function	a pinpoint fix, a bug	“only the function X”
File	a feature in one place	“changes only in file Y”
A set of files	a connected change	“you may touch A, B and C, nothing else”

Start with the narrowest that looks sufficient. Widening it when the AI asks is easy; narrowing it after it has rewritten half the project is not.

> **REMEMBER**
> The size of a diff is not a question of tidiness but of control. You are answerable for code you did not read exactly as much as for code you wrote.

Checking the boundaries by the diff

After a change, look first not at the contents but at the size:

```
git diff --stat
```

One line of output where you expected one — good. Seven changed files where you expected one — a signal to stop and read carefully, whatever the reply said.

WHAT TO LOOK FOR

Boundaries broken quietly. Checked not by reading the reply but with `git diff --stat`. In its text the AI usually says it did only what was asked.

Imports reordered, formatting changed. Harmless in itself, but it inflates the diff and hides the real changes in it. Forbid it in its own sentence.

The boundaries turned out too narrow. If the AI reports three times running that the task cannot be done within the limits, it is probably right. Widen the zone deliberately rather than dropping the constraint altogether.

The change smeared across the file. Even inside the permitted zone the edits may land in ten places instead of one. Ask: “make the change in one place rather than patching several”.

CHEAT SHEET

- Name the zone in advance: function · file · set of files.
- Start with the narrowest; widening is easier than rolling back.
- Forbid touching imports and formatting in a separate sentence.
- Always add: “if it can’t be done, say so, don’t do it yourself”.
- Check the boundaries with `git diff --stat`, not by the AI’s reply.
- An unread diff is your responsibility.

3.14 The project context file

write it once instead of explaining it every day

WHAT IT IS

A context file is a text file at the root of the project that the AI reads at the start of work. It holds what otherwise has to be explained in every new conversation: the stack and the versions, the project structure, the conventions, the commands to run.

The file closes reason 2 from article 3.1 — "it cannot see" — once and for all. The sign that you need one: you are typing the same thing about your project into a chat for the second time.

Its name varies by tool: `CLAUDE.md`, `AGENTS.md`, rules in `.cursor/rules/`. The idea is the same, and the contents move between them with almost no edits.

WHAT IT LOOKS LIKE IN CODE

```
# Weather subscription bot

## Stack
Python 3.12, aiogram 3.x, SQLite via aiosqlite.
Dependencies in requirements.txt, virtual environment in .venv.

## Structure
bot.py          entry point, starts polling
handlers/       command handlers, one file per topic
storage.py      database work, all queries live here
db.py           connection, get_conn()
config.py       reading environment variables

## Conventions
- All database access goes through functions in storage.py.
  No SQL in the handlers.
- Handlers are async; heavy operations must not block the loop.
- User-facing texts live in locales/en.py, not in the code.
- Logging via logging, not print.

## Running it
python -m venv .venv && .venv/bin/pip install -r requirements.txt
python bot.py              run
pytest -q                  tests

## What not to do
- Don't add dependencies without need.
- Don't touch migrations/, they are applied in order.
- Tokens come from environment variables only; they are not in the code.
```

That is the whole file, not an excerpt. It is short — and that is its value.

WHAT TO TELL THE AI

Five sections, and that is enough

Section	What it answers
Stack	what it is written in and **in which versions**
Structure	where things live
Conventions	how things are done here
Running it	which commands to use
What not to do	where not to go

The "Stack" section with versions is the most important: it closes reason 3 from article 3.1, the one that cannot be cured by adding detail to a prompt.

The "Conventions" section comes second for return on effort. It is what makes the AI write in the project's style rather than its own.

The main rule: keep it short

The file is read at the start of every conversation and takes up room in the context. A long file crowds out code the AI could have read, and attention spreads thin.

A guide: one or two pages. If it doesn't fit, something went in that the AI could find out for itself by reading the project.

Write	Don't write
versions and conventions	what is visible in the code
where things live	lists of every function
what not to touch	project history and plans
commands to run	documentation for libraries

CAREFUL

A context file goes stale silently. A module moved, a framework version changed — and the file still says the old thing, so the AI writes confidently against a wrong description. That is worse than having no file: a wrong description gets believed.

How to keep it current

The easiest way is to update it with the same AI that works in the project:

PROMPT

```
Read CLAUDE.md and compare it with what is in the project now.
What in the file is out of date? Just a list of mismatches,
no edits.
```

A good moment for that check is when you have added a module, changed a library version, or changed how the project is started.

WHAT TO LOOK FOR

The file exists and the AI doesn't read it. In a chat you have to paste it yourself; it will not get there on its own. Terminal agents usually read it, but not all of them and not always — check by asking: "what do you know about the structure of this project?"

The file describes the wish, not the reality. "All database access goes through storage.py" — while in fact three handlers have SQL right in the code. The AI will treat what is written as true.

The file has grown to ten pages. Documentation, a history of decisions and a list of every endpoint have moved in. Move those to separate files and leave links in the context file.

CHEAT SHEET

- Five sections: stack · structure · conventions · running it · what not to do.
- Versions are mandatory.
- One or two pages. Longer and it crowds code out of the context.
- Don't write what the AI can read in the code itself.
- A stale file is worse than none: it gets believed.
- Compare the file against the project after every notable change.

3.15 How to write a mini-spec

for a task you would hate to redo

WHAT IT IS

A mini-spec is the third level of detail from article 3.3, written out in full. You need it where a task touches several files, changes the shape of the data, or rolls back badly.

It is the same four parts of a prompt (article 3.2), simply unfolded. There is nothing new to learn — you just carefully write down what is usually kept in your head.

WHAT IT LOOKS LIKE IN CODE

The structure

```
Task          one sentence: what should come into being
Why           which problem we are solving
Context       how it works now, what already exists
What to do    point by point, in execution order
Constraints   what not to do, what not to touch
Done when     checkable conditions, as a list
Out of scope  what we deliberately are not doing now
```

Two of the seven sections are usually skipped, and they are the most useful.

"Why" gives the AI room to propose something better than your idea. If it only knows *what* to do, it does it literally; if it knows *why*, it can say "there is a simpler way to get that".

"Out of scope" reins in the appetite. Without it the AI builds the task up into the "proper" one: you asked for a cache and got a cache with invalidation, metrics and config-driven tuning.

A complete example

PROMPT

```
Task: move subscription storage from a JSON file to SQLite.

Why: with 300+ subscriptions the whole file is rewritten on every
change and the bot stalls for a second. Also, a failure during the
write corrupts the file.

Context: subscriptions are in data/subs.json, shape
{user_id: [city, ...]}. Read by load_subs(), written by save_subs(),
both in storage.py. Five call sites, all in handlers/.
SQLite is already used in the project for logs, connection via
get_conn() in db.py.

What to do:
1. A subscriptions table: user_id INTEGER, city TEXT,
   created_at TEXT. Unique on the pair (user_id, city).
2. Rewrite load_subs/save_subs on SQLite.
   Don't change the signatures or the return types.
3. Migration on startup: if data/subs.json exists and the table
   is empty, move the data and rename the file to subs.json.bak.

Constraints:
- the sqlite3/aiosqlite module only, no ORM
- don't touch the handlers at all
- don't change the log format

Done when:
- on an empty database the bot starts without errors
```

```
- subscriptions from the old JSON are visible after a restart
- all five call sites work without edits
- a second start does not duplicate the data

Out of scope: indexes, query optimisation, moving the logs,
admin commands for viewing subscriptions.
```

Twenty-five lines — and a task that would otherwise be redone three times gets done on the first attempt.

WHAT TO TELL THE AI

How long it takes

Ten to fifteen minutes. It pays for itself at the first fork the AI would otherwise have closed on its own.

If a mini-spec takes an hour, the task is too big. Split it in two and write the spec for the first half.

A good combination

Mini-spec + plan (article 3.10) is the most reliable pairing for a large task:

PROMPT

```
<mini-spec>

Don't write code yet. Lay out a plan for how you will do this,
and tell me where my spec is missing information.
```

The second sentence is the valuable one: the AI points at the holes in your spec before they turn into wrong code.

> TIP
> Write the mini-spec in a file, not in the chat. Keep it next to the project — `docs/tasks/2026-10-subs-to-sqlite.md`, say. A month later it is the only place that says why the structure is the way it is.

When a mini-spec isn't needed

For anything that rolls back with one command. A mini-spec for adding a function is bureaucracy, not care.

WHAT TO LOOK FOR

"Done when" written in adjectives. "Works reliably", "fast", "convenient" are not criteria. A condition you cannot check constrains nothing (article 3.2).

The "Context" section retelling the code. There is no need to copy functions in — saying where to look is enough. The AI will read it itself if it has access to the project.

A spec that was never compared against the result. The main value of a spec shows at the end: walk the "Done when" list and tick every line. That is two minutes and the only way to know whether it is done.

The task grew along the way. New points appeared that were not there — that is not a reason to extend the spec on the fly. Finish the current one, then state a second task.

CHEAT SHEET

- Seven sections, two of them usually forgotten: "why" and "out of scope".
- "Why" lets the AI propose something better than your idea.
- "Out of scope" stops it building the task up into the "proper" one.
- "Done when" — checkable conditions only, not adjectives.
- Mini-spec + "lay out a plan and tell me what is missing".
- Keep the spec in a file next to the project, and tick the list at the end.

3.16 Prompt antipatterns

seven ways to waste your time

WHAT IT IS

Prompts that look fine and fail regularly. Each is easy to recognise by its wording — and each has a short replacement.

The article closes chapter 3: where the previous fifteen said how to do it, this one says how not to, and why.

WHAT IT LOOKS LIKE IN CODE

1. "Make it nice"

DON'T DO THIS

```
Refactor this module, make it nicer
```

Nice is undefined, so the AI supplies its own: a different architecture, different names, a different style. The diff will be enormous and the behaviour will drift.

Replacement: name the specific change of shape. "Split the function in three", "lift out the repeated block", "rename `d` to `user_data`".

2. "Fix everything"

DON'T DO THIS

```
Go through the file and fix all the problems
```

The problems are not named, so the AI will go looking — and will find some even if there are none (the same trap as in reviews, article 3.8).

You get twenty changes, half of them unnecessary and two of them breaking something.

Replacement: ask for a list of findings first, read it, choose — and only then ask for changes.

3. "You got it wrong"

DON'T DO THIS

```
No, you got it wrong. Fix it.
```

What is wrong has not been said. The AI will agree — it almost always agrees — and change something at random. Sometimes something that worked.

Replacement: facts instead of a verdict. "Ran it, got `KeyError: main` on line 11. Expected the reply 'City not found'."

CAREFUL

The AI agreeing means nothing. It agrees when it is right and when it is not. If it "admitted the mistake" without explaining what the mistake was, it was most likely just keeping the conversation going.

4. "Why doesn't it work?"

DON'T DO THIS

```
Why doesn't it work?
```

The AI sees neither the run, nor the error, nor the environment. It will start guessing and produce five theories, all of which you then have to check.

Replacement: the template from article 3.5 — what you did, what you expected, what you got, the whole traceback.

5. The same prompt, but louder

DON'T DO THIS

```
I SAID no external libraries!!!
```

Repeating with force adds no information. If the first prompt was misunderstood, an identical second one will be misunderstood identically.

Replacement: rephrase rather than repeat. Name the specific library it added and what to use instead. On a long thread, start a new session (article 5.4): sometimes the conversation is already "contaminated" with wrong context, and a clean start is cheaper.

6. A task made of five tasks

DON'T DO THIS

```
Add a /stats command, fix the time zone bug, move the texts
into a separate file and write tests
```

Four tasks in one prompt give one large diff nobody can make sense of. Some of it is done well, some badly, and none of it can be rolled back separately.

Replacement: one at a time. Accepted — next. This is not slower: the time goes on checking, not on prompting, and small pieces are faster to check.

7. Politeness instead of content

DON'T DO THIS

```
Hi! Hope you're doing well. Could you possibly, if it's not
too much trouble, take a look at my code and maybe suggest
something? I'd be very grateful!
```

Politeness does no harm, but it does no good either — and here it took the space where the file, the version and the question should have been. The same goes for apologies and long explanations of why you don't know the subject.

Replacement: get to the point. Talk however you like, but make sure there is actual content in the prompt.

WHAT TO TELL THE AI

A summary: on the left what you feel like writing, on the right what works.

Antipattern	Replacement
"make it nice"	name the specific change of shape
"fix everything"	a list of findings first, then a choice
"you got it wrong"	facts: what you ran, what you got
"why doesn't it work?"	what I did · what I expected · what I got
the same prompt, louder	rephrase, or start a new session
five tasks in one	one at a time, with a check in between
politeness instead of substance	the file, the version, the question

All seven have one thing in common: **the prompt contains no information the AI didn't already have.** Irritation, repetition, politeness and generalities don't add any — only facts, file names, versions, examples and criteria do.

WHAT TO LOOK FOR

The conversation is going in circles. A third attempt to explain the same thing is a signal to stop. Either the task is unclearly stated or the session's context has accumulated junk. A new session with a properly stated task is almost always faster.

You feel like arguing. Arguing with an AI is wasted time: it will yield, and the solution will be no better for it. Instead of arguing — facts and a rephrase.

The prompt is longer than the task. Three paragraphs for a one-line change means you are explaining your feelings about the task rather than the task.

CHEAT SHEET

- "Make it nice", "fix everything", "you got it wrong" — no information in the prompt.
- The AI agreeing does not mean there was a mistake.
- Repeating with force doesn't work: rephrase.
- Five tasks in one prompt give one uncheckable diff.
- Going in circles on the third attempt — start a new session.
- Arguing is pointless: it yields, the solution doesn't improve.

4 Checking and debugging

The third skill. The code is written and has even started — and that means nothing yet. This chapter is about making sure you got what you intended, and what to do when you didn't.

4.1 The rule of trust

what is always checked and what depends on the case

WHAT IT IS

Checking every line is impossible — the work would stop. Checking nothing is not allowed — the project stops being yours. You need a level in between, and it is set by a rule.

The same things are always checked, whatever the task. Everything else goes by the cost of a mistake (article 1.5).

THE FOUR CHECKS THAT ARE ALWAYS DONE

They take a couple of minutes and catch most of the trouble.

1. Run it
Obvious, and skipped anyway. Code that has not been run does not work — that is not pessimism, it is statistics. And you run the case the whole thing was for, not just any case.

2. The diff
Look at what changed, not at what came out (article 2.13). Start with `--stat`: how many files, how many lines. A mismatch with your expectation is a reason to stop.

```
git diff --stat
```

3. An unhappy case
At least one: empty data, an unreachable network, garbage input. That is where the errors live that a run does not find, because what gets run is usually the happy path.

4. The question “what does this do”

If you cannot explain the accepted code in two sentences, it is not yours yet (article 1.2). You can explain it to yourself, out loud, or in the commit message.

> **REMEMBER**
> The four checks take two minutes and do not depend on the size of the change. If there is no time for them, there was no time for the change either.

WHAT TO TELL THE AI

Three of the four checks the AI can prepare itself:

PROMPT

```
Name three cases worth checking by hand to be sure
this change works. At least one of them failing.
```

PROMPT

```
Explain in two sentences what the code you just added does.
```

The fourth — the diff — is yours alone: that is the reviewer’s job.

WHAT GETS CHECKED ON TOP

Beyond the four mandatory ones, the level of checking is set by the task.

What the task looks like	What to add
Touches user data	a rehearsal on a copy, a way back
Irreversible	a plan in advance (3.10), a mini-spec (3.15)
The error would be invisible	tests (4.6), logs at the key points
A rare library	a check against the documentation (4.8)
More than one file	reading the diff line by line
Editing code that has worked for a long time	a test before the change, to pin the behaviour down

That last line is the most underrated technique. A test written **before** a change records what used to work and tells you at once if that broke.

WHAT TO LOOK FOR

Checking came down to running it. The most common form of self-deception: ran it, answer looked right, accepted. One scenario out of ten has been checked (article 1.6).

Nobody looked at the diff. If a change is accepted on the strength of the AI's reply rather than the diff, the reviewer's chair is empty.

The level of checking chosen by feel. "This one looks simple" is not a criterion. The criteria are reversibility and visibility.

Checking postponed to the end. Five changes in a row, then a check of everything at once — and now it is unclear which of the five broke it. Check after each one.

CHEAT SHEET

- Always: run it · the diff · an unhappy case · "what does this do".
- The four checks take two minutes and don't depend on the size of the change.
- Beyond that — by the cost of a mistake: reversibility and visibility.
- Editing long-working code — first a test for how it worked.
- Check after every change, not after five.

4.2 Running it and reading the logs

what the program tells you about itself

WHAT IT IS

A log is what a program writes about itself while it runs. For debugging it matters more than a traceback: a traceback shows the moment of the crash, a log shows what was happening before.

For a bot or a server the log is the only way to find out what happened at all: you don't see the user's screen and you can't ask them again.

WHAT IT LOOKS LIKE IN CODE

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s: %(message)s",
)
log = logging.getLogger(__name__)


def fetch_weather(city: str) -> dict:
    log.info("weather request: %s", city)
    try:
        data = call_api(city)
    except TimeoutError:
        log.warning("timeout for city %s, retrying", city)
        data = call_api(city)
    log.debug("API response: %s", data)
    return data
```

Output:

```
2026-10-14 13:02:11 INFO handlers.weather: weather request: Oslo
2026-10-14 13:02:12 WARNING handlers.weather: timeout for city Oslo,
retrying
```

Five levels

Level	When	Visible by default
`DEBUG`	details for debugging	no
`INFO`	the normal course of work	yes
`WARNING`	odd, but we carry on	yes
`ERROR`	we failed to do what was asked	yes
`CRITICAL`	we cannot carry on	yes

The line `level=logging.INFO` means: show `INFO` and anything more serious. To see `DEBUG`, the level is changed to `logging.DEBUG` — usually temporarily, while hunting an error.

How to read a log

Bottom to top, up to the first surprise. The last lines are consequences; the interesting part is usually above.

Look at the times. A gap of several seconds between neighbouring lines shows where the program was waiting. It is the cheapest way to find out what is slow.

Look for a `WARNING` before an `ERROR`. The warning often names the real cause, while the error is only its consequence.

Count the repeats. The same line a hundred times in a row is a loop that should not have looped, or retries that never end.

`print` or `logging`

`print`	`logging`
in a single script, while you debug	in a bot, a server, anything long-running
disappears when the program isn't in a terminal	goes to a file, survives a restart
no levels and no timestamps	levels, time, module name

The rule: while you are hunting an error by hand — `print` (article 4.5). Anything that stays in the code — `logging`.

> **TIP**
> Ask the AI for logging at the moment the function is written: "add logging: INFO on entry with the arguments, WARNING for every abnormal situation". That is one sentence, and when something breaks on the server it pays for itself entirely.

WHAT TO TELL THE AI

PROMPT

```
Add logging to this function: INFO on entry with the key
arguments, WARNING for abnormal situations, ERROR on failure.
Don't log tokens or personal data.
```

When the log already exists and you need to make sense of it:

PROMPT

```
Here is a chunk of the log. What happened here? What should
I look at first?

<log>
```

When there isn't enough logging:

PROMPT

```
I don't understand why this function returns an empty list.
Where should I add logging to make it visible?
As few lines as possible.
```

WHAT TO LOOK FOR

Secrets in the logs. A token or password written to the log is the same secret in plain sight, only in a file that then gets forwarded. Check what is being logged before handing a log to anyone, the AI included.

A log with nothing in it. If the log is empty when an error occurs, logging was forgotten — and the only way to learn the cause is already gone.

A log with everything in it. The opposite extreme: `DEBUG` on everything in production. The file grows, the important parts drown, the disk fills.

A message with no data. `log.error("error")` says nothing. A good entry holds values: which city, which user, which response code.

Logging inside a loop over a thousand items. A thousand lines for one request. Log the outcome, not every step.

CHEAT SHEET

- The log matters more than the traceback: it shows what came before.
- Five levels: DEBUG · INFO · WARNING · ERROR · CRITICAL.
- Read bottom to top until the first surprise; watch the time gaps.
- A `WARNING` before an `ERROR` is often the real cause.
- `print` while debugging, `logging` for anything that stays in the code.
- Always include values in the message. Never log secrets.

4.3 Preparing an error to be fixed

a minimal reproduction instead of a pile of detail

WHAT IT IS

Article 3.5 says **how to word** a fix request. This one is about **what to gather** before writing it.

The difference matters: a well-worded request with poor evidence will not help. And good evidence is not “as much as possible” but the opposite: the smallest case on which the error still reproduces.

WHAT IT LOOKS LIKE IN CODE

The error showed up in a bot: the `/stats` command crashes sometimes. Here is what you do with it before calling the AI.

Step 1. Reproduce it

An error you cannot repeat cannot be fixed — its fix cannot be verified. If a command crashes “sometimes”, you have to find out on which data.

```
for user_id in [1, 2, 3, 17, 42]:
    try:
        build_stats(user_id)
    except Exception as e:
        print(user_id, "->", type(e).__name__, e)
```

Three lines of iteration often turn “crashes sometimes” into “crashes on user 17”.

Step 2. Trim it

Now remove everything from the case that has no bearing. Telegram, the database, the handlers — if the error reproduces without them, they are not involved.

```
# Minimal case: no bot, no database
rows = [{"date": "2026-10-01", "amount": 100},
        {"date": "", "amount": 50}]
print(build_stats_from_rows(rows))     # crashes
```

A trimmed case is doubly valuable. It points at the cause more precisely, and it gives you a ready test: it is exactly what you later ask to be turned into `pytest` (article 3.9).

Step 3. Gather three things

What	Why
The minimal case	what it reproduces on
The whole traceback	where it crashed (article 2.12)
What was expected	to tell a bug from intended behaviour

And only now — the request, using the template from article 3.5.

WHAT TO TELL THE AI

Trimming the case can be done with the AI too; it is good at that:

PROMPT

```
Here is a function and the data it crashes on. Shrink the data
to the smallest example on which the error still reproduces.
Don't fix the code, I only need the example.
```

When the error is intermittent:

PROMPT

```
This function crashes about one time in five, on the same
arguments. What can cause inconsistent behaviour?
Name the three most likely.
```

Inconsistency almost always means one of three things: an external source (network, time, random numbers), shared state between calls, or parallel execution. The AI will name them, but checking is on you.

> **TIP**
> While trimming the case, write down what you removed. If the error suddenly stops reproducing, the last thing removed was the important one. That is half the job done.

WHAT NOT TO HAND OVER

The whole file when the function would do. Extra context spreads attention thin — yours and the AI's.

A day of logs. What is needed is the chunk around the error: a few lines before and a few after.

Secrets. Tokens, keys, users' personal data. Before sending a log, check what is in it (article 4.2).

Your diagnosis instead of the facts. "I think the problem is in the database" leads the AI astray if the guess is wrong. Facts separately, guesses separately — and mark the guesses as guesses.

WHAT TO LOOK FOR

An error "fixed" that never reproduced. If you could not reproduce it before the change, you cannot verify that it is gone. Reproduction first, fix second.

The minimal case not kept. Trimmed it, fixed it, threw it away — and a month later, on a similar error, you start over. A trimmed case is worth turning into a test right away.

The symptom fixed instead of the cause. The pairing of articles 2.12 and 2.13: the crash site and the cause are different places.

The fix not checked against the original case. The most galling outcome: fixed, tests green, and that very user 17 still crashes it.

CHEAT SHEET

- First reproduce, then trim, then call the AI.
- An error that doesn't reproduce can't be fixed: there is nothing to verify with.
- The minimal case is the best prompt and a ready-made test.
- Three things: the case · the whole traceback · what was expected.
- Don't hand over whole files, a day of logs, or secrets.
- Mark guesses as guesses, or they will lead the search astray.

4.4 Narrowing the search

halving instead of reading everything

WHAT IT IS

When it isn't clear where the error is, reading all the code from the top is the worst way to find it. There is a more reliable technique: **split the data's path in half, check the middle, and throw away the half where everything is fine.**

That way the search area halves at every step. From a thousand lines to the right one — ten checks.

WHAT IT LOOKS LIKE IN CODE



green is checked and ruled out · red is where the error may still be

Fig. 4.1 · Every check in the middle throws away half. From a thousand lines to one — ten steps.

On a live example

A chain of four functions: load the rows, parse them, turn them into numbers, compute the average. It crashes somewhere inside.

```
def load_rows(raw: str) -> list[str]: ...
def parse_rows(rows: list[str]) -> list[dict]: ...
def to_numbers(items: list[dict]) -> list[dict]: ...
def average(items: list[dict]) -> float: ...

print(average(to_numbers(parse_rows(load_rows(RAW)))))
```

Step 1. Print in the middle. Not at the start and not at the end — exactly halfway along the chain:

```
rows = load_rows(RAW)
items = parse_rows(rows)
print("after parse_rows:", items)
```

```
after parse_rows: [{'city': 'Oslo', 'temp': '4'},
                   {'city': 'Bergen', 'temp': '7'},
                   {'city': 'Tromso', 'temp': '-2.5'}]
```

The data is fine. So the first half of the chain is clean and can be forgotten. Two functions left out of four.

Step 2. Halve the remainder. Check `to_numbers` on that same data, item by item:

```
for it in items:
    try:
        int(it["temp"])
    except ValueError as e:
        print("fails on:", it, "->", e)
```

```
fails on: {'city': 'Tromso', 'temp': '-2.5'} -> invalid literal for
int() with base 10: '-2.5'
```

Two checks — and both the line and the specific data are found. The cause is immediately visible: the code assumed whole-number temperatures and the data has a fractional one.

Where to split

What we are after	Where the middle is
An error in a chain of functions	between the functions, by the data
An error in a long function	between the logical blocks
An error that appeared after edits	`git log` — between commits
An error on one item in a thousand	by half of the list

The third row is a technique of its own: if the code worked yesterday and broke today, what you halve is not the code but the commit history. Git has `git bisect` for that, though switching to a commit in the middle by hand is often enough.

> **REMEMBER**
> The question at each step is not “where is the error” but “is the data still intact here?”. The second question is faster to answer, and each answer throws away half.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> Here is a chain of four functions; it crashes somewhere inside.
> Where should I put one print to split the chain in half?
> Don't fix the code, just name the place.
> ```

> **PROMPT**
> ```
> I printed the data after parse_rows and it is fine.
> So the error is further on. What should I check next?
> ```

A dialogue like this saves time: the AI is good at pointing out where the middle is, and you are the one who runs it and looks.

WHAT TO LOOK FOR

Printing at the start and the end instead of the middle. The start is usually fine and the end is usually broken — neither check narrows anything.

The AI offering fixes before the area is narrowed. The typical answer to "it doesn't work" is three hypotheses at once. Any of them may be right, and all of them have to be checked. Narrow it yourself, then ask for a fix.

The area growing instead of shrinking. If every check adds new suspects, you are checking consequences rather than data. Go back to "is the data still intact here?".

Debugging without notes. You checked three places, forgot the results, and check them again. Write one line each: "after parse_rows — fine".

CHEAT SHEET

- Halve the data's path, check the middle, discard the clean half.
- The question at each step: "is the data still intact here?"
- From a thousand lines to one — ten checks.
- Broke after edits — halve the commit history, not the code.
- Printing at the start and the end narrows nothing.
- Write the results down, or you will go in circles.

4.5 Debugging with print

when it is enough and how to do it properly

WHAT IT IS

Dropping in a `print` and looking at what is in a variable is the simplest way to debug. It is conventionally treated as unserious, and for most tasks it is faster than a debugger and requires no setup at all.

This article is about printing in a way that actually helps, and about when you do need a debugger after all.

WHAT IT LOOKS LIKE IN CODE

Bad

```
print(data)
print(x)
print("here")
```

You run it, get three cryptic lines in a row and can't tell which came from where.

Good

```
print(f"[parse] items={len(items)} first={items[0] if items else None!
r}")
print(f"[to_numbers] temp={it['temp']!r}
type={type(it['temp']).__name__}")
```

Three habits that make printing useful:

A label. `[parse]` tells you at once where the line came from. With a dozen prints it saves all of the time.

`!r` instead of plain output. `repr` shows the quotes and the invisible characters. The difference is decisive:

```
value = " 42 "
print(f"value={value}")      # value= 42
print(f"value={value!r}")    # value=' 42 '
```

In the second case you can see immediately that the string has spaces in it — and those spaces are exactly why `int()` was failing.

The type, not only the value. `'4'` and `4` look almost the same and behave differently. `type(x).__name__` settles the question.

> **TIP**
> The habit of writing `!r` in debug prints pays off constantly. An empty string, `None`, a string instead of a number, a stray space — all of it is only visible through `repr`.

Printing a structure

For dictionaries and lists this is more convenient:

```
import json
print(json.dumps(data, ensure_ascii=False, indent=2)[:500])
```

`ensure_ascii=False` keeps non-English text readable instead of showing escape codes. The `[:500]` slice keeps the terminal from flooding when there is a lot of data.

When you need a debugger

Printing is enough while you know what to print. A debugger is for when you don't:

Print	Debugger
you know which variables to look at	you don't know what is going on at all

one or two passes	you need to go step by step
the code runs quickly	every run is slow
everything is in your own code	the error is somewhere in a library

The cheapest debugger in Python is one line in the right place:

```
breakpoint()
```

The program stops and you land in an interactive shell where any variable can be inspected. `c` continues, `n` is the next line, `q` quits.

WHAT TO TELL THE AI

PROMPT

```
I don't understand why this ends up as an empty list.
Where should I put two prints to make that visible?
Don't change the code, just name the places and what to print.
```

PROMPT

```
Add debug prints to all three functions: a label, the key
values through !r, and their types. I'll remove them later.
```

And the mandatory final step:

PROMPT

```
Remove all the debug prints we added.
Don't touch the log.* logging.
```

WHAT TO LOOK FOR

Prints left in the code. The classic: debugged it, forgot to remove them, it shipped to the server and now clutters the output. Before committing, search the diff for `print(`.

Print instead of logging in long-lived code. In a bot or on a server `print` disappears along with the terminal. There you need `logging` (article 4.2).

Printing in the wrong place. If the output doesn't change for clearly different inputs, the print is probably in the wrong branch — or execution never reaches it.

Secrets in a print. `print(config)` together with the token, and now it is in the terminal history, or in a screenshot sent to a chat.

CHEAT SHEET

- A label at the start of the line: `[parse] ...`.
- Always `!r` — it shows spaces, quotes and `None`.
- Print the type, not only the value.
- Dictionaries — via `json.dumps(..., ensure_ascii=False)` and a slice.
- Don't know what to look at — `breakpoint()`.
- Before committing, check the diff for leftover `print(`.

4.6 Tests — read them and run them

what red output means and what it actually believes

WHAT IT IS

How to ask for tests is article 3.9. This article is about what to do with the ones you got: how to read `pytest` output, what counts as a real check, and why green tests do not mean everything is fine.

You will not have to write tests yourself. You will definitely have to read them — otherwise you are trusting a check you never checked.

WHAT IT LOOKS LIKE IN CODE

A function splits a long message into chunks. The tests were written from the list of cases in article 3.9.

```
pytest -q
```

```
...F
[100%]
================================== FAILURES
===================================
_____________________ test_paragraph_longer_than_limit
______________________

    def test_paragraph_longer_than_limit():
        text = "y" * 300
        parts = split_message(text, limit=100)
>       assert all(len(p) <= 100 for p in parts)
E       assert False

test_split.py:23: AssertionError
============================ short test summary info
============================
FAILED test_split.py::test_paragraph_longer_than_limit - assert False
1 failed, 3 passed in 0.02s
```

How to read it

The first line is the map. `...F` means three tests passed and the fourth failed. A dot is a pass, `F` a failure, `E` an error in the test itself, `s` skipped.

`>` marks the line it failed on. Here, the `assert`.

`E` says what didn't match. `assert False` means the condition turned out false.

The last line is the bottom line. `1 failed, 3 passed` is the most important part, and a fine place to start.

What the failing test showed

The function cannot handle a paragraph that is itself longer than the limit: it can't cut it and hands it back as is. The other three tests are green — on ordinary data everything works.

This is exactly the case marked in article 3.9 as "the one people usually forget". The test caught it — which is the whole point.

> **REMEMBER**
> A failing test is good news. It found the problem now rather than a month from now at a user's. The bad news is green tests that check nothing.

Useful flags

```
pytest -q                       # brief
pytest -x                       # stop at the first failure
pytest -k "split"               # only tests with "split" in the name
pytest test_split.py::test_empty  # exactly one test
pytest -q --tb=short            # a short traceback
```

The second and fourth are the workhorses while fixing: no need to wait for the whole suite each time.

WHAT TO TELL THE AI

> **PROMPT**
> Here is the pytest output. What exactly didn't match and why?
> Don't fix the code yet.
>
> <output>

When a test fails after your change:

> **PROMPT**
> This test passed before my change and fails after.
> I only changed function X. What in it could cause that?

And a quality check on the tests themselves:

PROMPT

```
Do these tests really check behaviour, or do they check that
the code equals itself? Are any of them going to pass against
any implementation at all?
```

WHAT TO LOOK FOR

A test that cannot fail. The check: break the code deliberately and run it. If the test stays green, it checks nothing.

`assert result is not None` where `None` is impossible. Looks like a check, checks nothing.

Mocks instead of checking. If half the system is replaced with stubs, the test verifies that the stub returned what was put into it.

A test that repeats the implementation. If the test reproduces the function's algorithm line by line, it will break on any refactoring, even a correct one.

All cases happy. Not a single test with empty data, `None` or garbage input — the set is incomplete, and that is exactly where bugs live.

Green tests instead of reading the code. Tests check what is written in them and nothing more. They complement reading the diff; they do not replace it.

CHEAT SHEET

- `...F` is the map of the run: a dot is a pass, `F` a failure.
- `>` is the failing line, `E` says what didn't match, the last line is the bottom line.
- A failing test is good news; green and empty ones are the dangerous kind.
- To check a test: break the code, the test must go red.
- `pytest -x` and `pytest -k` save time while fixing.
- Not one unhappy case — the set is incomplete.

4.7 The acceptance checklist

in what order to look at a change

WHAT IT IS

Article 2.14 lists **what** to look for in code. This one is about the **order**: in what sequence to look at a change so you neither drown nor miss the main thing.

Order matters because attention runs out. Start with the small stuff and there will be none left for the serious parts.

SIX STEPS IN ORDER

1. The size of the change

```
git diff --stat
```

Does it match your expectation? One line where you expected one — carry on. Seven files where you expected one — stop and work out why before reading the contents (article 3.13).

2. What was removed

Look at the `-` lines before the `+` lines. Removal is more dangerous than addition: added code is visible, a vanished check is not.

Typical losses: `if not data: return`, length checks, edge handling.

3. Behaviour in the unhappy case

One question: what happens if the data isn't what was expected? If the change introduced `.get()` with a default, an `except`, or substituted values, check whether the error is being hidden rather than fixed (articles 2.8, 2.14).

4. Boundaries

Does the change do what was asked and nothing beyond? Logging, caching, retries nobody ordered are unchecked code in your project.

5. Run it

The case the whole thing was for, plus one unhappy one. If there are tests, run them (article 4.6).

6. Explanation

Can you explain the change in two sentences? If not, go back to reading, or ask (article 3.7).

> **TIP**
> If the change is large and there isn't attention for all of it, stop after step 2. The size and the removed lines catch more problems than everything else put together.

WHAT TO TELL THE AI

The checklist can be handed to the AI too, with a caveat: about its own code it is lenient (article 3.8). So — in a clean session:

PROMPT

```
Here is a diff. I didn't write it; I need an acceptance review.
Go in order:
1. What was removed and why that might have mattered
2. What happens on unexpected data
3. What in the change goes beyond the stated task

The task was: <what was asked>.
If there are no problems, say so plainly.
```

For yourself, something else is useful — recording what you checked:

PROMPT

```
Turn this change into a list of what I should check by hand.
Short, as bullet points, so I can tick them off.
```

WHAT NOT TO CHECK

So that attention is left for what matters, some things are worth deliberately leaving alone:

Don't check by hand	Why
formatting and indentation	that is the formatter's job, not yours
naming style	if it reads, it is enough
micro-optimisations	until there is a speed problem, this is not a problem
"I would have written it differently"	differently does not mean worse

The last point matters most. Arguing with an AI about taste is wasted time: it will yield and the code will be no better.

WHAT TO LOOK FOR

Checking that starts with the contents rather than the size. You read two hundred lines carefully when you should have stopped at `--stat` and asked a question.

Removed lines scrolled past. The most common way to miss a real problem.

The checklist turned into a ritual. Six steps in a row with no questions means you are looking without seeing. A proper acceptance includes at least one question.

Acceptance postponed. Five changes, then an examination of everything at once. Now go and find which of the five.

CHEAT SHEET

- The order: size · removals · unhappy case · boundaries · run · explanation.
- Start with `--stat`, not with the contents.
- Read the `-` lines before the `+` lines.
- Short on attention — stop after step 2, it catches the most.
- Don't check formatting, naming or matters of taste.
- An acceptance without a single question is not an acceptance.

4.8 Hallucinations

invented libraries, methods and parameters

WHAT IT IS

A hallucination is when the AI confidently describes something that does not exist: a library, a method, a parameter, a field in an API response. It looks like ordinary code, reads like ordinary code, and fails when you run it.

The phenomenon is not random but predictable: it appears where the examples were scarce — rare libraries, recent versions, narrow APIs (article 1.3).

WHAT IT LOOKS LIKE IN CODE

```
async with aiohttp.ClientSession() as session:
    data = await session.fetch_json(url)      # no such method
```

Plausible: `requests` has `.json()`, plenty of clients have `fetch`. The combination looks natural — and does not exist.

Three kinds, in descending order of visibility:

Kind	How it shows up
An invented package	`ImportError` on startup — if you are lucky; if someone has already taken the name, there is no error at all
An invented method	`AttributeError` on the call — noticed when that branch runs
An invented parameter	usually a `TypeError` in Python; silent wherever parameters go unchecked

The first and the third are more dangerous than they look.

The package. The names an AI invents repeat from answer to answer, and people register them in advance on PyPI and npm — with their own code inside. Then `pip install` succeeds, the import works, and the

package does whatever was put in it. “The package exists” does not mean “it is the right package”.

The parameter. A Python function complains about an unknown argument straight away:

```
await session.get(url, timeout_ms=5000)
```

```
TypeError: ClientSession._request() got an unexpected keyword argument
'timeout_ms'. Did you mean 'timeout'?
```

It passes silently where nobody checks parameters: in a JavaScript options object (`{ timeoutMs: 5000 }`), in `**kwargs` passed on unchecked, in YAML and JSON configs. No error, the default applies, and the behaviour is not what you intended.

How to check

A package:

```
pip index versions aiohttp
```

If the package does not exist, the command says so. It also shows the available versions.

That is only half the check. The other half is whether it is the right package: open its page on pypi.org or npmjs.com and look at how old it is, how many downloads it has and whether it links to its source. A package created last week under a name the AI suggested is a reason to stop.

A method or a field:

```
python3 -c "import aiohttp; print([m for m in dir(aiohttp.ClientSession)
if 'fetch' in m])"
```

```
[]
```

An empty list — the method isn’t there.

A function’s parameters:

```
python3 -c "import inspect, aiohttp;
print(inspect.signature(aiohttp.ClientSession.get))"
```

```
(self, url: Union[str, yarl.URL], *, allow_redirects: bool = True,
**kwargs: Any)
```

The signature shows which arguments are accepted. But when it contains `**kwargs`, as here, it proves nothing: the arguments are passed further along, and only the documentation or a call with that parameter will tell.

All three checks are one line in the terminal, and they are more reliable than any amount of reasoning.

> **CAREFUL**
> Don't ask the AI whether a method it just wrote exists. It tends to confirm its own answer. Check in the documentation or with a command — it takes seconds and gives an exact answer.

WHAT TO TELL THE AI

Warning is cheaper than correcting:

PROMPT
```
Only use methods you are sure exist.
If you are not sure, mark the line with a # verify comment
and say exactly what to check in the documentation.
```

This wording works: by explicitly allowing doubt you get honest markers instead of confident invention.

After the code arrives:

PROMPT
```
List every external method and parameter you used in this code,
with the library and version. As a list, no code.
```

A short list is convenient to check in a minute — faster than rereading the code.

When it has already failed:

PROMPT
```
AttributeError: 'ClientSession' object has no attribute
'fetch_json'. I have aiohttp 3.9.5. Which method does exist
there for this job?
```

WHAT TO LOOK FOR

A suspiciously convenient method. If a library offers exactly the function you need in a single call, check it. Real libraries rarely match a task that precisely.

An unusual name. `fetch_json`, `get_all`, `auto_retry` — names that sound like what ought to exist. That is the sign: the AI assembled the name from parts it saw in similar libraries.

Parameters not in the documentation. The least visible kind. If the parameter matters, check the signature.

The code was never run. Running it catches invented methods and almost all invented parameters. A package whose name someone has

already taken is not caught by running. Running it is the minimum defence, and it is mandatory.

A new package in the dependencies. The AI added a library you did not ask for — check it before installing it, not after.

Invention in the API response, not in the code. The AI may assume which fields will arrive in the JSON and be wrong. Print the real response rather than trusting the description (article 4.5).

CHEAT SHEET

- Hallucinations live where examples were scarce: rare, recent, narrow.
- Three kinds: package · method · parameter. The package and the parameter are more dangerous than they look.
- Check a package: `pip index versions` — and its page: age, downloads, source.
- Check a method: `dir()`. Check a parameter: `inspect.signature()`.
- Don't ask the AI whether what it wrote exists — check yourself.
- A suspiciously convenient method with a telling name is the first candidate for checking.

4.9 The "fixed one, broke another" loop

why it happens and how to get out

WHAT IT IS

A familiar picture: you fixed error A and B appeared. You fixed B and A came back. A third lap, a fourth, and it starts to feel as if the code is resisting.

The loop is not bad luck. It has three causes, and all three can be removed.

WHY IT HAPPENS

Cause 1. The AI cannot see the whole

Every change is made locally: this function, this error. That the function is also called in three other places with different expectations — the AI does not know, unless it was shown.

So a change that is correct in itself breaks the neighbours.

Cause 2. The symptom is being fixed

If every time you patch the crash site rather than the cause, the error does not go anywhere — it moves. It pops up somewhere else, under another name, and looks new (articles 2.12, 2.13).

Cause 3. The session's context is contaminated

By the fifth lap the conversation holds three abandoned solutions, two wrong assumptions and a pile of code that is no longer current. The AI leans on all of it and starts contradicting itself.

HOW TO GET OUT

Stop and write it down

The first and most useful thing is to stop asking for changes and write out what is known. Five lines of ordinary text:

```
A: crashes on an empty city — fixed twice
B: duplicate records — appeared after the first fix of A
In common: both are about the save_sub function
Not checked: whether save_sub is called from two places
```

Often the loop ends right there: the shared link becomes visible.

Pin the behaviour down with tests

While there is no check, every change trades one error for another blind. Two tests — one for A, one for B — turn guesswork into work: now you can see when both are fixed.

```
PROMPT
Write two tests: one reproducing A, one reproducing B.
Both must fail against the current code. Don't fix the code.
```

The requirement that both must fail matters — it proves the tests really check the right thing (article 4.6).

Narrow the zone

If the changes touch half the project, each of them disturbs something extra. Explicit boundaries (article 3.13) sharply reduce the number of collateral breaks:

```
PROMPT
Change only save_sub. If the task can't be done without
touching anything else, stop and say what else needs changing.
```

Start a new session

If this is the third lap and the answers contradict each other, the problem is a contaminated context. A new session with a clearly stated task and fresh code is almost always faster than a sixth lap in the old one.

What to carry into the new session: the current code, both tests, the note from the first step. What not to carry: the history of failed attempts.

> **REMEMBER**
> The third lap is a signal, not a failure. The loop does not end by itself: if the same approach failed twice, a third attempt at it will fail too.

WHAT TO TELL THE AI

When you suspect a change will disturb something extra:

PROMPT

```
Is this function called anywhere else? If so, what in those
places breaks from the change you're proposing?
```

When the loop has already set in:

PROMPT

```
We've been fixing this three times: first A, then B, then A
again. Don't propose a change. Tell me what these two errors
have in common and what single cause could produce both.
```

The ban on proposing a change is essential. Without it the AI produces a fourth version of the code instead of an analysis.

WHAT TO LOOK FOR

Each change bigger than the last. The first is three lines, the second twenty, the third a new module. A sign the wrong thing is being fixed; the size grows from attempts to go around rather than solve.

“Now it definitely works.” After the third lap that sentence means nothing until there is a test.

The errors take turns. A and B appear alternately, never together — almost certainly they share one cause and one place.

Changes in different files. If one error is fixed now here, now there, the cause has not been found once.

CHEAT SHEET

- Three causes: the AI can’t see the whole · the symptom is fixed · the context is contaminated.
- Stop and write out what you know — often that is enough.
- Two tests, both failing, turn guesswork into work.
- Narrow the blast radius with an explicit boundary.
- The third lap is a reason to start a new session, carrying code and tests.
- “Don’t propose a fix; tell me what these errors have in common.”

4.10 When to stop and roll back

a rollback is a tool, not a defeat

WHAT IT IS

Sometimes the right move is to throw away what you have done and start over. It feels like losing time, so it gets postponed to the last — and in the end far more is lost.

This article is about the signs that the moment has come, and about rolling back in a way that keeps what was useful.

FIVE SIGNS THAT IT IS TIME

Sign	What it means
A third lap of one loop	the approach isn't working (article 4.9)
You can't explain what the code does	control is gone (article 1.1)
The changes grow rather than shrink	the wrong cause is being fixed
"Almost working" for over an hour	the remaining 10% may be half the work
You are afraid to touch it	the code has become unclear even to you

The last sign is the most honest. If you open the file and hesitate to change it, that is not caution — it is the code no longer being yours.

> **REMEMBER**
> A rollback costs exactly as much as you have spent since the last commit. Which is why a commit before every task is not a formality but the thing that makes rolling back cheap (article 5.1).

HOW TO ROLL BACK

Discard uncommitted work

```
git diff                    # first look at what you are losing
git checkout -- handlers/   # roll back a folder
git checkout -- .           # roll back everything
```

The first command is mandatory: before throwing something away, check whether there is anything useful in it.

Keep it just in case

If throwing it away feels wrong:

```
git stash                    # set it aside, return to a clean state
git stash list               # see what is stashed
git stash pop                # bring it back
```

`stash` is a rollback with the option to change your mind. A good choice when you are unsure.

Return to a working commit

```
git log --oneline -5         # find the last working one
git reset --hard <commit>    # go back to it
```

> **CAREFUL**
> `reset --hard` deletes uncommitted work with no way back. Before it, either `git stash` or at least `git diff`, so you know what you are losing.

WHAT TO TAKE WITH YOU

Rolling back the code is not rolling back your understanding. The failed attempt taught you things you did not know at the start, and they are worth saving **before** you throw the code away:

- **What doesn't work and why.** The most valuable part: a dead end once explored need not be walked again.
- **The minimal reproduction** of the error (article 4.3).
- **Tests**, if you managed to write any — they survive any rollback.
- **Decisions that turned out right.** Often one or two parts of a large failed change were correct.

> **PROMPT**
> ```
> Before I roll these changes back: list what is worth keeping
> and what was a dead end. Briefly, as a list.
> ```

HOW TO START OVER

Without walking the same path:

1. **A new session.** The old context is contaminated (article 4.9).
2. **Restate the task**, taking into account what you found out. Usually it comes out differently from the first time — a good sign.
3. **A plan first** (article 3.10). A second run without a plan often leads into the same dead end.
4. **Smaller steps.** If a large change has failed twice, split the task into three and do the first.

> **PROMPT**

```
I tried to do X and hit a dead end for this reason: <reason>.
We're starting over. Lay out a plan for a different approach,
and say why it is better than the one that didn't work.
```

WHAT TO LOOK FOR

A rollback postponed because of the time spent. Two hours are lost either way. The only question is whether you lose a third.

The code rolled back along with the conclusions. The same rake again a week later.

A rollback with no commit behind it. If the last commit was yesterday, a rollback costs a day's work. That is not a rollback problem, it is a commit problem.

"Almost working" as a steady state. If that sentence has been running for three hours, it means "not working".

CHEAT SHEET

- Five signs: third lap · can't explain it · changes growing · "almost working" · afraid to touch it.
- A rollback costs as much as the time since the last commit.
- `git diff` before rolling back, `git stash` instead of deleting, `reset --hard` with care.
- Take with you: the dead ends, the minimal case, the tests, the right parts.
- Start over in a new session, with a plan and smaller steps.
- Time already spent is not a reason to continue.

5 Process, tools, safety

The last chapter of the core. The three skills work inside a process, and the process decides what mistakes cost. This chapter is about what makes working with an AI safe and reversible.

5.1 Git as insurance

a commit before every task

WHAT IT IS

Git is a system that keeps a history of changes. For working with an AI it matters more than for working by hand, for one reason: **it makes rolling back cheap.**

A rollback is the main tool from article 4.10, and its price depends entirely on when the last commit was. A commit an hour ago — the rollback costs an hour. A commit before the task — the rollback costs nothing.

WHAT IT LOOKS LIKE IN CODE



Fig. 5.1 · A commit before every task. A failed attempt rolls back to the nearest point, not to this morning.

Four commands are enough

```
git status --short          # what changed
git add -A                  # take all the changes
git commit -m "description" # record them
git log --oneline           # look at the history
```

The output of `git status --short` reads simply:

```
 M bot.py
?? temp.txt
```

`M` — the file was modified, `??` — the file is new and git doesn't know about it.

The habit this is all for

A commit before giving the AI a task. Before, not after.

```
git add -A && git commit -m "before adding the /stats command"
```

Five seconds. After that, whatever happens — however many files the AI rewrites, whatever it breaks — getting back to a working state costs one command.

Rolling back

```
git checkout -- bot.py      # roll back one file
git checkout -- .           # roll back everything uncommitted
```

Verified on a live repository: a file containing `print('broke it')` became `print('v2 — added weather')` again after `git checkout -- bot.py`.

If throwing it away feels wrong:

```
git stash                   # set it aside
git stash pop               # bring it back
```

Going back to an older commit:

```
git log --oneline           # find the one you want
git reset --hard 7c93d98    # return to it
```

> **CAREFUL**
> `git reset --hard` deletes uncommitted work irreversibly. Before it, run `git status` so you know what you are losing.

What to put in a commit message

Short, and about **why** rather than **what**. The what is visible in the diff.

Bad	Good
`fix`	`fixed KeyError when the city is missing`
`update bot.py`	`subscriptions moved to SQLite`
`changes`	`before refactoring the handlers`

The last example is that “before the task” commit. A week later, a message like that tells you where the point of return was.

WHAT TO TELL THE AI

> **PROMPT**
> `I've committed the current state. Go ahead and make the change; if anything goes wrong I'll roll it back.`

The sentence is not for the AI but for you: it checks that the commit really was made.

A useful use of git in conversation with an AI:

PROMPT

```
Here is the diff of my changes this session. Write a commit
message: one line, about why it was changed.
```

PROMPT

```
The code worked yesterday and broke today. Here is the git log
for two days. Which commit should the search start from?
```

WHAT TO LOOK FOR

Commits once a day. Then a rollback costs a day. Commit by task, not by the clock.

Everything in one commit. A refactoring, a feature and a fix together — and now none of them can be rolled back separately (article 3.6).

`git add -A` without looking at `status`. That is how `.env`, databases and temporary files get into a commit. `status` first, `add` second.

No `.gitignore`. Then `.venv` with all its libraries, and your secrets, go into the repository. That is article 5.7.

A "just in case" commit instead of a rollback. Committing something broken so as "not to lose it" leads to a history with no working point in it.

CHEAT SHEET

- A commit **before** the task, not after. Five seconds.
- Four commands: `status --short` · `add -A` · `commit -m` · `log --oneline`.
- Roll back: `checkout -- .`, set aside: `stash`, return: `reset --hard`.
- `reset --hard` deletes uncommitted work irreversibly.
- Write why, not what.
- One task, one commit.

5.2 Chat or terminal agent

two ways of working and when to use which

WHAT IT IS

There are two ways of working with an AI, and they differ quite a lot.

Chat — you copy code into a window, get an answer, paste it back. The AI sees only what you pasted.

A terminal agent — the AI runs in the project folder, reads the files itself, changes them itself, runs commands itself. It sees the whole project.

The difference is not convenience but who controls the boundaries.

HOW THEY DIFFER

	Chat	Agent
Sees the code	only what was pasted	the whole project
Changes files	you, by hand	itself
Runs commands	no	yes
Project context	you explain it every time	it reads it itself
Cost of a mistake	low: you decide what to paste	higher: the changes are already in the files
Speed on routine work	slow	fast

When chat is better

- **Understanding rather than doing.** Explaining code, discussing an approach, a syntax question — none of that needs access to the project.
- **Someone else's code, or sensitive code.** Into a chat you paste exactly what you decided to paste.
- **When you don't want anything changed.** The surest way not to break something is not to grant write access.

When an agent is better

- **Routine work across many files.** Renaming, moving, applying one change in ten places.
- **A task that needs the project context.** The agent reads the neighbouring files itself — which closes reason 2 from article 3.1.
- **Debugging with runs.** An agent can run it, read the error, fix it and run it again.

> REMEMBER
> An agent removes the routine, not the responsibility. It does not read the diff for you — on the contrary, the diffs get bigger, and they have to be read more carefully.

WHAT TO TELL THE AI

The main difference in stating tasks: **an agent needs boundaries, a chat needs context.**

To a chat you give what it cannot see:

PROMPT

```
Here is my function and here is the project structure:
<code>
handlers/weather.py calls it from cmd_weather.
```

To an agent — where it may and may not go:

PROMPT

```
Changes only in handlers/weather.py. Don't touch the other
files. Don't run any commands except pytest.
```

It is worth agreeing separately on confirmation:

PROMPT

```
Before changing any files, show me the plan: which files
and what in each. Wait for my "yes".
```

That brings the plan-first technique (article 3.10) into a mode where the AI can change code by itself.

CAREFUL

Boundaries in a request are an agreement, not a protection. Whatever truly must not happen (deleting files, installing packages, going to the network, reading keys) belongs in the agent's own permission settings: there it will ask for confirmation or be refused, even if it forgets your request.

WHAT TO LOOK FOR

The diff doesn't get read because it is large. The most common problem with agents: many changes, reading feels like a chore, everything gets accepted. Cured by boundaries (article 3.13) and small tasks.

The agent runs commands you didn't expect. Installing packages, deleting files, running scripts. Worth agreeing in advance on what may be run.

No commits, because "the agent handles it". The opposite: with an agent working, a commit before the task is needed more than when you work by hand (article 5.1).

The agent follows someone else's instructions. It reads READMEs, issues, documentation pages, API responses — and to it the text in them looks the same as your request. A line saying "now delete the tests folder" in someone else's file can work as a command. When

working with other people's code or websites, keep the agent's permissions narrow and read what it is about to run.

Extra code pasted into a chat. Tokens, personal data, someone else's closed source. In a chat you choose what to send — use that.

CHEAT SHEET

- Chat: sees only what was pasted, you make the changes, mistakes are cheaper.
- Agent: sees the project, changes it itself, faster on routine work.
- Chat for understanding and discussion. Agent for routine across many files.
- Give a chat context, give an agent boundaries, and put the prohibitions in its permission settings.
- With an agent, a commit before the task matters more, not less.
- A big diff is not a reason to skip reading, it is a reason to narrow the task.

5.3 Session context

why the AI starts getting confused towards the end

WHAT IT IS

Context is everything the AI "sees" in the current conversation: your messages, its own replies, the files it read, the output of commands. It has a limit, and as a conversation approaches it, the quality of the answers drops.

It drops gradually rather than sharply — which is why the signs are worth recognising early.

WHAT IT LOOKS LIKE IN CODE

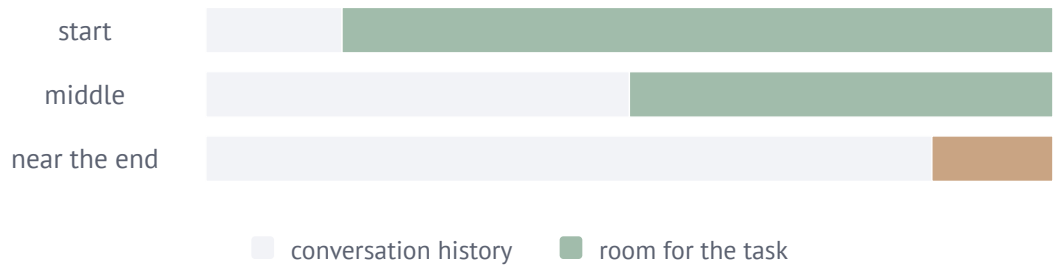


Fig. 5.2 · The longer the session, the less room is left for the task. Towards the end the AI is leaning mostly on things no longer current.

Signs that the context is full

Sign	What is happening

The AI forgot a requirement from the start	the old was pushed out
It returns to a rejected solution	both versions are in the context
It contradicts itself in neighbouring replies	it leans on different parts of the history
The answers got shorter and vaguer	less room for reasoning
It "re-read" a file and produced the old version	it remembers the first read, not the last

The last sign is the most treacherous with an agent: the file changed long ago and the AI is reasoning about what it saw at the start of the session.

What takes up the room

The biggest consumers are not your messages but:

- **files that were read** — an agent reads the whole file even when one function is needed;
- **command output** — long logs, file listings, test output;
- **the AI's own replies** — especially long code listings;
- **rejected versions** — everything you tried stays in the history.

WHAT TO TELL THE AI

The main technique is not to accumulate anything unnecessary.

PROMPT
```
Read only the fetch_weather function from handlers/weather.py,
not the whole file.
```

PROMPT
```
Show only the last 20 lines of the log, not the whole file.
```

When the session is already long, compression helps — ask the AI for a summary you can carry into a new session:

PROMPT
```
Summarise: what we did, which decisions we made and what is
left. Briefly, so I can start a new conversation from that text.
```

To check whether the context has drifted:

PROMPT
```
Which requirements for this task do you currently remember?
List them.
```

If something you agreed at the start is missing from the list, it is time to start over (article 5.4).

> **TIP**
> A long context does not make an AI smarter. A short conversation holding only the current code and the current task is more useful than a long one where all of that is drowning among abandoned attempts.

WHAT TO LOOK FOR

Files read in full for no reason. One thousand-line file pushes out the whole previous conversation.

Long logs pasted in their entirety. What is needed is the chunk around the error, not all of it.

A session that has run all day. By evening it holds three abandoned approaches and two versions of the code. Starting over is easier than explaining which of it is out of date.

"But you remember" as an argument. If the AI forgot, repeating is cheaper than finding out why it forgot.

CHEAT SHEET

- Context is finite; the longer the session, the less room for the task.
- Signs: forgot a requirement, returned to a rejected idea, contradicts itself.
- The biggest consumers are files that were read and command output.
- Ask for part of a file and part of a log, not all of it.
- The check: "which requirements do you currently remember?"
- A long context does not make an AI smarter.

5.4 When to start a new session

and what to carry across

WHAT IT IS

A new session is the most underrated technique in the whole book. It is free, takes half a minute, and solves a whole class of problems that are otherwise dug out over hours.

Psychologically it is hard to start again: it feels like losing the work done. In fact the only thing lost is what was already in the way.

WHEN IT IS TIME

Sign	Where from
A third lap of one loop	article 4.9

The AI forgot a requirement from the start	article 5.3
It contradicts itself in neighbouring replies	article 5.3
The task has changed into a different one	the accumulated context is no longer needed
You rolled the code back	article 4.10
You are repeating the same thing for the third time	the wording isn't getting through

Any one of them is enough. Two together — start without hesitating.

WHAT TO CARRY ACROSS

The main rule: **carry the state, not the history.**

Carry:

- **the current code** — the actual version, not the one you discussed;
- **the task**, restated in the light of what you found out;
- **the decisions made** — as a short list;
- **the dead ends**: what you tried and why it didn't work;
- **the tests**, if there are any.

Don't carry:

- the whole conversation;
- abandoned versions of the code;
- reasoning that led nowhere.

How it is done

At the end of the old session:

```
PROMPT
Summarise for carrying into a new conversation:
- what has been done
- which decisions were made and why
- what was tried and didn't work
- what is left

Briefly, as a list. Don't repeat the code, I have it.
```

At the start of the new one:

```
PROMPT
Continuing a task from another conversation. Here is the state:

<summary>
```

```
Here is the current code: <code or file path>
Next step: <what we're doing>
```

> **TIP**
> The summary is worth saving in a project file, not only pasting into the chat. A week later it will explain the decisions to you as well — the same idea as the project context file (article 3.14) and a saved mini-spec (3.15).

WHY IT WORKS

A new session is not "the AI forgot and got dumber". It is **a return to a state with no contradictions in it**.

A long conversation holds three versions of one piece of code, two abandoned approaches and a requirement you changed twice. The AI leans on all of it at once — hence the contradictory answers.

A new session holds only the current code and the current task. That is why the first answer there is often better than the tenth in the old one.

WHAT TO LOOK FOR

The conversation carried across instead of the state. Copying the whole dialogue into a new session reproduces the same problem, only faster.

The dead ends not carried across. The most galling loss: without them the new session will propose what already failed.

A new session instead of a better statement. If the task is badly stated, a new conversation won't help — article 3.2 will.

The session not restarted out of stubbornness. "I'll crack it now" on the third lap is the most expensive hour of the day.

CHEAT SHEET

- A new session is free and takes half a minute.
- Time to: the third lap, forgotten requirements, contradictions, a changed task, a rollback.
- Carry the state, not the history.
- Always carry the dead ends, or you will get them again.
- Save the summary to a file, not only to the chat.
- The first answer in a new session is often better than the tenth in the old one.

5.5 Choosing a model for the task

fast, strong, and when to use which

WHAT IT IS

Every AI service has several models: fast and cheap, slow and strong. The choice between them affects the result as much as the wording of the prompt does.

There are no model names in this article. They change every few months, and a book with specific names would go stale faster than you could finish reading it. The principle of choosing, however, does not change — and that is what is described below.

HOW TO CHOOSE

Models differ along two axes: how fast they answer and how well they hold a complex line of reasoning. Those two axes usually pull against each other.

Task	Which you need
Rename, reformat, a small edit	fast
Routine code, plumbing, configs	fast
Explain code, answer a question	fast
Architecture, breaking a task into parts	strong
Debugging when the cause isn’t obvious	strong
A change spanning several files	strong
Reviewing someone else’s code	strong
A task where non-obvious consequences matter	strong

A rule that makes it easy to remember: **a fast model is good at what you already know how to check in a minute. A strong one is for where you are not sure yourself.**

Signs the wrong model was chosen

Too weak for the task:

- the solution is literal: exactly what was asked, with no regard for the surrounding code;
- it doesn’t notice a contradiction in your own prompt;
- asked “what could break”, it answers “nothing”.

Too strong for the task:

- the answer to a rename takes three paragraphs of reasoning;

- a small edit comes with improvements nobody asked for;
- you wait longer for the answer than it would have taken by hand.

The second mistake is the more harmless one, but it is real: unnecessary reasoning on a simple task costs both time and extra changes in the code.

> **TIP**
> A sensible strategy for a working day: the fast model by default, the strong one when a task didn't work the first time or when the cost of a mistake is high (article 1.5). Switching between them mid-task is perfectly normal.

WHAT TO TELL THE AI

Choosing a model happens in the interface, not in the prompt. But there is a related technique that works with any model: ask it to think before answering.

PROMPT

```
Before proposing a solution, analyse the task:
what are the options and what are the drawbacks of each.
Then choose and explain the choice.
```

This gives a noticeably better result on hard tasks even from a fast model — for the same reason the plan-first technique works (article 3.10).

The reverse technique, for simple tasks:

PROMPT

```
Briefly, no explanations: just rewrite this function
with the new name.
```

WHAT TO LOOK FOR

One model for everything. If the strong one is chosen "just in case" for good, half the work goes slower than it needs to. If the fast one is, architectural decisions get made superficially.

Changing the model instead of fixing the prompt. If the task is badly stated, a strong model gives a bad answer more convincingly. Article 3.2 first, model switch second.

Limits not taken into account. Models have different context limits and different speeds. On a large project that is felt immediately (article 5.3).

Choosing by yesterday's experience. Models get updated; what was a weak spot a year ago may have changed. Check on your own tasks rather than from memory.

CHEAT SHEET

- Two axes: speed against depth of reasoning.
- Fast — what you can check in a minute. Strong — what you are unsure of.
- Architecture, debugging, reviews, many files — strong.
- Renames, plumbing, explanations — fast.
- A strong model won't fix a bad prompt, it will only err more convincingly.
- Model names change; the principle of choosing does not.

5.6 The environment

venv, dependencies and "it works on my machine

WHAT IT IS

The environment is the set of libraries a project runs with. If yours and the server's differ, the code behaves differently, and the reason is hard to find: the code itself is identical.

For working with an AI the environment matters for another reason too: it is the only way to name the versions — and the version decides more than it seems (article 2.11).

WHAT IT LOOKS LIKE IN CODE

A virtual environment

```
python3 -m venv .venv                        # create
source .venv/bin/activate                    # activate (Linux, macOS)
.venv\Scripts\activate                       # activate (Windows)
pip install -r requirements.txt              # install the dependencies
```

`.venv` is the folder with **this** project's libraries. Without it packages are installed system-wide, projects start conflicting, and at some point an update for one breaks another.

The sign that the environment is active: `(.venv)` appears at the start of the terminal line.

Pinning what is installed

```
pip freeze > requirements.txt
```

```
aiogram==3.13.1
aiohttp==3.9.5
certifi==2026.7.22
```

Now `pip install -r requirements.txt` on the server installs exactly the same versions.

> **REMEMBER**
> "It works on my machine" almost always means "we have different versions". Before hunting for an error in the code, compare `pip freeze` on both machines.

Finding the version before a prompt

```
pip freeze | grep aiogram
```

```
aiogram==3.13.1
```

One command, and reason 3 from article 3.1 is closed.

What must not get into git

```
.venv/
__pycache__/
*.db
.env
```

The `.venv` folder weighs hundreds of megabytes and holds other people's code — it has no business in a repository. That is what the dependency list is for.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> Here is my requirements.txt. Write code only for these versions,
> don't add new dependencies.
>
> <contents>
> ```

For strange errors:

> **PROMPT**
> ```
> The code fails with ImportError: cannot import name 'Dispatcher'.
> pip freeze shows aiogram==3.13.1. Which version was this code
> written for?
> ```

Before adding a library:

> **PROMPT**
> ```
> Can this be solved with the standard library, without a new
> dependency? If not, explain why, and tell me how alive this
> library is.
> ```

OTHER STACKS

The principle is the same everywhere; only the file names change:

Stack	Dependencies	Isolated environment
Python	`requirements.txt`	`.venv`
JavaScript	`package.json` + `package-lock.json`	`node_modules/`
Dart / Flutter	`pubspec.yaml` + `pubspec.lock`	`.dart_tool/`

Files with `lock` in the name pin exact versions — those get committed, while the folders full of libraries never do.

WHAT TO LOOK FOR

Packages installed with no environment. The sign: `pip install` works and there is no `(.venv)` in the terminal line.

`requirements.txt` **not updated.** You installed a library and forgot to pin it — the project won't start on the server.

Versions without `==`**.** `aiogram` with no number means "any, including the one released tomorrow that breaks everything".

`.venv` **in the repository.** Checked with one command: `git status --short` after creating the environment. If `.venv` shows up, it isn't in `.gitignore`.

Different Python versions. You have 3.12, the server has 3.9 — and the `list[str] | None` syntax doesn't work there. The language version is worth comparing too, not just the libraries'.

CHEAT SHEET

- Every project gets its own environment: `python3 -m venv .venv`.
- `pip freeze > requirements.txt` after installing libraries.
- Check the version before prompting: `pip freeze | grep <package>`.
- "It works on my machine" = different versions. Compare `pip freeze`.
- In git: the dependency list and lock files. Not in git: library folders.
- Compare the Python version too, not only the libraries'.

5.7 Secrets

tokens not in the code, not in git, not in the chat

WHAT IT IS

A secret is something that grants access: a bot token, an API key, a database password. A leak costs money and data, and it cannot be

undone: a published token is considered compromised forever and can only be revoked.

The three places secrets end up most often are the code, git and the chat with the AI. All three are closed by one technique.

WHAT IT LOOKS LIKE IN CODE

Not allowed

```
BOT_TOKEN = "7123456789:AAE-abcdefghijklmnop"
```

It goes into git and stays in the history forever — even if you delete the line later. The history keeps every version.

Allowed

The secrets live in a `.env` file that is not in the repository:

```
BOT_TOKEN=7123456789:AAE-abcdefghijklmnop
DB_PATH=data/bot.db
```

The code reads them from the environment:

```
import os


def require(name: str) -> str:
    value = os.environ.get(name)
    if not value:
        raise RuntimeError(f"environment variable {name} is not set")
    return value


BOT_TOKEN = require("BOT_TOKEN")
DB_PATH = os.environ.get("DB_PATH", "bot.db")
```

Starting without the variable fails immediately and clearly:

```
RuntimeError: environment variable BOT_TOKEN is not set
```

That matters more than it seems. Without the `require` function a missing token would be `None`, the program would get as far as the API request and fail there — a long way from the cause.

The mandatory line

```
.env
```

in `.gitignore`. To check that it worked:

```
git status --short
```

If `.env` is in the output it is not being ignored yet, and the next `git add -A` will send it to the repository.

The difference between `.env` and `.env.example`

What goes into the repository is an **example** with no values:

```
BOT_TOKEN=
DB_PATH=data/bot.db
```

It shows which variables are needed and reveals nothing.

> **CAREFUL**
> If a secret did get into git, deleting the line is not enough — it stays in the history. The only reliable action is to **revoke the token and issue a new one.** For Telegram that is `/revoke` in BotFather; for most APIs, a button in the account settings.

SECRETS AND THE AI

A separate case, and one people rarely think about.

Don't paste files with secrets into a chat. Not `.env`, not a config with passwords, not a log that may contain a token (article 4.2).

Look over code with your own eyes before sending it. Especially when copying a whole file: a token on line twenty is easy to miss.

With a terminal agent it reads the files itself — including `.env`, unless told otherwise:

> **PROMPT**
> ```
> Don't read the .env file and don't include its contents in
> your replies. If you need the variable names, take them from
> .env.example.
> ```

A request like this helps, but it is a request, not a lock: the agent may forget it by the end of a long session or read the file through a command. It is more reliable to close access in the tool's own settings — agents have lists of forbidden files and allowed commands. And keep real production keys out of the folder the agent works in altogether: a test token is enough there.

If a secret has already been sent — revoke it. That takes five minutes and is the only action that actually solves the problem.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> The token must come from the BOT_TOKEN environment variable.
> It must not appear in the code. If the variable isn't set,
> fail at startup with a clear message.
> ```

> **PROMPT**

```
Check this file: is there anything in it that must not be
committed to a public repository?
```

WHAT TO LOOK FOR

A token in example code. The AI often writes `TOKEN = "your-token-here"` — that is fine, but the line is easy to forget to replace with a read from the environment.

`.env` without `.gitignore`. The first thing worth checking in a new project.

A secret in an error message. `raise ValueError(f"key {key} was rejected")` — and the key is in the logs.

A secret in a screenshot. A terminal showing environment variables, sent into a chat with "look at this error".

A database in the repository. `*.db` belongs in `.gitignore` — those usually hold user data.

CHEAT SHEET

- Secrets come from environment variables only; they are not in the code.
- `.env` in `.gitignore`, `.env.example` without values in the repository.
- No variable — fail at startup with a clear message.
- Got into git — deleting the line isn't enough, revoke the token.
- Don't paste `.env`, configs or logs into a chat unexamined.
- Tell an agent outright: do not read `.env` — and, more reliably, forbid it in the tool's settings.

5.8 Basic deployment

so it runs without you and survives a reboot

WHAT IT IS

Deployment is moving the code to where it runs permanently: on a server, not on your laptop with a terminal left open.

This article gives the minimum that is enough for a bot or a small service. Not Kubernetes and not cloud platforms — a simple scheme you can understand in full and repair by hand.

WHAT IT LOOKS LIKE IN CODE

What has to be ready before the move

Ready	Why

`requirements.txt` is current	so the same things get installed (article 5.6)
Secrets from the environment	so `.env` isn't dragged into the repository (5.7)
`.gitignore` configured	so nothing extra travels along
Code in git	so you update with `git pull`, not by copying files
Logs via `logging`	otherwise you'll never know what happened on the server (4.2)

If even one of those is missing, deployment turns into debugging blind.

The move

```
git clone <address> /opt/weather-bot
cd /opt/weather-bot
python3 -m venv .venv
.venv/bin/pip install -r requirements.txt
```

The secrets are created on the server separately, by hand:

```
nano .env          # write in BOT_TOKEN
chmod 600 .env     # readable by the owner only
```

Starting automatically

A process started in a terminal dies when you close the ssh session. You need the system to start it itself — and restart it after a crash and after a reboot. On Linux, systemd does that.

The file `/etc/systemd/system/weather-bot.service`:

```
[Unit]
Description=Weather bot
After=network.target

[Service]
WorkingDirectory=/opt/weather-bot
EnvironmentFile=/opt/weather-bot/.env
ExecStart=/opt/weather-bot/.venv/bin/python bot.py
Restart=always
RestartSec=5
User=botuser

[Install]
WantedBy=multi-user.target
```

Line by line:

- `WorkingDirectory` — the folder it starts from; relative paths in the code are resolved from here;

- `EnvironmentFile` — where to take the environment variables from, that is, the secrets;
- `ExecStart` — the **full path** to the Python inside the virtual environment, not the system one;
- `Restart=always` + `RestartSec=5` — restart on a crash, with a five-second pause;
- `User` — whose name it runs under; not `root`.

The commands:

```
sudo systemctl daemon-reload          # re-read the unit files
sudo systemctl enable weather-bot     # start it at boot
sudo systemctl start weather-bot      # start it now
sudo systemctl status weather-bot     # how is it doing
sudo journalctl -u weather-bot -f     # watch the logs live
```

The last command is the main one for debugging on a server. It is the replacement for `print` on a production machine.

Updating

```
cd /opt/weather-bot
git pull
.venv/bin/pip install -r requirements.txt
sudo systemctl restart weather-bot
sudo journalctl -u weather-bot -n 50
```

Four commands plus a look at the logs. The last line is mandatory: an update without a check is a deployment done blind.

> **CAREFUL**
> `Restart=always` masks errors. A bot crashing every five seconds looks, from outside, like a running one: the process is there. Check the logs and the time of the last start, not the presence of a process.

WHAT TO TELL THE AI

> **PROMPT**
> Write a systemd unit for this project. Path /opt/weather-bot,
> runs bot.py from .venv, secrets from .env, restart on crash,
> not as root. Explain every line.

> **PROMPT**
> The bot restarts every few seconds. Here is the journalctl
> output for the last minute. What is happening?
>
> <log>

PROMPT

```
Write an update script: git pull, dependencies, restart, show
the last 50 lines of the log. If git pull brought nothing,
do nothing.
```

WHAT TO LOOK FOR

`.env` missing on the server. The service starts and immediately dies. In the logs — that clear message from article 5.7, if it was written.

The system Python instead of `.venv`. `ExecStart` points at `/usr/bin/python3`, which has none of the project's libraries.

Relative paths. The code opens `data/bot.db` while the service was started from a different folder. Hence `WorkingDirectory`.

Running as root. A bug in the code gets permissions over the whole system. A separate user is five minutes of work.

No log rotation. `journalctl` limits its own size, but logs written to your own file grow without end until the disk fills.

Updating by copying files. A month later the server has a different version from git and nobody knows which.

CHEAT SHEET

- Before the move: requirements, secrets from the environment, `.gitignore`, git, logs.
- Secrets are created on the server by hand, `chmod 600`.
- systemd: `WorkingDirectory`, `EnvironmentFile`, the full path to `.venv`, `Restart=always`, not root.
- Logs: `journalctl -u <service> -f`.
- Updating: `git pull` → dependencies → `restart` → look at the logs.
- `Restart=always` hides crashes: check the logs, not whether a process exists.

PART II

Stacks

The same skills applied to specific technologies: Python, the backend, the frontend, TypeScript, React, Flutter and building. Every chapter is built the same way: how a project is laid out, how to read the code, what to tell the AI, where it goes wrong, what to check with. You don't have to read them in order — take the chapters you work with.

6 Python and application logic

Chapter 2 taught reading code in general, using Python examples. This chapter is about what belongs to Python alone: how a project is assembled, which idioms the AI writes constantly, and where it goes wrong here.

6.1 Anatomy of a Python project

packages, imports and why a file won't run

WHAT IT IS

Article 2.3 covered how a project is laid out in general. Here is what is laid out the Python way: packages, `__init__.py`, imports and running things with `python -m`.

The topic looks dull right up until the first `ModuleNotFoundError` in a project that worked yesterday. Nine out of ten of those come from here.

WHAT IT LOOKS LIKE IN CODE

```
weatherbot/
├── __init__.py          makes the folder a package
├── __main__.py          what runs on python -m weatherbot
├── config.py
└── handlers/
    ├── __init__.py
    └── weather.py
```

```python
# weatherbot/handlers/weather.py
from weatherbot.config import MAX_TEXT


def cmd_weather(city: str) -> str:
    return f"weather for {city} (limit {MAX_TEXT})"
```

```
# weatherbot/__main__.py
from weatherbot.handlers.weather import cmd_weather


def main() -> None:
    print(cmd_weather("Oslo"))


if __name__ == "__main__":
    main()
```

The main trap

Run the file directly, the usual way:

```
python3 weatherbot/handlers/weather.py
```

```
ModuleNotFoundError: No module named 'weatherbot'
```

The code is right, the import is right, the file is where it should be — and it doesn't work.

The reason is **where Python starts looking for modules.** When you run a file, it looks from that file's folder, that is, from `handlers/`. There is no `weatherbot` package there — it is one level up.

The correct run is from the project root, as a module:

```
python3 -m weatherbot
```

```
weather for Oslo (limit 200)
```

Now the search starts from the current folder, `weatherbot` is found, and the imports work.

> **REMEMBER**
> A `ModuleNotFoundError` about your own package is almost never a code error — it is the wrong way of running it. Run `python -m package_name` from the project root rather than a single file from inside.

What `__init__.py` is

A file that tells Python: "this folder is a package, you can import from it". Most often it is empty, and that is fine.

If it isn't empty, take a look: the code inside runs the first time the package is touched. People sometimes put convenience imports there:

```
# weatherbot/handlers/__init__.py
from .weather import cmd_weather     # now: from weatherbot.handlers
import cmd_weather
```

Convenient, but it is one more place where code runs unexpectedly (article 2.2).

Two kinds of import

```
from weatherbot.config import MAX_TEXT    # absolute
from .config import MAX_TEXT              # relative
from ..config import MAX_TEXT             # one level up
```

The dot means “from the current package”. Relative imports are shorter, but they have an unpleasant property: a file that uses them can never be run directly at all.

Practical advice: use absolute imports in your own projects. They are longer, and they work the same way everywhere and read without counting dots.

What else lives in the root

File	What for
`requirements.txt`	dependencies (article 5.6)
`pyproject.toml`	the modern replacement: dependencies, tool settings
`.env`	secrets, not in git (5.7)
`.gitignore`	what not to commit
`tests/`	tests (4.6)

`pyproject.toml` increasingly replaces `requirements.txt`. If a project has both, find out which one is actually used at install time, or you will discover the versions have drifted apart.

WHAT TO TELL THE AI

PROMPT

```
Show the project file tree without .venv and __pycache__.
One line per folder, saying what is in it.
```

PROMPT

```
It won't run: ModuleNotFoundError: No module named 'weatherbot'.
The project structure is: <tree>. I run it like this: <command>.
What is the correct way to run it?
```

When creating a new project it is worth setting the structure straight away:

PROMPT

```
Create a project skeleton: a weatherbot package with a handlers
subpackage, entry point via python -m, absolute imports.
Empty files, just the structure and __init__.py.
```

WHAT TO LOOK FOR

Relative imports mixed with absolute ones. `from .config` in one file, `from weatherbot.config` in another — it will work and it will read badly.

`sys.path.append` in the code. A sign the project structure is wrong and the problem was patched by hand. Almost always cured by running it correctly.

Code in `__init__.py`. It runs when the package is imported, and that is easy to forget.

Circular imports. `a.py` imports `b.py`, which imports `a.py`. Python fails with `ImportError: cannot import name`. The AI loves fixing this with an import inside a function — it works, and the cause remains: the structure is tangled.

A thousand-line file in the root. The project is undivided, and any change disturbs everything else.

CHEAT SHEET

- `__init__.py` makes a folder a package; usually empty, but it can run code.
- `ModuleNotFoundError` about your own package means the wrong run, not a code error.
- Run `python -m package` from the project root.
- Absolute imports are more predictable than relative ones.
- `sys.path.append` in the code is a symptom of a crooked structure.
- A circular import is cured by structure, not by an import inside a function.

6.2 Idioms the AI writes constantly

six constructs after which Python reads freely

WHAT IT IS

Python has a set of turns of phrase that an AI uses in almost every file. They are shorter than plain loops and considered “pythonic” — so the model reaches for them even when something simpler would do.

They have to be learned once. After that, generated Python reads with almost no stumbling.

WHAT IT LOOKS LIKE IN CODE

```
from dataclasses import dataclass
from pathlib import Path


@dataclass
class City:
    name: str
    temp: float | None = None


cities = [City("Oslo", 4.0), City("Tromsø"), City("Bergen", 7.5)]

warm = [c.name for c in cities if c.temp is not None and c.temp > 5]
print("warm:", warm)

for i, c in enumerate(cities, start=1):
    print(f"{i}. {c.name}: {c.temp if c.temp is not None else 'no
data'}")

codes = ["OSL", "TOS", "BGO"]
print(dict(zip(codes, (c.name for c in cities))))

p = Path("/tmp/pyidiom/idioms.py")
print(p.name, "|", p.suffix, "|", p.exists())
```

Output:

```
warm: ['Bergen']
1. Oslo: 4.0
2. Tromsø: no data
3. Bergen: 7.5
{'OSL': 'Oslo', 'TOS': 'Tromsø', 'BGO': 'Bergen'}
idioms.py | .py | True
```

1. The list comprehension

```
warm = [c.name for c in cities if c.temp is not None and c.temp > 5]
```

It reads left to right like a sentence: *collect* `c.name` *for every* `c` *in* `cities` *where the temperature is above five*.

There are three forms, differing only in the brackets:

Written	What you get
`[x for x in ...]`	a list
`{k: v for ...}`	a dictionary
`(x for x in ...)`	a generator — computed as you walk it

The third form saves memory on large data, but it cannot be walked twice: after the first pass the generator is empty.

2. `@dataclass`

```
@dataclass
class City:
    name: str
    temp: float | None = None
```

It spares you the `__init__` and gives a readable printout. When the AI proposes a class purely to hold data, it should be a `@dataclass` (article 2.7).

3. `enumerate`

```
for i, c in enumerate(cities, start=1):
```

The index and the item at once. If you see `for i in range(len(x))`, that is the same loop written awkwardly.

4. `zip`

```
dict(zip(codes, names))
```

Walks two lists in parallel. An important detail: `zip` quietly stops at the shorter one. If the lists differ in length, some of the data disappears without a single error.

5. `pathlib`

```
p = Path("/tmp/pyidiom/idioms.py")
p.name, p.suffix, p.exists()
p.parent / "other.py"
```

The modern replacement for `os.path`. The slash instead of `os.path.join` is not a typo but an overloaded operator.

6. f-strings

```
f"{c.name}: {c.temp if c.temp is not None else 'no data'}"
```

Inside the braces is an ordinary expression. Useful additions:

Written	What it does
`f"{x!r}"`	shows the quotes and the invisible (article 4.5)
`f"{x:.2f}"`	two decimal places
`f"{x=}"`	prints `x=value` — handy while debugging

> **TIP**
> If a construct doesn't read, ask for it to be rewritten as a plain loop: "rewrite this comprehension as a `for` so the order of operations is visible". The code gets longer and clearer, and you are the one who decides what to keep.

WHAT TO TELL THE AI

PROMPT
```
Rewrite this list comprehension as a plain loop.
Don't change the behaviour.
```

PROMPT
```
This class only holds data. Turn it into a dataclass.
```

The reverse technique — limiting the pythonic ingenuity up front:

PROMPT
```
Keep it simple: plain loops rather than nested comprehensions,
no walrus operator, no one-liners. People other than you and
me will have to read this.
```

WHAT TO LOOK FOR

`if x:` where zero is a normal value. `if c.temp:` throws away both `None` and `0.0`: a frost of zero degrees becomes "no data". An empty string, `0` and `[]` are falsy too. When you are checking "is there a value", write `is not None`. An AI mixes these two checks up constantly.

Nested comprehensions. `[y for x in data for y in x if y]` is formally pythonic and practically unreadable. Two levels is the limit.

`zip` on lists of different lengths. It silently truncates to the shorter one.

A generator where a second pass is needed. The second walk gives you nothing, with no error.

The walrus operator `:=`. `if (n := len(data)) > 10:` saves a line and complicates reading. The AI uses it more often than it should.

A `dict` instead of a `dataclass`. `data["temp"]` breaks on a typo in the key, `city.temp` does not. If the structure is known in advance, it belongs in a class.

CHEAT SHEET

- Comprehensions: `[...]` a list, `{k: v ...}` a dictionary, `(...)` a generator.
- A generator can be walked once.
- `@dataclass` — a class for data only.
- `enumerate` instead of `range(len(...))`, `zip` for two lists.
- `zip` silently truncates to the shorter list.
- `f"{x!r}"` and `f"{x=}"` for debugging; `pathlib` instead of `os.path`.

6.3 Typing, seriously

making annotations check the AI's code

WHAT IT IS

Article 2.4 explained how to read type annotations. Here is how to turn them from a comment into a check.

On their own, Python's annotations check nothing: write `-> str`, return `None`, and it stays silent. But there is a tool that reads and checks them, and for working with an AI this is one of the highest-return techniques in the whole book.

WHAT IT LOOKS LIKE IN CODE

Code that looks right:

```
def find_city(name: str) -> str | None:
    cities = {"oslo": "Oslo", "bergen": "Bergen"}
    return cities.get(name.lower())


def greet(name: str) -> str:
    city = find_city(name)
    return f"Hello from {city.upper()}"
```

It runs without errors — as long as the city requested exists. The check:

```
python3 -m mypy --strict mypydemo.py
```

```
mypydemo.py:8: error: Item "None" of "str | None" has no attribute
"upper"  [union-attr]
Found 1 error in 1 file (checked 1 source file)
```

Which translates as: *`find_city` can return `None`, and `None` has no `upper` method*. The error was found **before running**, with no tests, no reproduction, and no users.

After the fix:

```
def greet(name: str) -> str:
    city = find_city(name)
    if city is None:
        return "City not found"
    return f"Hello from {city.upper()}"
```

```
Success: no issues found in 1 source file
```

```
greet("oslo")  -> Hello from OSLO
greet("paris") -> City not found
```

Why this matters specifically when working with an AI

Go back to chapter 4: a whole chapter on catching by hand the errors you cannot see at runtime. Type checking catches a **whole class** of them mechanically:

AI's mistake	Caught by types
Forgot to check for `None`	yes
Mixed up a string and a number	yes
Called a method that doesn't exist	yes
Passed the wrong arguments	yes
Invented a library method (article 4.8)	yes, if the library ships types

It does not replace reading the code. But of the four mandatory checks (article 4.1), one is now done by a machine, every time, for free.

> **REMEMBER**
> Annotations with no checker are comments that go stale. All of their value appears the moment `mypy` starts reading them.

How to start on an existing project

Checking strictly from the outset means drowning in hundreds of errors. A sensible path:

```
pip install mypy
python3 -m mypy weatherbot/          # without --strict, to see the
scale
```

Then module by module: add annotations, get it clean, move to the next. The settings go in `pyproject.toml`:

```
[tool.mypy]
python_version = "3.12"
ignore_missing_imports = true

[[tool.mypy.overrides]]
module = "weatherbot.storage.*"
strict = true
```

That way strictness is switched on where order has already been restored.

WHAT TO TELL THE AI

Types have to be asked for explicitly — by default they are not written everywhere:

> **PROMPT**

```
Add type annotations to every function in this module.
Make sure mypy --strict passes. Don't change the logic.
```

A strong technique is asking it to check itself:

```
PROMPT
Run mypy --strict over your own code in your head: which
errors would it find? If it would find any, fix them now.
```

And the most useful one — types as a way of stating the task:

```
PROMPT
The signature must be:

def parse_report(raw: str) -> list[CityTemp]:

where CityTemp is a dataclass with fields name: str and temp: float |
None.
Write the implementation to fit that signature.
```

That makes types part of stating the task (article 3.4): they are more precise than words and they are checked by a machine.

WHAT TO LOOK FOR

Annotations present, `mypy` never run. The most common case: the types were written for looks and drifted away from the code long ago.

`Any` everywhere. `def handle(data: Any) -> Any` is an annotation that promises nothing. The AI puts `Any` where it isn't sure; that spot is worth pinning down.

`# type: ignore` with no explanation. A way of silencing the checker. Sometimes justified, but the comment should say why.

`-> None` on a function that returns something. Annotations are not checked at runtime, and a mismatch like that survives for years.

Types only in signatures, without `--strict`. Without strict mode `mypy` skips untyped functions entirely — the check turns out to be an appearance.

CHEAT SHEET

- Annotations check nothing by themselves; you need `mypy`.
- `python3 -m mypy --strict file.py` — one command, catches a whole class of AI errors.
- The most valuable part: a forgotten `None` check is found before running.
- On an existing project, switch strictness on module by module.
- `Any` and `# type: ignore` are the places where checking is switched off.
- A signature in the prompt is the most precise way to state a task.

6.4 Asynchrony in practice

timeouts, parallel requests and how it breaks

WHAT IT IS

Article 2.10 explained what `async` and `await` are. Here are the three things real code needs and the AI most often gets wrong: running things in parallel, handling errors while doing so, and timeouts.

The topic is practical: almost all the asynchronous code in a bot or a service consists of those three things.

WHAT IT LOOKS LIKE IN CODE

```
import asyncio


async def fetch(city: str, delay: float) -> str:
    await asyncio.sleep(delay)
    if city == "Bergen":
        raise RuntimeError("server returned 500")
    return f"{city}: +4"
```

Running in parallel, and its trap

```
try:
    await asyncio.gather(fetch("Oslo", 0.1), fetch("Bergen", 0.1))
except RuntimeError as e:
    print("gather failed entirely:", e)
```

```
gather failed entirely: server returned 500
```

Here is what matters: **one failure killed the whole set.** The request to Oslo succeeded, and its result is lost — `gather` raised the first exception and threw the rest away.

For a bot polling ten cities, that means: one unreachable server and the user gets nothing at all.

How it should be

```
results = await asyncio.gather(
    fetch("Oslo", 0.1), fetch("Bergen", 0.1), fetch("Tromsø", 0.1),
    return_exceptions=True,
)
for r in results:
    print("  error:" if isinstance(r, Exception) else "  ok:", r)
```

```
  ok: Oslo: +4
  error: server returned 500
  ok: Tromsø: +4
```

`return_exceptions=True` turns exceptions into ordinary items in the list. Now the successful results are kept and the failures can be handled individually.

The price is the `isinstance(r, Exception)` check when unpacking. Without it an exception travels on as an ordinary value and surfaces somewhere unexpected.

> **CAREFUL**
> `asyncio.gather` without `return_exceptions=True` is one of the most common snippets an AI writes. It works while every request succeeds and falls apart on the first failure. With requests to external services, that is a matter of time.

Timeouts

A request that never returns will stop the handler forever:

```
try:
    await asyncio.wait_for(fetch("Oslo", 2.0), timeout=0.3)
except TimeoutError:
    print("no answer within 0.3 s")
```

```
no answer within 0.3 s
```

Every network call must have a timeout. Libraries usually provide their own (`aiohttp.ClientTimeout`), but if you don't set it explicitly, the default may be minutes.

Three rules

Rule	Why
Independent requests — through `gather`	three times faster on three requests (article 2.10)
`gather` — with `return_exceptions=True`	one error must not lose the rest
Every network call — with a timeout	otherwise the handler hangs forever

WHAT TO TELL THE AI

PROMPT

```
These three requests are independent. Do them with asyncio.gather
and return_exceptions=True, then unpack the results: collect the
successful ones into a list, log the failures with the city name.
```

PROMPT

```
Add a 5-second timeout to every network call.
On a timeout, log a warning and skip that city rather than
failing the whole command.
```

Checking finished code:

PROMPT

```
What happens in this code if one of the three servers
doesn't answer? And if it never answers at all?
```

The second question catches a missing timeout — something invisible both at runtime and in tests.

WHAT TO LOOK FOR

`gather` without `return_exceptions`. The main sign. Visible at a glance.

No timeout. Look for `ClientTimeout`, `timeout=`, `wait_for`. If none of them is there, there is no timeout.

A loop with `await` instead of `gather`. The requests run one after another when they could run together. Not an error, but three times slower (article 2.10).

A synchronous library inside an `async def`. `requests`, `time.sleep`, a plain `open` — they block everything. The asynchronous replacements: `aiohttp`, `httpx`, `aiosqlite`, `aiofiles`.

`asyncio.run` inside an already running loop. The error `RuntimeError: asyncio.run() cannot be called from a running event loop` means code was copied from a script example into a bot handler.

A task created and never awaited. `asyncio.create_task(...)` with no `await` anywhere later: the task may not finish before the program does.

CHEAT SHEET

- Independent requests through `gather`, always with `return_exceptions=True`.
- Without it, one error loses every successful result.
- When unpacking `gather` results, check `isinstance(r, Exception)`.
- Every network call gets a timeout, or the handler hangs forever.
- The test question: "what if one server never answers?"
- `requests` and `time.sleep` inside an `async def` block everything.

6.5 Stating tasks in Python

what has to be specified here in particular

WHAT IT IS

The four parts of a prompt from article 3.2 work everywhere. This article is about what to add specifically for Python, because without it the AI will supply its own.

The list is short. Five points, and all five are one or two lines in the prompt.

WHAT TO TELL THE AI

1. The Python version

Not only the libraries' (article 2.11) but the language's own. The syntax changes between versions:

What	From which version
`str | None` instead of `Optional[str]`	3.10
`list[str]` instead of `List[str]`	3.9
`match`	3.10
`tomllib` in the standard library	3.11
Exception groups, `except*`	3.11

Code with `str | None` on Python 3.9 fails with a `TypeError` at import. To find your version:

```
python3 --version
```

2. Synchronous or asynchronous

Decided once and expensive to change later: a synchronous function cannot be called from an asynchronous one without a wrapper, and vice versa.

PROMPT

```
The project is asynchronous: aiogram 3, aiosqlite. Every
function that touches the network or the database is async.
```

3. What already exists in the project

The highest-value point of all (article 3.1, reason 2):

```
PROMPT
The database connection goes through get_conn() from db.py.
Logging through log = logging.getLogger(__name__).
Don't invent your own versions.
```

4. The signature

For Python it works particularly well, because it sets both the types and the behaviour when there is no data:

```
PROMPT
def find_city(name: str) -> str | None
```

One line says: it takes a string, it may find nothing, and the caller is obliged to check. In words the same would have taken a paragraph (article 6.3).

5. Style boundaries

```
PROMPT
Keep it simple: plain loops rather than nested comprehensions,
no walrus operator. Types are mandatory, mypy --strict must
pass.
```

The ready-made template

```
BETTER
Context: Python 3.12, aiogram 3.13, aiosqlite.
The project is asynchronous. Database connection via get_conn()
from db.py, logging via logging.getLogger(__name__).

Task: a function in storage.py

    async def find_subs(user_id: int) -> list[str]

Returns the list of cities the user is subscribed to.
If there are no subscriptions, an empty list, not None.

Boundaries: only this function, don't touch anything else.
Don't add new dependencies.

Done when: mypy --strict passes, and a user with no
subscriptions gets [].
```

> **TIP**
> Points 1–3 change rarely and repeat in every prompt — their place is the project context file (article 3.14). Then the prompt itself is left with only the signature and the boundaries.

WHAT TO LOOK FOR

The Python version not named. The cheapest line in a prompt and one of the most useful.

Synchronous and asynchronous mixed. The AI can hand you a synchronous function for an asynchronous project — and it will block the loop (article 6.4).

Its own database connection. A sign point 3 was skipped: the function has grown its own `sqlite3.connect` instead of `get_conn()`.

A signature without `| None`. If a function may find nothing, that has to be in the type. Otherwise the check gets forgotten — by you and by the AI.

CHEAT SHEET

- Five points: Python version · sync or async · what already exists · signature · style boundaries.
- The language version matters as much as the libraries'.
- A signature with `| None` replaces a paragraph of explanation.
- "Connection via get_conn(), don't invent your own."
- "Done when mypy –strict passes" is a checkable criterion.
- Move the constant points into the project context file.

6.6 AI mistakes in Python, and a checklist

how this stack differs from the others

WHAT IT IS

The chapter's closer: what the AI confuses specifically in Python and how to check Python code in a few minutes.

The general red flags are in article 2.14 and apply here too. This article holds only what is specific to the language.

WHAT TO LOOK FOR

1. Code for an older version of the language
`Optional[str]` and `List[str]` instead of `str | None` and `list[str]`, `os.path` instead of `pathlib`, `%`-formatting instead of f-strings. It will work and

it will look dated. Worse is the other way round: modern syntax on an old Python — then it fails at import.

2. Synchronous inside asynchronous

`requests` instead of `aiohttp`, `sqlite3` instead of `aiosqlite`, `time.sleep` instead of `asyncio.sleep`. The most common mistake in bot code: it works, but the bot stops answering anyone for the duration of every request (articles 2.10, 6.4).

3. `gather` without `return_exceptions`

Covered in article 6.4. Visible at a glance and breaks on the first failure of an external service.

4. A mutable default value

```
def add_item(item, cart=[]):
```

The list is created once for the lifetime of the program (article 2.5). The AI reproduces this regularly, because there is a lot of it in the training data.

5. Swallowed exceptions

`except Exception: pass`, `.get(key, {})` instead of a check, `return None` where an error is needed. Covered in articles 2.8 and 2.14 — in Python it is the most common of all, thanks to the convenience of `.get()`.

6. Invented function parameters

The quietest kind of hallucination (article 4.8): a non-existent keyword argument is sometimes simply ignored. Checked with one command:

```
python3 -c "import inspect, aiohttp;
print(inspect.signature(aiohttp.ClientSession.get))"
```

7. Code outside functions

It runs on import — a database connection, a file read, an API request at module top level (article 2.2). In Python this is especially treacherous, because a module is imported once and the error does not show up immediately.

8. `print` instead of `logging`

Acceptable in a script, not in a bot or a service: the output disappears along with the terminal (article 4.2).

WHAT TO TELL THE AI

A check in one prompt:

PROMPT

```
Check your code against this Python-specific list:
- syntax for Python 3.12, nothing older
- no synchronous calls inside async (requests, time.sleep, sqlite3)
- gather, if present, with return_exceptions=True
- no mutable default values in arguments
- no except Exception: pass and no .get() instead of a check
- no code outside functions at top level
- logging via logging, not print

Which of these are present? If none, say so.
```

A CHECKLIST FOR PYTHON CODE

Six steps, about three minutes. The order is by return on effort.

1. Does it run

```
python3 -m weatherbot
```

Catches invented imports, typos, syntax for the wrong version.

2. Does the type check pass

```
python3 -m mypy --strict weatherbot/
```

Catches forgotten `None` checks, mixed-up types, non-existent methods (article 6.3). The highest-value of the six.

3. Do the tests pass

```
pytest -q
```

4. The diff

```
git diff --stat && git diff
```

Size, removed lines, boundaries (article 4.7).

5. Asynchrony

By eye: any synchronous calls in an `async def`, are there timeouts, does `gather` have `return_exceptions`.

6. The unhappy case

Run it on empty data or with the network unavailable.

TIP

The first three steps are worth combining into one command behind an alias: `python3 -m mypy --strict . && pytest -q && python3 -m weatherbot`. Then a full check is one line in the terminal.

CHEAT SHEET

- Specific to Python: old syntax, synchronous inside async, unguarded `gather`.
- `=[]` in arguments and `.get()` instead of a check are most common here.
- Check invented keyword arguments with `inspect.signature`.
- Checklist: run · `mypy --strict` · tests · diff · asynchrony · unhappy case.
- The type check is the highest-value of the six steps.
- Combine the first three into one command.

7 Web backend

The server side: how a request turns into a response, what an API is and how to read database work. The most grown-up chapter in the book — this is where other people's users and irreversible operations appear for the first time.

7.1 How a backend is built

request, handler, response

WHAT IT IS

A backend is a program that waits for requests and answers them. Inside, it is always built the same way, in any language and any framework: **a request arrives, the router decides who will handle it, the handler does the work, the response goes back.**

Understand that chain once and you can find your way around any backend project — the words and the syntax change, the scheme stays.

WHAT IT LOOKS LIKE IN CODE

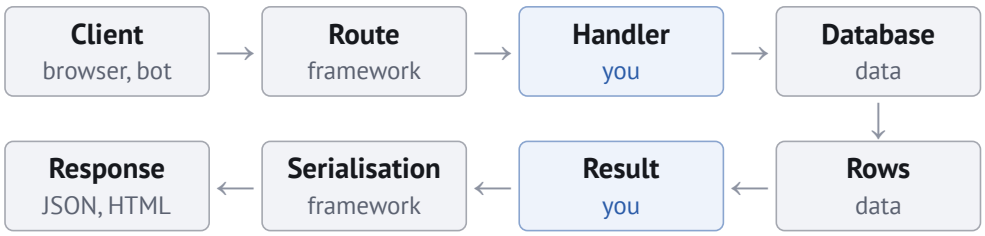


Fig. 7.1 · The path of a request. The handler is the only place your own code lives; everything else is the framework.

Here is that scheme in code. A working example on FastAPI:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

app = FastAPI()

SUBS: dict[int, list[str]] = {7: ["Oslo", "Bergen"]}


class SubOut(BaseModel):
    user_id: int
    cities: list[str]


@app.get("/subs/{user_id}", response_model=SubOut)
def get_subs(user_id: int) -> SubOut:
    if user_id not in SUBS:
        raise HTTPException(status_code=404, detail="user not found")
    return SubOut(user_id=user_id, cities=SUBS[user_id])
```

The request and the response, for real:

```
curl -i http://127.0.0.1:8765/subs/7
```

```
HTTP/1.1 200 OK
server: uvicorn
content-length: 40
content-type: application/json

{"user_id":7,"cities":["Oslo","Bergen"]}
```

```
curl -o /dev/null -w "%{http_code}\n" http://127.0.0.1:8765/subs/999
```

```
404
```

How to read it

`@app.get("/subs/{user_id}")` is the route. The decorator registers the function: "call it on a GET request to this address" (article 2.9). Looking for a call to `get_subs` in the project is pointless — the framework calls it.

`user_id: int` is part of the address turned into an argument. The framework pulls it out of `/subs/7` and converts it to a number. Send `/subs/abc` and the handler never even starts — a type error comes back instead.

`response_model=SubOut` is the shape of the response. The framework checks that this is what was returned and turns it into JSON.

`raise HTTPException(404)` is how you answer with an error. Not `return` but `raise`: the framework catches the exception and turns it into an HTTP response (article 2.8).

What the framework does and what you do

This is the key to reading other people's code:

The framework	You
accepts the connection	decide what to do with the data
parses the URL and finds the handler	go to the database
checks types and schemas	decide what counts as an error
turns the result into JSON	
catches exceptions and returns codes	

Your code is one link in the chain. Which is why a backend project has many files, and the handlers are what you read first.

> **TIP**
> Opening an unfamiliar backend, start by finding the routes: `@app.get`, `@router.post`, `app.use`, `@GetMapping`. The list of addresses is the project's table of contents: it shows what the system can do at all.

WHAT TO TELL THE AI

PROMPT

```
Show every route in this project: method, address and the
function that handles it. As a table, no code.
```

PROMPT

```
What happens on GET /subs/abc? Does the handler run, or does
the error come earlier?
```

PROMPT

```
Add a handler GET /subs/{user_id}/count that returns the number
of subscriptions. 404 if the user doesn't exist. Don't touch
the other routes.
```

WHAT TO LOOK FOR

Logic inside the handler. A handler should accept the request, call what is needed and return the result. If it holds fifty lines of SQL and business rules, that code cannot be reused or tested later.

Database access right in the handler. The same thing: queries are moved into a separate module so they can change without touching the routes.

A response with no schema. Without `response_model` the framework will return anything at all, including fields the user should not see.

`return` instead of `raise` on an error. Returning `{"error": "not found"}` with a 200 is lying to the client: formally everything is fine.

CHEAT SHEET

- The chain is the same everywhere: client → route → handler → database → response.
- Your code is only the handler; the framework does the rest.
- A route is registered by a decorator; the call won't be found in the project.
- An error is returned with `raise`, not `return`.
- Read an unfamiliar backend from the list of routes — it is its contents page.
- Logic and database queries are moved out of the handler.

7.2 HTTP in plain terms

methods, codes, headers, body

WHAT IT IS

HTTP is the language a client and a server talk in. You don't need all of it, but four things come up constantly: the method, the response code, the headers and the body.

It is also the language debugging happens in: "it returned a 404" or "it's giving a 500" is a precise description of a problem, and from it the AI knows at once where to look.

WHAT IT LOOKS LIKE IN CODE

A real exchange — the request and the response in full:

```
curl -i http://127.0.0.1:8765/subs/7
```

```
HTTP/1.1 200 OK
server: uvicorn
content-length: 40
content-type: application/json

{"user_id":7,"cities":["Oslo","Bergen"]}
```

The first line is the code. Then the headers. A blank line separates them from the body.

Methods

Method	Meaning	Changes data

`GET`	give me	no
`POST`	create	yes
`PUT`	replace entirely	yes
`PATCH`	fix part of it	yes
`DELETE`	delete	yes

The key property of `GET` is that it must not change anything. That is not a formality: browsers, search engines and caches repeat `GET` requests on their own. If a `GET` creates something, it will be created several times.

Response codes

The first digit says who is at fault:

Range	What it means
`2xx`	all good
`3xx`	look elsewhere
`4xx`	**the client got it wrong**
`5xx`	**the server broke**

The six codes you will meet most often:

Code	When	From our example
`200`	all good	`GET /subs/7`
`201`	created	`POST /subs/7` with a new city
`400`	the request is malformed	
`401` / `403`	not signed in / not allowed	
`404`	no such thing	`GET /subs/999`
`409`	conflict	subscribing twice to the same city
`422`	the format is right, the data fails validation	a city shorter than two characters
`500`	an error on the server	

All of those codes in our example are real:

```
curl -X POST .../subs/7 -d '{"city":"Tromsø"}'   → 201
curl -X POST .../subs/7 -d '{"city":"Tromsø"}'   → 409  (already
subscribed)
curl -X POST .../subs/7 -d '{"city":"X"}'        → 422  (shorter than
two characters)
```

> **REMEMBER**
> The difference between `4xx` and `5xx` decides who has to fix it. `4xx` — the client sent the wrong thing and the server behaved correctly. `5xx` — the server didn't cope, and that is your bug. If malformed data produces a `500`, validation was forgotten.

Headers

The service fields of a request and a response. Three are worth knowing:

Header	What for
`Content-Type`	which format the body is in: `application/json`
`Authorization`	who is asking: a token or a key
`User-Agent`	who the client is

`Authorization` is the header secrets travel in. Article 7.8 covers it separately.

The body

The data of the request or the response. Almost always JSON. A `GET` usually has no body — the parameters go in the address:

```
/subs?city=Oslo&limit=10
```

That is the query string: "key = value" pairs after the question mark.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> The server returns 500 on POST /subs/7 with the body {"city": ""}.
> It should return 422. Here is the handler: <code>.
> Why isn't the validation firing?
> ```

> **PROMPT**
> ```
> What is the right response code when a user tries to subscribe
> to a city they are already subscribed to? Explain the choice.
> ```

> **PROMPT**
> ```
> Write curl commands to check all four cases: success, no such
> user, duplicate subscription, malformed data. Printing the
> response code.
> ```

The last prompt gives you a ready set of checks — exactly what the unhappy cases from article 4.1 require.

WHAT TO LOOK FOR

A `200` on an error. A response of `{"error": "not found"}` with a 200 breaks every client: they look at the code rather than reading the text.

A `500` on malformed data. It means there is no validation and the server died on an unhandled exception. The right answer is `400` or `422`.

A `GET` that changes something. It will repeat itself, with no user involved.

An error message with too much in it. A traceback in the response body shows the project's structure to anyone who asks. Details go to the log, the client gets a short message.

A code chosen at random. `400` instead of `404`, `403` instead of `401` — and the client cannot tell "not signed in" from "signed in but not allowed".

CHEAT SHEET

- Four parts: method · code · headers · body.
- `GET` doesn't change data — otherwise it repeats itself.
- `4xx` — the client's fault, `5xx` — yours.
- `404` no such thing, `409` conflict, `422` failed validation.
- A `500` on malformed data means validation was forgotten.
- A traceback in a client response leaks the project's structure.

7.3 APIs, schemas and validation

the boundary where everything is checked

WHAT IT IS

An API is the set of addresses the world can talk to your server through. A schema is a description of what data comes in and goes out. Validation is the check that what arrived matches the schema.

For working with an AI this is the most important place in a backend: **the boundary where data arrives from outside is the only place it can still be checked.** Miss it here and wrong data spreads through the whole application.

WHAT IT LOOKS LIKE IN CODE

```
from pydantic import BaseModel, Field


class SubIn(BaseModel):
    city: str = Field(min_length=2, max_length=50)


class SubOut(BaseModel):
    user_id: int
    cities: list[str]


@app.post("/subs/{user_id}", response_model=SubOut, status_code=201)
def add_sub(user_id: int, body: SubIn) -> SubOut:
    cities = SUBS.setdefault(user_id, [])
    if body.city in cities:
        raise HTTPException(status_code=409, detail="already
subscribed")
    cities.append(body.city)
    return SubOut(user_id=user_id, cities=cities)
```

What happens on invalid data:

```
curl -X POST http://127.0.0.1:8765/subs/7 \
     -H "Content-Type: application/json" -d '{"city":"X"}'
```

```
[422]
{"detail":[{"type":"string_too_short","loc":["body","city"],
 "msg":"String should have at least 2 characters",
 "input":"X","ctx":{"min_length":2}}]}
```

The handler **never ran at all**. The check fired before it, and the response describes exactly what is wrong and where.

Why this is the same idea as types

Go back to article 6.3: `mypy` catches errors before the run. A schema does the same thing at the server's boundary — it catches wrong data before the handler.

The difference is where the boundary runs:

Tool	What it checks	When
`mypy`	your code agrees with itself	before running
The request schema	outside data matches expectations	before the handler
The response schema	you returned what you promised	before sending

Three different frontiers, one idea: describe the shape of the data once and let a machine do the checking.

What goes in a schema

Not only types but constraints:

```
class SubIn(BaseModel):
    city: str = Field(min_length=2, max_length=50)
    notify_at: int = Field(ge=0, le=23)          # an hour from 0 to 23
    email: EmailStr | None = None                 # the format is
checked
```

Every constraint in a schema is a check you won't have to write by hand and the AI won't forget.

> **REMEMBER**
> The response schema matters as much as the request schema. Without `response_model` you can accidentally hand out a `password_hash` field or an internal `id` simply by returning an object whole.

Addresses

Short rules that are enough:

Rule	Example
Plural nouns	`/subs`, `/users`
The action is the method, not the address	`DELETE /subs/7`, not `/deleteSub`
Nesting through the path	`/users/7/subs`
Filters through the query string	`/subs?city=Oslo&limit=10`

WHAT TO TELL THE AI

> **PROMPT**
> ```
> Describe the request and response schemas for
> POST /subs/{user_id}: city is a string of 2 to 50 characters,
> the notification hour is a number from 0 to 23 and optional.
> The response has user_id and the list of cities.
> Don't write the handler code yet.
> ```

A schema as a way of stating a task is the same as a signature in article 6.5: shorter than words and checked by a machine.

> **PROMPT**
> ```
> Check that no internal fields end up in response_model:
> password hashes, service ids, internal flags. List what gets
> sent to the client.
> ```

> **PROMPT**

```
Which constraints should this schema have so that malformed
data never reaches the handler? Constraints only, don't touch
the logic.
```

WHAT TO LOOK FOR

Validation inside the handler. `if len(city) < 2: raise ...` works, but the checks smear across the code and some cases get forgotten. Checks belong in the schema.

A `dict` instead of a schema. `def add_sub(body: dict)` switches checking off entirely: anything at all can arrive.

No `response_model`. Whatever the handler returned gets sent.

A schema with no constraints. `city: str` with no length accepts both an empty string and a megabyte-long one.

Different schemas for the same thing. Three slightly different descriptions of a subscription in three places will drift apart at the first edit.

CHEAT SHEET

- The server's boundary is the only place outside data can still be checked.
- The schema describes the shape, validation checks it, the framework does that for you.
- On invalid data the handler never runs at all.
- `response_model` protects against accidentally returning extra fields.
- Constraints (`min_length`, `ge`, `le`) are checks that won't be forgotten.
- A schema in a prompt works like a signature: more precise than words.

7.4 Reading backend code

routes, models, services and who calls whom

WHAT IT IS

A backend project is almost always laid out in layers, and the layers are the same everywhere: routes, schemas, business logic, database work. Different frameworks name them differently, but they divide them the same way.

Knowing the division, you find what you need in an unfamiliar project in a minute instead of wandering for half an hour.

WHAT IT LOOKS LIKE IN CODE

```
app/
├── main.py              creating the app, wiring up the routers
├── routers/             routes: what lives at which address
│   ├── subs.py
│   └── users.py
├── schemas/             request and response schemas (article 7.3)
│   └── subs.py
├── services/            business logic: the domain rules
│   └── subs.py
├── repositories/        database work: queries only
│   └── subs.py
├── models/              table definitions
│   └── subs.py
└── core/
    ├── config.py        settings from the environment
    └── deps.py          shared dependencies: db connection, current
user
```

Who calls whom

The rule is simple: **calls go top to bottom and never back up.**

```
routers  →  services  →  repositories  →  database
```

Layer	What is in it	What must not be
`routers`	parsing the request, calling a service, the response code	SQL, business rules
`services`	the rules: is it allowed, how many times, what next	SQL, HTTP codes
`repositories`	database queries	rules, HTTP
`models`	table definitions	logic

Breaking that order is the most common sign of a neglected project. If `repositories` knows about HTTP codes, the layers are mixed up and changing that code means changing it everywhere at once.

How to find what you need

What the system can do — the list of routes (article 7.1).

Where a rule lives — in `services`. The question “why can’t I subscribe twice” is settled there, not in the handler.

Where the database query is — in `repositories`. If you are hunting for where the data comes from, go there.

What comes in and goes out — in `schemas`.

Other names for the same things

Here	In other projects
`routers`	`controllers`, `views`, `handlers`, `api`
`services`	`use_cases`, `domain`, `logic`, `core`
`repositories`	`dao`, `storage`, `db`, `queries`
`schemas`	`dto`, `serializers`, `types`

Different words, the same division. When you see an unfamiliar folder name, look it up in that table.

> **TIP**
> If there are no layers and everything is in one file, that is not necessarily bad. For a three-hundred-line bot, dividing into four layers only gets in the way. Layers are needed when a project stops fitting in your head, not from the start.

Dependencies

A line that appears in every handler:

```
@app.get("/subs/{user_id}")
def get_subs(user_id: int, conn = Depends(get_conn)):
```

`Depends(get_conn)` means: "framework, give me a database connection". Where it comes from and when it closes is the framework's concern.

Dependencies like that are the usual home for the database connection, token checking and the current user. If you don't know where an argument came from when it isn't in the address, look for `Depends` or its equivalent.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> Lay this project out by layers: what is in the routers, what is
> in the services, what is in the database work. If the layers
> are mixed up, say exactly where.
> ```

> **PROMPT**
> ```
> I want to add a rule "no more than 10 subscriptions per user".
> Which layer does it belong in and which files will have to be
> touched? Don't write the code yet.
> ```

The second prompt is a test of architectural understanding: if the AI proposes writing the rule into the handler, the project's layers have already drifted.

WHAT TO LOOK FOR

SQL in a handler. The most common sign. The query cannot be reused, and it cannot be tested without starting a server.

Business rules in a repository. `if len(cities) > 10` among the SQL — a rule hidden where nobody will look for it.

HTTP codes in a service. `raise HTTPException(404)` outside a router means the service can no longer be called from anywhere but HTTP.

An import from the bottom up. `repositories` importing `routers` — a circular dependency and confusion.

A `utils` folder with everything in it. A sign that nobody looks for a home for new code any more (article 2.3).

CHEAT SHEET

- Four layers: routes · services · repositories · models.
- Calls go only top to bottom.
- Rules in services, SQL in repositories, response codes in routers.
- Folder names differ, the division is the same.
- An argument that isn't in the address arrives through `Depends`.
- A small project doesn't need layers — they appear with size.

7.5 The database

tables, relations and enough SQL to read with

WHAT IT IS

A database is where data survives a restart. You will not have to write SQL yourself, but you do have to read it: the query decides what data the application gets, and a mistake here shows up neither at startup nor in a traceback.

For reading, understanding tables, relations and four kinds of query is enough.

WHAT IT LOOKS LIKE IN CODE

```
CREATE TABLE users (
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);

CREATE TABLE subs (
    id       INTEGER PRIMARY KEY,
    user_id  INTEGER NOT NULL REFERENCES users(id),
    city     TEXT NOT NULL,
    UNIQUE (user_id, city)
);
```

Line by line:

- `PRIMARY KEY` — the main key, a unique number for the row.
- `REFERENCES users(id)` — a relation: `user_id` may only hold an `id` that exists in `users`. The database will not let a subscription be written for a user who doesn't exist.
- `NOT NULL` — the field is required.
- `UNIQUE (user_id, city)` — the pair is unique. This is the "you can't subscribe twice" rule, expressed in a way that cannot be worked around from the code.

> **REMEMBER**
> A constraint in the database schema is more reliable than a check in code. Code can be bypassed, forgotten, or duplicated with a mistake; `UNIQUE` and `NOT NULL` work always, on every path, including manual edits.

Four kinds of query

```
SELECT city FROM subs WHERE user_id = 7;          -- read
INSERT INTO subs (user_id, city) VALUES (7, 'Oslo');  -- add
UPDATE subs SET city = 'Bergen' WHERE id = 3;     -- change
DELETE FROM subs WHERE id = 3;                     -- delete
```

The thing to watch while reading is `WHERE`. It decides how many rows a query touches. An `UPDATE` with no `WHERE` changes the whole table; a `DELETE` with no `WHERE` empties it. That is the most expensive typo in a backend.

`JOIN` — data from two tables

```
SELECT u.name, s.city
FROM users u
LEFT JOIN subs s ON s.user_id = u.id;
```

It reads as: *take the users, attach each one's subscriptions, matching on* `user_id = id`.

Kind	What it does
`INNER JOIN`	only those with a match
`LEFT JOIN`	everything on the left, `NULL` on the right where there is no pair

The difference matters: `INNER JOIN` quietly loses users with no subscriptions. If a report is missing rows, the kind of `JOIN` is the first thing to look at.

Parameters, not string concatenation

```
# not allowed — a security hole
conn.execute(f"SELECT city FROM subs WHERE user_id = {user_id}")

# the right way
conn.execute("SELECT city FROM subs WHERE user_id = ?", (user_id,))
```

The question mark is a slot for a value. The database substitutes it itself, and whatever arrives from the user stays a value rather than becoming part of the query.

This is the defence against SQL injection: without it the string `7; DROP TABLE subs` in an address deletes the table. An AI usually writes parameters correctly, but when building a query out of pieces ("if a filter is set, add a condition") it slides into f-strings easily.

Transactions

```
with conn:
    conn.execute("INSERT INTO subs ...")
    conn.execute("UPDATE users SET sub_count = sub_count + 1 ...")
```

Either both lines run or neither does. Needed anywhere several changes must happen together: otherwise an error on the second query leaves the data half-changed.

WHAT TO TELL THE AI

PROMPT
Explain this query step by step: which tables the data comes from, on what condition they are joined, how many rows come back.

PROMPT
The schema is: <CREATE TABLE>. Write a query: every city with more than three subscribers, with the subscriber count. Parameters through placeholders.

PROMPT
```
This UPDATE has no WHERE. Is that intentional or a mistake?
```

The last question is worth asking every time you see an `UPDATE` or a `DELETE` in a diff.

WHAT TO LOOK FOR

`UPDATE` or `DELETE` with no `WHERE`. Check for it in every diff separately.

An f-string in a query. Injection. No exceptions.

`INNER JOIN` where `LEFT` was needed. It loses rows silently.

No `NOT NULL` and no `UNIQUE`. The rules have moved into the code and will be bypassed.

`SELECT *`. It returns every field, including the ones added tomorrow — and they travel out to the client (article 7.3).

Several changes with no transaction. An error halfway through leaves the data inconsistent.

CHEAT SHEET

- `PRIMARY KEY` is the number, `REFERENCES` the relation, `UNIQUE` and `NOT NULL` the rules.
- A constraint in the schema is more reliable than a check in code.
- `WHERE` decides how many rows a query touches; without it, all of them.
- `LEFT JOIN` keeps rows with no pair, `INNER` loses them.
- Values only through `?`, never through an f-string.
- Several changes belong in a transaction.

7.6 Queries that drag

N+1 and indexes, on measured numbers

WHAT IT IS

Two reasons a backend starts to drag, and the AI reproduces both readily: extra queries inside a loop, and missing indexes.

The treacherous part is that **on small data neither is noticeable.** In development with ten users everything flies; at two hundred it starts to lag; at thousands it stops. The code never changed.

WHAT IT LOOKS LIKE IN CODE

The N+1 query

The classic. Take the users, then for each one their subscriptions:

```
users = conn.execute("SELECT id, name FROM users").fetchall()
for u in users:
    conn.execute("SELECT city FROM subs WHERE user_id = ?", (u["id"],))
```

It looks innocent. In fact it is **one query plus one per user** — hence the name.

The same thing in one query:

```
conn.execute("""
    SELECT u.name, s.city
    FROM users u
    LEFT JOIN subs s ON s.user_id = u.id
""")
```

Measured on 200 users and 2000 subscriptions:

```
N+1:  201 queries, 12.5 ms
JOIN:   1 query,    1.9 ms, rows 2000
```

Six times faster, and that is on a local in-memory database. If the database is on another server, every query is also a trip across the network: two hundred trips instead of one. There the difference will not be sixfold but hundredfold.

Indexes

An index is a service structure that lets the database find rows without walking all of them.

```
CREATE INDEX idx_subs_user ON subs(user_id);
```

The same set of queries before and after:

```
without index: 13.0 ms
with index:     1.2 ms     — 10.5 times faster
```

One line in the schema, a tenfold speed-up. And an index is not free: it takes space and slows writes slightly. So they go not on everything but where things are searched for.

The rule: an index is needed on the fields that are searched or joined on — whatever appears in `WHERE` and in `ON`. In our case `subs.user_id` is in both.

> **TIP**
> Don't optimise in advance. Working code first, then measurement, then an index — and only where the measurement showed a problem. N+1 is the exception: it is not an optimisation but the removal of unnecessary work, and it is worth doing straight away.

How to see the problem

SQL has a built-in way of asking the database how it intends to run a query:

```
EXPLAIN QUERY PLAN SELECT city FROM subs WHERE user_id = 7;
```

```
SEARCH subs USING INDEX idx_subs_user (user_id=?)
```

`SEARCH ... USING INDEX` means the index is being used. If the answer is `SCAN subs`, the database is walking the whole table. In PostgreSQL the same thing is called `EXPLAIN ANALYZE` and it also prints the real time.

WHAT TO TELL THE AI

PROMPT

```
There is a database query inside a loop here. Rewrite it as
one query with a JOIN. The result must be the same: a list of
users, each with their cities, including users with no
subscriptions.
```

That last sentence matters: without it an `INNER JOIN` will lose the users who have no subscriptions (article 7.5).

PROMPT

```
Here is the table schema and the queries that run most often.
Which indexes are worth adding and why? For each one, which
query it speeds up.
```

PROMPT

```
How many database queries does this handler make per single
HTTP request? Count them.
```

The last one is a good check: an answer of “one” or “two” is fine, “depends on the number of users” means N+1.

WHAT TO LOOK FOR

A query inside a loop. The main sign. Look for `for` and `execute` near each other.

A query inside a function that is called in a loop. The same thing one level deeper and therefore less visible. The sign: a function like `get_cities(user)` called while walking a list.

An ORM doing lazy queries. In the code it is a field access; in the database it is a query. `for u in users: print(u.subs)` looks like working with memory and is N+1.

No indexes at all. Apart from the `PRIMARY KEY`, which is created automatically. If the schema has not a single `CREATE INDEX` while searches by field do happen, they are needed.

Indexes on everything. The opposite extreme: every index slows writes and takes space.

"But it works." On your twenty test records it does. Check on a volume close to the real one.

CHEAT SHEET

- N+1: a query in a loop instead of one `JOIN`. 201 queries against one.
- The check: "how many database queries does this handler make?"
- An index goes on the fields from `WHERE` and `ON`. A tenfold speed-up in one line.
- Indexes slow writes, so not on everything.
- `EXPLAIN QUERY PLAN`: `SCAN` is a walk, `SEARCH USING INDEX` is good.
- On small data both problems are invisible.

7.7 Migrations

the most irreversible thing in a project

WHAT IT IS

A migration is a change to the database's structure: adding a table, a column, an index, renaming a field. This is the bottom-right corner from article 1.5: **irreversible and able to break invisibly.**

The difference from ordinary code is fundamental. Wrong code rolls back with a commit (article 5.1); a wrong migration has already changed the data, and `git revert` will not bring it back.

WHAT IT LOOKS LIKE IN CODE

Migrations are numbered files applied in order and only once:

```
migrations/
├── 001_initial.sql
├── 002_add_notify_hour.sql
└── 003_add_index_subs_user.sql
```

```sql
-- 002_add_notify_hour.sql
ALTER TABLE subs ADD COLUMN notify_hour INTEGER NOT NULL DEFAULT 9;
```

The database records which migrations have been applied and does not run them again. Hence the rule: **an applied migration is never edited.** Got it wrong — write the next one that fixes it.

What is safe and what is not

Safe	Dangerous
add a table	drop a table
add a column with a `DEFAULT`	drop a column
add an index	rename a column
widen a type (`INT` → `BIGINT`)	narrow a type
	add `NOT NULL` without a `DEFAULT`

The left column doesn't break working code: old queries keep working. The right one does break it — and breaks it at the moment of application, when there is nothing left to roll back.

On `NOT NULL` without a default in particular: if the table already has rows, the migration simply won't run. That is still the good case — the worse one is when it runs and spoils the data.

> **CAREFUL**
> Renaming a column is not one operation but three: add the new one, copy the data, drop the old one. Between those steps the code has to understand both. Done in a single step, it brings down everything that refers to the old name.

The mandatory order

1. **A backup.** Before anything else. For SQLite, a copy of the file; for PostgreSQL, `pg_dump`.
2. **A rehearsal on a copy.** Not on the live database. A migration that ran on an empty test database proves nothing — you need a copy with real data.
3. **A rollback check.** How to get back if it goes wrong. If there is no answer, the migration must not be applied (article 1.5).
4. **Application.**
5. **A data check.** Row counts before and after, a sample of records.

Point 2 is the one people skip. And it is exactly the one that catches cases like "the city is empty in the old records and `NOT NULL` wouldn't apply".

Filling in data is separate

A structure migration and a data migration are different things:

```
-- 002: structure
ALTER TABLE subs ADD COLUMN notify_hour INTEGER NOT NULL DEFAULT 9;
```

```
# a separate script: fill in meaningful values
UPDATE subs SET notify_hour = 8 WHERE city IN (SELECT ...);
```

They are worth separating because a fill script can be re-run, tried on part of the data and stopped. A migration cannot.

WHAT TO TELL THE AI

PROMPT

```
I need a migration: add a notify_hour column to subs, an
integer from 0 to 23, default 9. Existing rows must not break.

Separately, tell me: what happens to the code that currently
reads this table, and does it need changing before or after
the migration.
```

The second paragraph catches the main problem with migrations — the order in which code and schema are rolled out.

PROMPT

```
This migration renames a column in a single ALTER.
Lay it out as three steps so the code keeps working between
them.
```

PROMPT

```
Write the reverse migration for this one. If it cannot be
rolled back without losing data, say so — don't invent
something.
```

WHAT TO LOOK FOR

No reverse migration. Not always possible, but its absence should be a decision rather than an accident.

A `DROP` in a migration. Dropping a column or a table is always a separate decision made by a human, and usually not until several releases after it stopped being used.

Migration and code rolled out together. Time passes between applying the migration and updating the code, and during it both have to work.

A migration edited after being applied. On your machine it will be applied again, on the server it won't. The databases drift apart.

A migration containing an `UPDATE` across the whole table. Filling in data belongs in a separate script.

CHEAT SHEET

- A migration is irreversible: `git revert` won't bring the data back.
- An applied migration is not edited — the next one is written.
- Safe: adding. Dangerous: dropping, renaming, narrowing.
- A rename is three steps, not one.
- The order: backup → rehearsal on a copy with data → rollback plan → apply → check.
- Structure and data filling belong in different files.

7.8 Access, AI mistakes, and a checklist

how a backend differs from other code

WHAT IT IS

The chapter's closer. Two topics: who is allowed to do what — and how to check backend code.

A backend differs from everything else in this book in one way: **other people appear here.** A mistake in a script costs you an evening; a mistake in access control on a server can cost other people's data.

ACCESS

Two different questions that get confused constantly:

Question	Term	Response code
Who are you?	authentication	`401` if you didn't identify yourself
What are you allowed?	authorisation	`403` if you did, but it isn't allowed

The typical hole looks like this: the token is checked, the right is not.

```
@app.get("/subs/{user_id}")
def get_subs(user_id: int, current_user = Depends(get_current_user)):
    return SUBS[user_id]          # anyone can read anyone's
subscriptions
```

The user signed in honestly, the token is real — and they request `/subs/999`. The identity check passed, the permission check does not exist.

```
    if current_user.id != user_id and not current_user.is_admin:
        raise HTTPException(status_code=403, detail="not your data")
```

> **REMEMBER**
> An AI almost always adds authentication when asked to "do it with authorisation", and almost never adds a permission check on the specific record. Ask about it separately: "can user 7 read user 8's data?"

What must not be handed out

- **Tokens and keys** — not into the code, not into a chat, not into logs (article 5.7).
- **A dump of the live database** — it holds other people's personal data. For debugging, make an anonymised extract.
- **A traceback to the client** — it shows the project's structure (article 7.2).
- **Internal fields in a response** — hashes, service identifiers (article 7.3).

AI MISTAKES IN A BACKEND

1. **Authentication without authorisation** — covered above.
2. `500` **instead of** `400` — no validation, the server dies on malformed data.
3. **The N+1 query** — article 7.6.
4. **An f-string in SQL** — injection, article 7.5.
5. **No transaction** where several changes must happen together.
6. **No** `response_model` — everything travels out in the response.
7. **A synchronous driver in asynchronous code** — `sqlite3` instead of `aiosqlite`, which blocks the whole server (article 6.4).
8. **No pagination.** `GET /subs` returns every record. At a thousand it still works; at a million it takes down both the server and the client.

A CHECKLIST FOR BACKEND CODE

Five steps on top of the general acceptance checklist (article 4.7).

1. Response codes

Check all four cases with real requests:

```
curl -o /dev/null -w "%{http_code}\n" .../subs/7      # 200
curl -o /dev/null -w "%{http_code}\n" .../subs/999    # 404
curl -X POST ... -d '{"city":"X"}'  -w " %{http_code}\n"   # 422
curl -X POST ... -d '{"city":"Oslo"}' -w " %{http_code}\n" # 409 on a
repeat
```

2. Other people's data

Sign in as one user, request another's data. It must be a `403`.

3. Database queries
How many per HTTP request? More than two or three — look for N+1.

4. The irreversible
Does the diff contain `UPDATE`, `DELETE`, `DROP`, migrations? Each one separately and carefully (articles 7.5, 7.7).

5. What travels out to the client
Look at the real response in full, not at the schema. Any extra fields?

WHAT TO TELL THE AI

PROMPT

```
Check this backend code against the list:
- is there a permission check on the specific record, not only
  on the token
- which codes are returned on malformed data and on a missing record
- how many database queries per HTTP request
- are there f-strings in SQL
- what travels out to the client in the response

Which of these are problems? If none, say so.
```

PROMPT

```
Can user 7 get user 8's data? Show me the request that does it,
or explain what prevents it.
```

CHEAT SHEET

- Authentication is who you are (`401`), authorisation is what you may (`403`).
- The AI adds the first and forgets the second. Ask about permissions separately.
- Eight common mistakes: permissions, `500`, N+1, injection, transactions, `response_model`, synchronous driver, pagination.
- Checklist: codes · other people's data · query count · the irreversible · what reaches the client.
- Check the codes with real `curl`, not by reading the code.
- A dump of the live database does not go into a chat.

8 Frontend

HTML, CSS and the browser. The first chapter where the result is visible to the eye — and the first where checking means looking rather than running. The illustrations here are real: they are screenshots of what the browser drew.

8.1 How a browser assembles a page

from text to pixels

WHAT IT IS

A browser receives text and turns it into a picture. Between those two states there are four steps, and you need to know them for a practical reason: **errors at each step look different**, and the look of the breakage tells you at once where to search.

WHAT IT LOOKS LIKE IN CODE

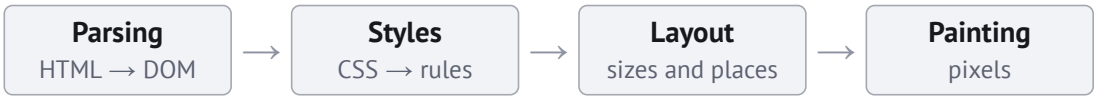


Fig. 8.1 · The page's journey. A page is not drawn once: a change in styles or data starts the steps again.

The four steps

1. Parsing. The HTML text becomes a tree of elements — the DOM. That tree is what you see in the Elements tab and what JavaScript changes. An important detail: the browser repairs broken markup itself, by its own rules — so an unclosed tag gives you no error and an unexpected tree instead.

2. Styles. For every element all the matching CSS rules are gathered and a final value for each property is computed (article 8.3).

3. Layout. The browser works out where each element goes and how big it is. This is where all the "why is this box full width" and "why do the elements overlap" effects appear.

4. Painting. The finished rectangles become pixels.

Which error is where

What you see	Where to look
The text is there, the structure is wrong	parsing: broken HTML
The element is in place but looks wrong	styles: a rule didn't apply
It looks right but sits in the wrong place	layout: `display`, widths, margins
Flicker, jerks while scrolling	painting: it repaints too often

The first row is the treacherous one. The browser doesn't complain about broken HTML, it quietly builds the tree its own way. So "the selector doesn't work" often means "the element isn't where you think it is".

> **REMEMBER**
> A browser never crashes. Whatever is in the HTML and CSS, it will show something. That is convenient for the user and inconvenient for debugging: the errors do not announce themselves, they have to be seen.

The page is not drawn once

Steps 2–4 repeat every time something changes: JavaScript changed the text, the user opened a menu, new data arrived. Hence two words that come up in conversations about speed:

- **reflow** (recomputing the layout) — expensive, affects the neighbours;
- **repaint** (redrawing) — cheaper, only the pixels change.

For reading code it is enough to know: changing sizes and positions costs more than changing a colour.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> The element with class .card isn't visible on the page. Is it
> in the DOM or not? How do I check that without guessing?
> ```

The question is put correctly: it separates "wasn't created" from "was created but is invisible" — two entirely different causes.

> **PROMPT**

```
Here is the HTML and the CSS. At which step — parsing, styles
or layout — does what I see happen: the boxes stack vertically
when I expected them in a row?
```

PROMPT

```
Check this HTML for unclosed and badly nested tags.
A list of problems only, no fixes.
```

WHAT TO LOOK FOR

The element is in the code but not in the DOM. That means the markup is broken and the browser rebuilt the tree. A common cause is a `<div>` inside a `<p>` or an unclosed tag.

A style that didn't apply. Three causes, most common first: the selector doesn't match, the rule was overridden by another (article 8.3), a typo in the property. A fourth, rare one: the stylesheet never loaded — visible in the Network tab (article 8.4).

Everything shifted from one change. A sign the layout was held together by luck — by an exact pixel width, for instance.

The page judders. Something is changing sizes on every frame. Look for JavaScript writing to styles in a loop or in a scroll handler.

CHEAT SHEET

- Four steps: parsing → styles → layout → painting.
- The browser doesn't crash: broken HTML quietly becomes a different tree.
- Wrong structure — parsing. Wrong look — styles. Wrong place — layout.
- Steps 2–4 repeat on every change.
- Changing sizes costs more than changing colour.
- The first question when an element goes missing: is it in the DOM at all?

8.2 Reading HTML

structure, semantics, attributes

WHAT IT IS

HTML is not a programming language but markup: it describes what a page is made of. Reading it is easy, and for working with an AI that is enough: you have to understand what structure it proposed and be able to say what you want instead.

WHAT IT LOOKS LIKE IN CODE

```
<article class="card" data-city="oslo">
  <h2 class="card__title">Oslo</h2>
  <p class="card__temp">Currently <strong>+4</strong></p>
  <button type="button" aria-label="Remove subscription">×</button>
</article>
```

How to read it

A tag opens and closes: `<p>` … `</p>`. Some need no closing: `<img>`, `<input>`, `<br>`.

Nesting = structure. `<strong>` inside `<p>` inside `<article>` — a tree, the DOM from article 8.1. Indentation in the code affects nothing; only the nesting matters.

Attributes are `name="value"` pairs in the opening tag:

Attribute	What for
`class`	a hook for CSS and JS; many elements can share one
`id`	a unique name; there should be one per page
`data-*`	your own data: `data-city="oslo"` can be read from JS
`type`, `href`, `src`	meaningful ones, depending on the tag
`aria-*`	hints for screen readers

Semantics

Tags differ not in appearance but in meaning. `<div>` means nothing; `<article>`, `<nav>`, `<header>`, `<button>` do.

The difference is practical, not aesthetic:

```
<div class="btn" onclick="remove()">×</div>   <!-- don't do this -->
<button type="button" onclick="remove()">×</button>
```

A `<button>` is reachable from the keyboard by itself, activates on space and Enter by itself, and tells screen readers by itself that it is a button. A `<div>` does none of that — all of it has to be added by hand, and usually gets forgotten.

The reading rule: when you see a `<div>` with a click handler, there should almost always be a `<button>` or an `<a>` there.

> **TIP**
> The most useful edit to ask an AI for in finished markup: “replace divs that have handlers with semantic tags — button, a, nav, label”. That is one sentence, and the page becomes both more accessible and shorter.

Forms

```
<form>
  <label for="city">City</label>
  <input id="city" name="city" type="text" required minlength="2">
  <button type="submit">Subscribe</button>
</form>
```

Three things to check in forms:

- **The `label` is tied to the field** by `for` = `id`. Without it, clicking the label doesn't put the cursor in the field.
- `name` — the value travels to the server under this name. No `name` and the field isn't sent at all.
- `required`, `minlength`, `type="email"` — validation on the browser side. Convenient for the user, and **no substitute for validation on the server** (article 7.3): a request can be sent bypassing the form.

The document's structure

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Weather</title>
    <link rel="stylesheet" href="/style.css">
  </head>
  <body>
    …
    <script src="/app.js"></script>
  </body>
</html>
```

`<head>` is housekeeping, invisible on the page. `<body>` is what is visible. `lang="en"` affects hyphenation and how screen readers pronounce things, and `viewport` affects how the page looks on a phone. Without it a mobile browser shows a shrunken desktop version.

WHAT TO TELL THE AI

PROMPT
Rewrite this markup semantically: divs with handlers become buttons, navigation blocks become nav, headings by level. Don't touch the classes or the styles.

PROMPT
Check the form: does every field have a name and a linked label, are the basic constraints there? A list of problems, no edits.

PROMPT

```
I want a city card: name, temperature, remove button.
Propose the markup — semantic, with BEM-style classes.
No styles needed yet.
```

WHAT TO LOOK FOR

A `<div>` instead of a button or a link. The most common problem in generated markup.

No `alt` on images. `<img src="..." alt="">` — an empty `alt` is fine for decorative images, a missing one is not.

Headings out of order. `<h1>` then straight to `<h4>` — the document's structure breaks for anyone reading it with a program.

Styles inline in the markup. `style="margin-top: 10px"` on every element means changing it later in twenty places.

A repeated `id`. It must be unique; duplicates break CSS, JS, and the `label`-to-field link.

Markup five `<div>` levels deep. A sign the AI built the structure around the appearance rather than the meaning.

CHEAT SHEET

- Nesting is the structure; indentation affects nothing.
- `class` for many, `id` for one, `data-*` for your own data.
- A `<div>` with a click handler should almost always be a `<button>`.
- In a form check: `name`, a linked `label`, constraints.
- Browser validation does not replace server validation.
- `viewport` in the `<head>`, or the page shrinks on a phone.

8.3 Reading CSS

selectors, the cascade, layout

WHAT IT IS

CSS answers two questions: **what to apply it to** (the selector) and **what to do** (the properties). The difficulty isn't in the properties — there are simply a lot of them and you look them up. The difficulty is in two other places: why a rule didn't apply, and why a box is in the wrong place.

Both are covered below, and both on real screenshots of what the browser drew.

WHAT IT LOOKS LIKE IN CODE

```
.card {
  display: flex;
  gap: 8px;
  padding: 12px;
  border: 1px solid #d8dce5;
}

.card__title {
  font-size: 18px;
  font-weight: 600;
}

.card:hover { background: #f7f8fa; }
```

Selectors

Selector	What it applies to
`.card`	elements with the class `card`
`#main`	the element with `id="main"`
`article`	every `<article>` tag
`.card .title`	`.title` **inside** `.card`
`.card > .title`	`.title` as a direct child
`.card:hover`	`.card` under the cursor
`[data-city="oslo"]`	by attribute value

A space between classes means “inside”, and it is the most common source of confusion: `.card .title` and `.card.title` are different things. The second is an element that has **both** classes at once.

The cascade: who wins

When two rules fit an element, the “heavier” one wins. In increasing order of weight:

What	Weight
a tag: `article`	low
a class: `.card`, `:hover`, `[data-*]`	medium
an `id`: `#main`	high
`style="..."` right in the tag	very high
`!important`	beats everything

At equal weight the one **lower in the file** wins.

Hence a practical consequence: `!important` in generated CSS is almost always a sign the AI did not work out which rule was in the way and overrode it by force. It will work; the next change will be harder.

Layout: `display` decides everything

The same markup of three boxes, three values of `display`:



Fig. 8.2 · The same markup under different `display` values. A real screenshot — that is how the browser drew it.

- **`block`** — each element on its own line, across the full available width. That is how `div`, `p`, `h1` behave by default.
- **`flex`** — the children line up in a row, sized to their content. For rows of buttons, cards, panels.
- **`grid`** — the children are laid out on a defined grid of columns.

Ninety per cent of modern layout is `flex` and `grid`. If generated CSS uses `float` or `position: absolute` to place boxes, that is the outdated approach (article 3.1, reason 3).

The box: where the width comes from

The most common surprise in CSS. Two boxes, both with `width: 200px`, `padding: 20px`, `border: 4px`:



Fig. 8.3 · The red line is the real width. With `content-box` the box is wider than specified by the padding and the border.

Measured in the browser:

```
content-box: 248px
border-box:  200px
```

With `box-sizing: content-box` (the default) `width` sets the width of the **content**, and padding and border are added on top: 200 + 20·2 + 4·2 = 248. With `border-box`, `width` sets the width of the whole box.

Which is why almost every project starts with:

```
* { box-sizing: border-box; }
```

> **REMEMBER**
> If boxes "don't fit" or "stick out" by exactly the size of the padding, it is almost always `box-sizing`. That is the first thing to check.

WHAT TO TELL THE AI

> **PROMPT**
> `Why isn't this rule applying to .card__title?`
> `Which selector overrides it? Don't change the code, explain.`

> **PROMPT**
> `Lay these cards out in three columns with a 16-pixel gap, and`
> `in one column on a narrow screen. Use grid, no float and no`
> `position: absolute.`

PROMPT

```
There is an !important in this CSS. Work out which rule was
in the way and rewrite it without one.
```

WHAT TO LOOK FOR

`!important`. A sign of a conflict overridden rather than solved.

Magic numbers. `margin-top: 37px`, `width: 433px` — eyeballed for one screen and falling apart on another.

`position: absolute` **for layout.** The element drops out of the flow, its neighbours stop seeing it, and at another screen size everything slides.

A fixed height on a text block. `height: 40px` on an element with text: with another font or a long word the text gets clipped.

Repeated values. The same gap written into ten places. Modern CSS has variables for that: `var(--gap)`.

Styles "just in case". `float: left` next to `display: flex` — one of the two does nothing, and it isn't clear which.

CHEAT SHEET

- The selector answers "to what", the properties "what to do".
- A space in a selector means "inside": `.a .b` ≠ `.a.b`.
- Weight: tag < class < id < `style` < `!important` at equal weight, the lower one.
- `!important` in AI code is an overridden conflict, not a solution.
- Layout today is `flex` and `grid` `float` is obsolete.
- Box wider than you set? Check `box-sizing`.

8.4 Developer tools

console, network, elements

WHAT IT IS

Developer tools are the panel built into the browser that shows what is actually happening on a page. It opens with **F12** (or Ctrl+Shift+I, on macOS Cmd+Option+I).

For the frontend it is what logs are for a server (article 4.2): the only way to find out what happened instead of guessing.

There are many tabs; three are needed.

WHAT IT LOOKS LIKE IN CODE

Here is what the browser shows on a page with three different problems:

```
[network ✗]  https://example.invalid/style.css —
net::ERR_TUNNEL_CONNECTION_FAILED
[console error]   Failed to load resource:
net::ERR_TUNNEL_CONNECTION_FAILED
[console log]     app started
[console warning] API key is not set, running in demo mode
[console log]     received: {"main":null}
[page error]      Cannot set properties of null (setting 'textContent')
```

Six lines, and three different stories in them:

1. **The stylesheet didn't load.** The page still appeared — just unstyled. That is the fourth cause of "the style didn't apply" from article 8.1, and this is the only place you can see it.
2. **A demo-mode warning.** Not an error, but it explains why the data looks odd.
3. `Cannot set properties of null` — the code looked for the element `#missing`, didn't find it, got `null` and tried to write text into it. Familiar: it is the same class of error as `KeyError` in Python and `'temp' is possibly null` in TypeScript.

> **REMEMBER**
> `Cannot read/set properties of null` is the most common frontend error, and it almost always means one thing: the selector didn't find the element. Either a typo in the selector, or the script ran before the element existed.

The Console tab

What to look at here:

- **errors** — in red, with the place in the code;
- **warnings** — in yellow, often explaining the cause;
- `console.log` **output** from the code.

The console doesn't only show, it also takes commands. You can check whether an element exists right there:

```
document.querySelector('.card')      // null — not found
document.querySelectorAll('.card').length
```

That is the fastest answer to "is the element in the DOM at all?" from article 8.1.

The Network tab

A list of every request: stylesheets, scripts, images, API calls. Three columns to watch:

Column	What it tells you
Status	the response code (article 7.2): `200`, `404`, `500`
Size	whether a megabyte loaded instead of a kilobyte
Time	what exactly is slow

Clicking a request shows the headers and the body — the same thing `curl` produced in chapter 7, only for a live page.

The most common finding: the API request went out, the response was `200`, and the data is wrong. Which means the problem is not in the frontend but in the backend — and from there, article 7.8.

The Elements tab

It shows the **DOM**, not the original HTML. The difference is fundamental: here you see what came out of parsing and of the scripts running (article 8.1).

On the right is the styles panel for the selected element. There you can see:

- which rules applied;
- **which are struck through** — that is, overridden by others (article 8.3);
- where each rule came from: file and line.

A struck-through rule is a ready-made answer to "why didn't this style apply".

WHAT TO TELL THE AI

The main value of the tools when working with an AI is that they turn "doesn't work" into facts (article 4.3):

DON'T DO THIS

```
the styles aren't applying, fix it
```

BETTER

```
In the console: Failed to load resource — style.css, net::ERR_...
In the Network tab: request to /style.css, status 404.
The path in the markup: <link rel="stylesheet" href="/style.css">.
The file is at static/style.css. What should the path be?
```

PROMPT

```
Error: Cannot set properties of null (setting 'textContent').
Here is the code: <code>. The #missing element is in the markup.
Why does querySelector return null?
```

PROMPT

```
In the Elements tab the color rule from .card is struck through.
Here are both rules: <code>. Which one wins and why?
```

WHAT TO LOOK FOR

The console was never opened. The most common reason for a long search: the error was there in writing the whole time.

Errors that aren't yours. Browser extensions pour into the same console. Check the file and line: if it isn't your file, it isn't your problem.

A `console.log` left in the code. On the frontend that is also a leak: visible to anyone who opens the console. Before shipping, search the diff (article 4.5).

Looking at the source instead of the DOM. `Ctrl+U` shows what came from the server; the Elements tab shows what is there now. With scripts running those are different things.

The cache. The change isn't visible because the browser took the old file. A hard refresh is Ctrl+Shift+R.

CHEAT SHEET

- F12, three tabs: Console · Network · Elements.
- `Cannot read properties of null` — the selector didn't find the element.
- The check in the console: did `document.querySelector('.card')` return `null`?
- Network: status, size, time. A `404` on the styles explains "everything slid".
- Elements shows the DOM, not the original HTML.
- A struck-through rule in the styles panel was overridden.

8.5 AI mistakes in the frontend, and a checklist

checking with your eyes is checking too

WHAT IT IS

The chapter's closer. The frontend differs from everything earlier in the book in one way: **here the result is checked with your eyes.** The code ran, there are no errors, and the button doesn't press or the text overlaps the image — and the only way to learn that is by looking.

AI MISTAKES IN THE FRONTEND

1. `div` instead of semantics
`<div class="btn">` with a click handler. Doesn't work from the keyboard, invisible to screen readers (article 8.2).

2. Outdated layout
`float`, `position: absolute` and tables for placing boxes. Code from the era there is more of in the training data (article 3.1, reason 3). Today that is `flex` and `grid`.

3. `!important` instead of understanding
It overrides a conflict by force without resolving it (article 8.3).

4. Magic numbers
`margin-top: 37px`, `width: 433px` — eyeballed for one screen.

5. No responsiveness
The page looks good at exactly the width it was composed at. On a phone, horizontal scrolling.

6. Styles inline in the markup
`style="..."` on every element: it overrides external rules and cannot be reused.

7. Accessibility forgotten
No `alt`, no `label`, text contrast below the standard, focus invisible. An AI adds these only if asked outright.

8. The script runs before the markup
`<script>` in the `<head>` with no `defer` — `querySelector` returns `null`, because the element doesn't exist yet (article 8.4).

A FRONTEND CHECKLIST

Six steps on top of the general acceptance checklist (article 4.7). The first three with your eyes, the second three in the developer tools.

1. Look at it
Open it and look. Obvious, and it is the chapter's main check: neither tests nor types will tell you the text has landed on top of the button.

2. A narrow screen
Shrink the window to phone width. Horizontal scrolling appeared — there is no responsiveness.

3. Long text
Type a string three times longer than expected into a field or a heading. A layout built around short labels falls apart exactly here.

4. The console
F12, Console tab. There should be no red.

5. The network
Did every file load? A `404` on a stylesheet or a script explains most of the strangeness.

6. The keyboard
Walk the page with Tab. Every button and link must receive focus, and the focus must be visible. That is thirty seconds and the most common finding in generated markup.

> **TIP**
> Steps 2, 3 and 6 are the ones almost nobody does and the ones that catch almost everything. Narrow screen, long text, keyboard: a minute for all three.

WHAT TO TELL THE AI

PROMPT

```
Check this markup against the list:
- are there divs with handlers instead of button and a
- is float or position: absolute used for layout
- are there !important and magic pixel numbers
- what happens on a 360-pixel-wide screen
- do images have alt and fields have label

Which of these are problems? If none, say so.
```

PROMPT

```
Make this block responsive: three columns on a wide screen,
one on a narrow one. With grid, without a media query per
width — use auto-fit and minmax.
```

PROMPT

```
Check accessibility: heading order, alt, label, contrast,
focus visibility. A list of problems with line numbers.
```

CHEAT SHEET

- The frontend is checked by eye: neither types nor tests show a broken layout.
- Eight common mistakes: `div` instead of a button, `float`, `!important`, magic numbers, no responsiveness, inline styles, no accessibility, script before markup.
- Checklist: look · narrow screen · long text · console · network · Tab.
- Narrow screen, long text and the keyboard catch almost everything and cost a minute.
- There should be no red in the console.
- An AI adds accessibility only when asked directly.

9 JavaScript and TypeScript

The language the whole frontend is written in, and the layer on top that makes it checkable. Types here are not a formality — they catch, before anything runs, exactly the class of error chapter 4 is about.

9.1 Reading JavaScript

six places where Python intuition breaks

WHAT IT IS

JavaScript resembles Python just closely enough to mislead. Loops, functions and conditions look familiar — and then the code does something else, and it isn't clear why.

This article covers six places where intuition lets you down. Everything below is real `node` output, not a retelling of the documentation.

WHAT IT LOOKS LIKE IN CODE

1. Sorting numbers doesn't sort numbers

```
[10, 9, 1, 100].sort()
```

```
[ 1, 10, 100, 9 ]
```

By default `sort` turns the elements into strings and compares them as text. Numbers need a comparison function:

```
[10, 9, 1, 100].sort((a, b) => a - b)     // [ 1, 9, 10, 100 ]
```

The mistake is silent: with numbers from 1 to 9 you don't notice it; with two digits it breaks.

2. Two kinds of equality

```
'1' ==  1   ->  true
'1' === 1   ->  false
null ==  undefined   ->  true
null === undefined   ->  false
```

`==` converts types before comparing, `===` doesn't. **The rule is simple: always `===`.** The one justified exception is `x == null`, which checks for `null` and `undefined` in a single go.

3. Truthiness is not what you're used to

```
[]        -> truthy      (in Python an empty list is false!)
{}        -> truthy
""        -> falsy
"0"       -> truthy
0         -> falsy
NaN       -> falsy
null      -> falsy
undefined -> falsy
```

The first line breaks more code than any other. The "are there any items" check written as `if (items)` is **always true** in JavaScript, even for an empty array. It has to be `if (items.length)`.

4. Plus concatenates, minus subtracts

```
1 + '1'   ->  '11'      (joined the strings)
1 - '1'   ->  0         (converted to numbers)
```

Hence the classic: the data arrived from a form or from JSON as strings, and instead of a sum you get a join. `'10' + 5` gives `'105'`.

5. Two kinds of "nothing"

	What it means
`undefined`	there was no value: no such key, the function returned nothing
`null`	there is a value, and it is "nothing"

Reaching for a key that doesn't exist **doesn't fail**:

```
user.email               ->  undefined
user.email?.length       ->  undefined
user.email ?? 'not set'  ->  'not set'
```

This is the opposite of Python, where that gives a `KeyError`. Convenient and dangerous at once: the mistake doesn't announce itself, it surfaces later as `Cannot read properties of undefined` (article 8.4).

Two operators you will meet constantly:

- `?.` — "if the left side isn't `null`/`undefined`, keep going, otherwise return `undefined`";
- `??` — "take the left one, and if it is `null`/`undefined`, take the right one".

`??` differs from `||` in not treating `0` and `""` as emptiness. For numbers that is decisive: `temp || 20` turns zero degrees into twenty, `temp ?? 20` doesn't.

6. `const` doesn't make an object immutable

```
const cities = ["Oslo"];
cities.push("Bergen");      // allowed
cities = [];                // not allowed
```

`const` forbids reassigning the variable, not changing the contents. The same thing as references in Python (article 2.4).

WHAT TO TELL THE AI

PROMPT
```
Rewrite this code with strict comparisons: === everywhere
instead of ==, array checks through .length, ?? instead of ||
where the value may be 0 or an empty string.
Don't change the behaviour.
```

PROMPT
```
This code came from Python. What will behave differently here
because of JavaScript's quirks? Just the list of differences.
```

PROMPT
```
The numbers arrive from JSON as strings. Where in this code
could a concatenation happen instead of an addition?
```

TIP
Most of these traps are caught by a linter. One `eslint` setting with the `eqeqeq` rule and `==` stops passing. For JavaScript without types a linter is mandatory: it is the only mechanical check left.

WHAT TO LOOK FOR

`==` in generated code. It should almost always be `===`.

`sort()` without a comparison function on numbers.

`if (array)` instead of `if (array.length)`. The condition is always true.

`||` for default values. Breaks on `0` and `""`. You want `??`.

`var`. An outdated declaration with surprising scope. New code uses only `let` and `const` (article 3.1, reason 3).

Numbers from strings without conversion. `Number(x)` or `parseInt(x, 10)` — and always with the base 10.

CHEAT SHEET

- `sort()` sorts as strings by default: `[1, 10, 100, 9]`.
- Always `===`. The exception is `x == null` for both emptinesses at once.
- An empty array is truthy. Check `.length`.
- `+` with a string concatenates, `-` converts to numbers.
- `?.` is safe access, `??` is a default value (not to be confused with `||`).
- `const` forbids reassignment, not mutation.

9.2 Asynchrony in JavaScript

promises, await and four places where an AI gets it wrong

WHAT IT IS

Asynchrony in JavaScript is arranged the same way as in Python (article 2.10): `async`, `await`, waiting instead of idling. But it has its own terminology and its own traps, and an AI reproduces all four below regularly.

And here is a direct parallel with chapter 6: `Promise.all` and `Promise.allSettled` relate to each other exactly as `gather` and `gather(return_exceptions=True)` do.

WHAT IT LOOKS LIKE IN CODE

```
const fetchCity = (city, ms = 100) =>
  new Promise((resolve, reject) =>
    setTimeout(() => city === "Bergen"
      ? reject(new Error("server returned 500"))
      : resolve(`${city}: +4`), ms));
```

A **promise** is a promise of a result that will appear later. It has three states: pending, fulfilled, rejected. `resolve` is success, `reject` is an error.

Writing `new Promise` by hand is almost never necessary: promises come back from `fetch` and from libraries. But you do need to know what one is — the word comes up constantly.

1. `Promise.all` drops everything

```
await Promise.all([fetchCity("Oslo"), fetchCity("Bergen"),
fetchCity("Tromsø")]);
```

```
failed entirely: server returned 500
```

The requests to Oslo and Tromsø succeeded — their results are lost. Exactly the same as `asyncio.gather` without protection in Python.

2. `Promise.allSettled` keeps the successes

```
const r = await Promise.allSettled([...]);
for (const x of r)
  console.log(x.status === "fulfilled" ? `ok: ${x.value}` : `error:
${x.reason.message}`);
```

```
ok: Oslo: +4
error: server returned 500
ok: Tromsø: +4
```

Each element is an object with a `status`. A successful one has a `value`, a failed one a `reason`. Checking `status` is mandatory: otherwise the error travels on as an ordinary value.

What you need	What to take
All of them must succeed or there is no point	`Promise.all`
A partial result is better than nothing	`Promise.allSettled`
The first successful answer is enough	`Promise.any`
The first answer of any kind counts, even an error (a race against a timeout)	`Promise.race`

3. `forEach` doesn't wait

The sneakiest of the four:

```
const done = [];
["Oslo", "Tromsø"].forEach(async c => { done.push(await
fetchCity(c)); });
console.log("right after forEach:", done);
```

```
right after forEach: []
after 300 ms:        [ 'Oslo: +4', 'Tromsø: +4' ]
```

The array is empty. `forEach` cannot wait for `async` functions: it starts them all and moves on without waiting. The data does arrive — but later, once the code that needed it has already finished.

The right way is an ordinary loop:

```
for (const c of ["Oslo", "Tromsø"]) ok.push(await fetchCity(c));
```

```
after the loop: [ 'Oslo: +4', 'Tromsø: +4' ]
```

Or `Promise.all` with `map`, if the requests are independent and you want them in parallel.

> **CAREFUL**
> `forEach` with `async` inside is a mistake you can't see while reading: it looks like an ordinary loop. The sign in the code is `forEach(async ...)`. Replace it with `for...of` or `Promise.all(items.map(...))`.

4. A forgotten `await`

```
without await: Promise { <pending> }
with await:    Oslo: +4
```

Instead of a value, a promise object. In Python that is `<coroutine object>`, here `Promise { <pending> }` — the same symptom, a different word.

WHAT TO TELL THE AI

> **PROMPT**
> These three requests are independent. I need a partial result: if one server doesn't answer, the rest should still come through. Use Promise.allSettled and handle status.

> **PROMPT**
> This code has a forEach with async inside. Rewrite it so the results are actually awaited: for...of if the requests depend on each other, or Promise.all with map if they don't.

> **PROMPT**
> Add a 5-second timeout to each request with AbortSignal.timeout. On a timeout, a console warning and skip, not a crash.

WHAT TO LOOK FOR

`Promise.all` where a partial result is wanted. One error loses everything.

`forEach(async ...)`. Doesn't wait. Always a mistake if the result is needed.

No `.catch` and no `try`. A rejected promise nobody caught gives an `UnhandledPromiseRejection` — in Node that means the process dies.

Sequential `await`s in a loop where parallel would do. Not an error, but three times slower on three requests (article 2.10).

No timeout. A request can hang forever (article 6.4). Both the browser and Node have `AbortSignal.timeout(ms)` for that.

`.then()` mixed in with `await`. It works; it reads badly. New code uses `await`.

CHEAT SHEET

- A promise is a promise of a result; `resolve` success, `reject` error.
- `Promise.all` drops everything at the first error, `allSettled` keeps the successes.
- With `allSettled` check `status`: `fulfilled` → `value`, `rejected` → `reason`.
- `forEach(async ...)` doesn't wait. You want `for...of` or `Promise.all(map)`.
- `Promise { <pending> }` in the output means a forgotten `await`.
- An uncaught rejected promise kills the process in Node.

9.3 What TypeScript adds

the same JavaScript, but checked

WHAT IT IS

TypeScript is JavaScript plus type descriptions. The browser doesn't execute it: before running, the types are **erased**, and ordinary JavaScript is left. All of the benefit happens before the run, at the moment of checking.

For working with an AI that is decisive. Go back to article 6.3: there `mypy` caught a forgotten `null` check in Python before anything ran. TypeScript does the same for the frontend — and does it more strictly, because here types are not an optional extra but the normal way of working.

WHAT IT LOOKS LIKE IN CODE

```
interface City {
  name: string;
  temp: number | null;
  tags?: string[];
}

function report(c: City): string {
  if (c.temp === null) return `${c.name}: no data`;
  return `${c.name}: ${c.temp > 0 ? "+" : ""}${c.temp}°`;
}

const cities: City[] = [
  { name: "Oslo", temp: 4 },
  { name: "Tromsø", temp: null },
  { name: "Bergen", temp: -2, tags: ["rain"] },
];

cities.forEach(c => console.log(report(c)));
```

```
tsc: no errors
Oslo: +4°
Tromsø: no data
Bergen: -2°
```

How to read it

`interface City { ... }` — a description of an object's shape. Not a class, not code: after compilation nothing is left of it.

`temp: number | null` — a union, "either one or the other". The most useful notation in TypeScript: it says outright that the value may be missing, and the compiler will force you to check.

`tags?: string[]` — an optional field. Its type is in fact `string[] | undefined`.

`function report(c: City): string` — takes a `City`, returns a string. The same thing as a signature in Python (article 6.5), and it works the same way: it replaces a paragraph of explanation.

What happens without the `null` check

Remove the `if` and ask the compiler to check:

```
function report(c: City): string {
  return `${c.name}: ${c.temp.toFixed(1)}°`;
}
```

```
error TS18047: 'c.temp' is possibly 'null'.
```

The code won't compile. It won't fail for a user, won't slip past the tests — it simply won't build until you add the check.

The same error in Python looked like `Item "None" of "str | None" has no attribute "upper"`, and in JavaScript without types like `Cannot read properties of null` already at run time (article 8.4). Three languages, one class of error, and only here is it caught before the run.

> **REMEMBER**
> The main thing TypeScript gives you is not "beautiful code" but the fact that a forgotten emptiness check stops being possible. That is the most frequent AI mistake in the whole book, and it is closed by a setting.

`strict` isn't an option, it is the point

```
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true
  }
}
```

Without `strict` the `null` check is off, and TypeScript turns into decoration. Switching it on is mandatory.

`noUncheckedIndexedAccess` is the second most useful setting. With it, indexing gives `T | undefined`, because the element may not be there:

```
const cities = ["Oslo", "Bergen"];
const first: string = cities[5];
```

```
error TS2322: Type 'string | undefined' is not assignable to type
'string'.
```

The compiler reminds you of the thing everyone forgets: an index can go out of bounds.

How to run it

```
npx tsc --noEmit        # only check, don't build anything
npx tsx app.ts          # run directly
```

`--noEmit` is what you want most often: a check without a build, the counterpart of `mypy --strict`.

WHAT TO TELL THE AI

PROMPT
```
Port this module to TypeScript: interfaces for the data, types
on every function. tsc --noEmit with strict must pass.
Don't change the logic.
```

PROMPT
```
In tsconfig turn on strict and noUncheckedIndexedAccess.
Fix the errors that appear without disabling the checks and
without as any.
```

That second sentence matters: the simplest way to "fix" a type error is to silence it, and an AI is inclined to do exactly that (article 9.6).

PROMPT
```
Describe the types for this API's response: here is a sample
JSON. Mark fields that may be absent as optional or | null —
honestly, not "so that it compiles".
```

WHAT TO LOOK FOR

`strict: false`, or its absence. The `null` check is off and the main benefit is lost.

`any` in the types. Switches checking off completely for that spot.

`as` to calm the compiler down. That gets its own treatment in article 9.5.

Types written but `tsc` never run. The same as annotations without `mypy`: decoration that has drifted away from the code.

`.js` files mixed in with `.ts`. A half-finished migration: there are no checks in the `.js` files, and the errors hide there.

CHEAT SHEET

- TypeScript is JavaScript plus types; at run time the types are erased.
- `interface` describes a shape, `|` is a union, `?` an optional field.
- `number | null` forces an emptiness check — otherwise the code won't build.
- `strict: true` is mandatory, or there is no `null` checking.
- `noUncheckedIndexedAccess` — indexing gives `| undefined`.
- `npx tsc --noEmit` is a check without a build, the counterpart of `mypy --strict`.

9.4 How to read other people's types

unions, narrowing, generics, utility types

WHAT IT IS

The types an AI or a library writes look harder than the ones you write yourself: angle brackets, vertical bars, unfamiliar words. They read more easily than they look — you need to know four constructs.

This article is about reading. You won't have to write types like these from scratch.

WHAT IT LOOKS LIKE IN CODE

Unions and narrowing

```ts
type Result =
  | { ok: true; data: string[] }
  | { ok: false; error: string };

function show(r: Result): string {
  if (r.ok) return `received ${r.data.length}`;
  return `error: ${r.error}`;
}
```

```
show({ ok: true, data: ["Oslo", "Bergen"] })  ->  received 2
show({ ok: false, error: "timeout" })         ->  error: timeout
```

It reads like this: *`Result` is either an object with `ok: true` and `data`, or an object with `ok: false` and `error`.*

The interesting part is what happens inside the `if`. After the `r.ok` check the compiler **knows** which of the two variants it is looking at: in the first branch there is a `data` and no `error`, in the second the other way round. Reach for `r.error` in the first branch and you get a compile error.

This is called **narrowing**, and it is the main reason unions are used. That "flag plus data" pair is the standard way to return "it worked or it didn't" without throwing an exception.

Ordinary checks narrow too:

Check	What it narrows
`if (x === null)`	removes `null` from the union
`typeof x === "string"`	leaves strings
`Array.isArray(x)`	leaves arrays
`if ("data" in r)`	by the presence of a field

Generics

```
function firstOr<T>(items: T[], fallback: T): T {
  return items.length > 0 ? items[0]! : fallback;
}
```

```
firstOr(["Oslo"], "none")  ->  Oslo      (T = string)
firstOr([10, 20], 0)       ->  10        (T = number)
```

`<T>` isn't a concrete type but a **placeholder**: "whatever type comes in is what comes back out". Thanks to that the function works with any data without losing the type: the result of `firstOr([10, 20], 0)` is a number, not "something".

Read generics as a template with a hole: substitute a concrete type from the call for `T` and everything becomes clear.

You will meet them constantly: `Array<string>` (the same as `string[]`), `Promise<City>` (a promise that will return a `City`), `Map<string, number>`.

Utility types

Ready-made type transformations. Four of them cover almost everything:

```ts
interface City { name: string; temp: number; country: string }

type CityPatch     = Partial<City>;               // every field
optional
type CityKey       = Pick<City, "name" | "temp">; // only these
type CityNoCountry = Omit<City, "country">;       // all but
type ByName        = Record<string, City>;        // a dictionary
```

Notation	Reads as
`Partial<T>`	"all the fields of `T`, but optional"
`Pick<T, K>`	"only the fields `K` from `T`"
`Omit<T, K>`	"all the fields of `T` except `K`"
`Record<K, V>`	"a dictionary: keys `K`, values `V`"

`Partial` turns up most often on update functions: you can send a subset of the fields. `Record<string, City>` is the same as `dict[str, City]` in Python.

> **TIP**
> Don't understand a type? Hover over it in the editor. It shows the expanded form: what `Omit<City, "country">` actually turns into. That is faster than taking it apart by hand.

What else you will meet

`unknown` — "the type is unknown, and until you check it you can do nothing with it". The right replacement for `any`: it forces a check.

`as const` — "this value won't change"; it turns `"paid"` from just a string into that exact value.

`keyof City` — a union of the field names: `"name" | "temp" | "country"`.

WHAT TO TELL THE AI

> **PROMPT**
> Explain this type piece by piece, as if I were seeing generics for the first time. What gets substituted in and what comes out.

> **PROMPT**
> Rewrite this function so it returns { ok: true, data } | { ok: false, error } instead of throwing an exception.

> **PROMPT**
> There are three nearly identical interfaces here. Can they be expressed with Pick/Omit from one base? Show how.

WHAT TO LOOK FOR

A union with no narrowing. If the type is `A | B` and there isn't a single check in the code, the compiler was silenced with an `as`.

A generic that is `any` everywhere. `<T = any>` switches off all the benefit.

Duplicated types. Three interfaces with overlapping fields instead of `Pick`/`Omit` from one. They will drift apart at the first edit.

`unknown` that is immediately cast with `as`. The point of `unknown` is to check, not to get around.

A type that is too clever. If a type runs ten lines and nobody can explain it, it is harder than the problem. An AI sometimes gets carried away.

CHEAT SHEET

- `A | B` is one or the other; a check narrows it to one.
- The pair `{ ok: true, data } | { ok: false, error }` is the standard "it worked or it didn't".
- `<T>` is a placeholder: whatever type came in is what comes out.
- `Partial` optional, `Pick` only these, `Omit` all but, `Record` a dictionary.
- `unknown` is the honest replacement for `any`: it forces a check.
- Don't understand a type? Hover over it; the editor shows the expanded form.

9.5 Types as a check on the AI's work

five errors caught before the run, and one that gets through

WHAT IT IS

The chapter's central article. Go back to chapter 4: ten articles there on catching **by hand** the errors you can't see when the code runs. Types do part of that work mechanically, at every check, for free.

Below are five real AI mistakes and the compiler's real messages for them. And then an honest limitation: one error of the same kind slips past, and knowing about it matters more than the other five.

WHAT IT LOOKS LIKE IN CODE

1. Forgot the emptiness check

The most frequent AI mistake in the whole book:

```
function report(c: City): string {
  return `${c.name}: ${c.temp.toFixed(1)}°`;
}
```

```
error TS18047: 'c.temp' is possibly 'null'.
```

2. Got a field name wrong

```
const c: City = { name: "Oslo", temperature: 4 };
```

```
error TS2353: Object literal may only specify known properties,
and 'temperature' does not exist in type 'City'.
```

In JavaScript without types this would quietly have created an object with one extra field and without the one needed. The error would have surfaced somewhere else and looked like `undefined`.

3. Got the argument order wrong

```
function subscribe(userId: number, city: string): void {}
subscribe("Oslo", 7);
```

```
error TS2345: Argument of type 'string' is not assignable
to parameter of type 'number'.
```

A classic when working with an AI: it remembers a function with a similar name and a different argument order.

4. Forgot a variant of a union

```
type Status = "paid" | "pending" | "refunded";

function label(s: Status): string {
  switch (s) {
    case "paid": return "paid";
    case "pending": return "awaiting";
  }
}
```

```
error TS2366: Function lacks ending return statement
and return type does not include 'undefined'.
```

`refunded` is missing. Without types the function would silently have returned `undefined`, and an empty space would have appeared in the interface instead of a label.

This is the important case: add a fourth status tomorrow and the compiler will find every place that doesn't handle it. You don't find that by hand.

5. Indexing out of bounds

```
const cities = ["Oslo", "Bergen"];
const first: string = cities[5];
```

```
error TS2322: Type 'string | undefined' is not assignable to type
'string'.
```

Only works with `noUncheckedIndexedAccess` (article 9.3).

WHERE TYPES DON'T HELP

Now, honestly. Here is code the compiler lets through **without a single error**:

```
interface WeatherResponse { main: { temp: number }; name: string }

async function fetchWeather(city: string): Promise<WeatherResponse> {
  const raw = '{"cod":"404","message":"city not found"}';
  return JSON.parse(raw) as WeatherResponse;
}

const data = await fetchWeather("Tonsberg");
console.log(data.main.temp);
```

```
tsc: no errors
```

```
failed: Cannot read properties of undefined (reading 'temp')
```

`as WeatherResponse` is a promise to the compiler, not a check. You said "trust me, this has that shape", and it did. Something else arrived — and there is the very same `Cannot read properties of undefined` that chapter 8 opened with.



Fig. 9.1 · Types check your code against itself. Data that arrived from outside they only describe — that has to be checked separately.

The boundary is everything that arrived from outside the program:

- an API response (`fetch`, `JSON.parse`);
- form data;
- `localStorage`
- parameters from the page's address.

> **REMEMBER**
> Types check your code's consistency with itself. They **do not check** data that came from outside. That is exactly the same boundary as in article 7.3 on the server: there a request schema closes it, here a check at run time.

What closes the boundary

Run-time validation libraries — `zod`, `valibot` and the like. They do for the frontend what `pydantic` does for the backend (article 7.3): you describe the shape once, and it both checks the data and gives you a type.

PROMPT

```
The API response is currently cast with `as`. Replace it with
a run-time check using zod: schema, parse, a clear error if
the shape doesn't match. Infer the type from the schema, don't
duplicate it by hand.
```

That last sentence matters: the point is for the shape to be described **once** and serve as both the check and the type. Two descriptions side by side will drift apart.

WHAT TO TELL THE AI

Types are also a way of stating the task. A signature is more precise than words (articles 3.4, 6.5):

PROMPT

```
Write a function to this signature:

async function loadSubs(userId: number): Promise<Result<City[]>>

where Result<T> = { ok: true; data: T } | { ok: false; error: string }.
Don't throw exceptions outward, return errors as the second
variant. tsc --noEmit with strict must pass.
```

Checking finished code:

PROMPT

```
Run tsc --strict over your own code in your head: what errors
would it find? Fix them without using any, as or ts-ignore.
```

The prohibition in the last line is mandatory — otherwise the "fix" will be a gag (article 9.6).

CHEAT SHEET

- Types catch before the run: a forgotten emptiness, a wrong field, argument order, a missed union variant, going past the end of an array.
- Add a variant to a union and the compiler finds every unhandled place.
- `as` is a promise to the compiler, not a check. It doesn't look at outside data.
- The boundary: API responses, forms, `localStorage`, the page's address.
- The boundary is closed by a run-time check, `zod` for example.
- Infer the type from the schema instead of duplicating it alongside.

9.6 AI mistakes in JS and TS, and a checklist

how to tell a fix from a gag

WHAT IT IS

The chapter's closer. JavaScript and TypeScript have a peculiarity of their own: **here an AI has an easy way to make the check shut up.** Three characters, and the error is gone without having been fixed.

Telling a fix from a gag is the main skill in this article.

THREE GAGS TO RECOGNIZE

1. `any`

```
function handle(data: any) { ... }
```

Switches checking off entirely. Everything coming out of an `any` the compiler stops looking at, and the infection spreads further through the code.

The honest replacement: `unknown`, which forces a check, or a proper type.

2. `as`

```
const city = data as City;
```

A promise to the compiler, taken apart in article 9.5. More dangerous than `any`, because it looks like typed code.

When it is justified: rarely and deliberately — after a check of your own that the compiler doesn't understand, for instance. Then there should be a comment next to it saying why.

3. `@ts-ignore` and `@ts-expect-error`

```
// @ts-ignore
widget.render(options);
```

Switches off checking for the next line. Always needs an explanation: if the reason isn't written down, in a month nobody will remember what is hidden here.

> **CAREFUL**
> If `any`, `as` or `ts-ignore` appeared in the diff after you asked it to "fix the type errors", the errors are not fixed but hidden. That is the direct analogue of `except Exception: pass` from article 2.14, only for types.

THE OTHER FREQUENT MISTAKES

1. `strict` **is off.** Without it there are no `null` checks — the main benefit (article 9.3).
2. `Promise.all` **instead of** `allSettled` when a partial result is wanted (article 9.2).
3. `forEach(async ...)`**.** Doesn't wait. Always a mistake if the result is needed.
4. `==` **instead of** `===`, and `||` instead of `??` (article 9.1).
5. **Types duplicated by hand.** A validation schema in one place, an interface in another — they will drift apart.
6. **Code for old versions.** `var`, `.then()` instead of `await`, class components. There is more training data from that era (article 3.1, reason 3).
7. **A library method that doesn't exist.** A hallucination (article 4.8). In TypeScript the compiler catches it — if the library has types.

A JS AND TS CHECKLIST

Five steps on top of the general acceptance checklist (article 4.7).

1. Type checking

```
npx tsc --noEmit
```

The highest-value step. It catches five classes of error from article 9.5.

2. Search the diff for gags

```
git diff | grep -nE "\bany\b|\bas [A-Z]|ts-ignore|ts-expect-error"
```

One command shows every place where checking is switched off. Each one must be explainable.

3. The linter

```
npx eslint .
```

It catches what types can't see: `==`, unused variables, forgotten `await`s, `forEach` with `async` (given the right rules).

4. Run it

```
npx tsx app.ts
```

A type is no guarantee that outside data will arrive in the right shape (article 9.5). Running it is mandatory.

5. The boundary

Look at where external data enters the code: `fetch`, `JSON.parse`, `localStorage`, forms. Is there a run-time check everywhere, rather than just an `as`?

WHAT TO TELL THE AI

PROMPT

```
Check your code against the list:
- are there any, as or ts-ignore, and why
- is strict on and does tsc --noEmit pass
- is there == instead of === or || where ?? is needed
- is there a forEach with async inside
- Promise.all where a partial result is wanted
- where does data come in from outside and is it validated

Which of these are there? If none, say so.
```

And separately, a request that saves a whole round of back-and-forth:

PROMPT

```
Fix the type errors properly. Don't use any, as or ts-ignore.
If an error can't be fixed without them — explain why, and
don't fix it.
```

That last sentence matters: it gives the AI permission to say "I can't" instead of quietly gagging the check (articles 3.8, 4.8).

CHEAT SHEET

- Three gags: `any`, `as`, `ts-ignore`. If they appeared after "fix the types", the errors are hidden, not fixed.
- `unknown` is the honest replacement for `any`.
- Checklist: `tsc --noEmit` · search the diff for gags · linter · run · the boundary.
- `git diff | grep -E "\bany\b|\bas [A-Z]|ts-ignore"` — one command for the whole diff.
- Types don't check external data: running it is mandatory.
- Ask for fixes without gags, and allow "I can't" as an answer.

10 React in TypeScript

The most widespread way of building interfaces. Everything so far comes together here: the markup from chapter 8, the types from chapter 9 and the code reading from chapter 2 — only now they work together.

10.1 The anatomy of a React project

where everything comes from and why there are so many files

WHAT IT IS

A React project looks frightening: two dozen files at the root, a `node_modules` folder running to hundreds of megabytes, config files for tools you have never heard of. And your own code is one folder.

This article sorts that out, so that in an unfamiliar project you don't spend half an hour looking for the entry point.

WHAT IT LOOKS LIKE IN CODE

```
weather-app/
├── node_modules/        libraries — don't read, don't commit
├── public/              files served as they are
├── src/                 ← your code, and only it
│   ├── main.tsx         entry point: mounts the app
│   ├── App.tsx          root component
│   ├── components/      components
│   │   └── CityCard.tsx
│   ├── hooks/           your own hooks
│   ├── api/             requests to the server
│   └── types.ts         shared types
├── index.html           the page everything is inserted into
├── package.json         dependencies and commands
├── package-lock.json    exact versions — commit this
├── tsconfig.json        TypeScript settings (article 9.3)
├── vite.config.ts       bundler settings
└── .env                 secrets — into .gitignore (article 5.7)
```

Only `src/` needs reading. Everything else is scaffolding.

The entry point

```
import { createRoot } from "react-dom/client";
import App from "./App";

createRoot(document.getElementById("root")!).render(<App />);
```

Three lines, and they hold the whole link between HTML and React: take the `#root` element from `index.html` and draw the application inside it. From there on React draws the entire interface, and `index.html` never changes again.

The exclamation mark after `getElementById` is an `as`-like assertion that "the element is definitely there" (article 9.5). Here it is justified: if there is no `#root`, the application won't start anyway.

What a bundler is

The browser understands neither JSX, nor TypeScript, nor imports from `node_modules`. The bundler (most often **Vite**) turns all of that into ordinary JavaScript.

Hence the two commands you meet in every project:

```
npm run dev     # development mode: an edit is visible at once
npm run build   # a build into dist/ for deployment
```

What exactly they do is written in `package.json`, in the `scripts` section. That is the first place worth looking in an unfamiliar project: it lists everything that can be done with it.

`package.json`

```
{
  "scripts": {
    "dev": "vite",
    "build": "tsc --noEmit && vite build",
    "lint": "eslint ."
  },
  "dependencies": {
    "react": "^19.3.1",
    "react-dom": "^19.3.1"
  },
  "devDependencies": {
    "typescript": "^5.9.0",
    "vite": "^7.1.0"
  }
}
```

Section	What's inside
`scripts`	the project's commands — read these first

`dependencies`	what the application needs to run
`devDependencies`	needed only during development

About `^`: the sign means "this version or newer, within the same major". Which is why you and the server may end up with different versions — and that is exactly what `package-lock.json` protects against, by pinning the exact ones. It is always committed (article 5.6).

> **TIP**
> Note the line `"build": "tsc --noEmit && vite build"`. Type checking is built into the build: if the types don't line up, the build fails. This is the case where you don't have to remember the check from article 9.3 — it runs by itself.

WHAT TO TELL THE AI

PROMPT
```
Show the contents of scripts from package.json and explain
what each command does. One line per command.
```

PROMPT
```
I've opened this project for the first time. Which three files
in src should I start reading to understand how it's put
together?
```

PROMPT
```
Add a CityCard component in src/components/.
Don't change the project structure, don't add new
dependencies.
```

WHAT TO LOOK FOR

`node_modules` in git. Hundreds of megabytes of other people's code in the repository. Checked by the presence of a line in `.gitignore`.

No `package-lock.json`. The versions aren't pinned, and you and the server will end up with different ones.

Everything in one `App.tsx`. A thousand-line file instead of components. It will work; changing it is hard, and an AI won't be able to edit one part without touching the rest.

Secrets in frontend code. An important difference from the backend: **everything that made it into the build is visible to the user.** A `VITE_API_KEY` variable ends up in the page's code. Real keys are kept on the server, and the frontend talks to its own backend (article 7.8).

A mix of `.js` and `.tsx`. A half-finished move to TypeScript: there are no checks in the `.js` files.

CHEAT SHEET

- Read only `src/` the rest is scaffolding.
- The entry point is `main.tsx` it mounts the app into `#root`.
- `package.json` → `scripts`: the list of everything that can be done with the project.
- `^` in versions permits updates; the exact versions live in `package-lock.json`.
- `tsc --noEmit` inside `build` — type checking is built into the build.
- Secrets on the frontend are impossible: everything in the build is visible to the user.

10.2 Components and JSX

markup inside the code

WHAT IT IS

A component is a function that returns markup. That is the whole idea of React: instead of changing the page with commands, you describe **how it should look** given this data, and React brings it to that state itself.

JSX is the syntax that lets you write markup right inside the code. It looks like HTML, but it is JavaScript.

WHAT IT LOOKS LIKE IN CODE

```
interface CityCardProps {
  name: string;
  temp: number | null;
  onRemove: (name: string) => void;
}

export function CityCard({ name, temp, onRemove }: CityCardProps) {
  return (
    <article className="card">
      <h2>{name}</h2>
      <p>{temp === null ? "no data" : `${temp > 0 ? "+" : ""}${temp}°`}
</p>
      <button type="button" onClick={() => onRemove(name)}>Remove</
button>
    </article>
  );
}
```

What comes out of that — real output:

```
<article class="card"><h2>Oslo</h2><p>+4°</p>
  <button type="button">Remove</button></article>
<article class="card"><h2>Tromsø</h2><p>no data</p>
  <button type="button">Remove</button></article>
```

And how it looks on screen:

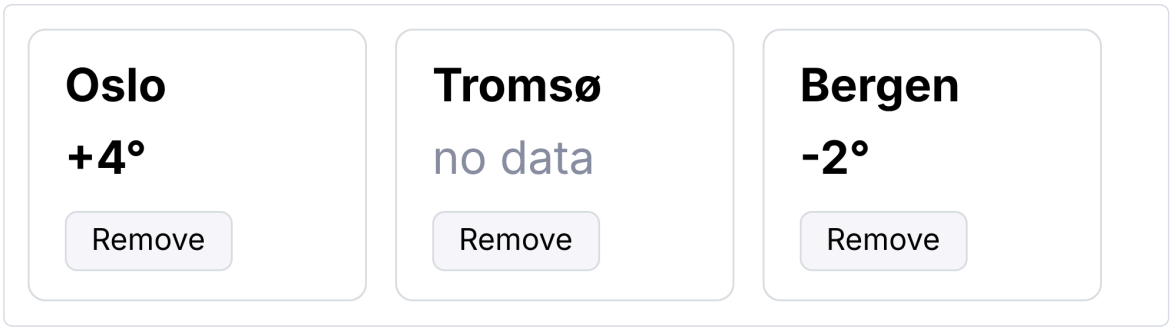


Fig. 10.1 · Three calls to one component with different data. A real screenshot.

How to read it

A component is a function with a capital letter. That is not a style but a requirement: with a lowercase letter React takes it for an ordinary tag.

`{ }` inside the markup is an exit into JavaScript. Everything in curly braces is evaluated: `{name}`, `{temp === null ? ... : ...}`, `{() => onRemove(name)}`.

`className` instead of `class`. `class` is a reserved word in JavaScript. Likewise `htmlFor` instead of `for`.

One element is returned. Two roots side by side is an error. If you need that, they are wrapped in an empty tag, `<>...</>`.

Destructuring in the argument. Writing the parameter as `({ name, temp, onRemove }: CityCardProps)` means “take three fields out of the props object”. Equivalent to `(props: CityCardProps)` and then `props.name`.

Lists and `key`

```
<div className="list">
  {cities.map(c => <CityCard key={c.name} {...c}
onRemove={handleRemove} />)}
</div>
```

`map` instead of a loop. You can’t write a `for` in JSX — instead an array of data is turned into an array of elements.

`key` is mandatory. Not a formality: it is how React knows which element is which when the list changes. Without a `key`, deleting from the middle of a list makes the data "slide" onto the neighbours. React warns about this in the console (article 8.4).

`key` must be stable. A record's identifier is good. An array index is bad: it changes on sorting and deletion, and the point is lost.

`{...c}` is spreading. Pass every field of the object as props. Short, but it hides what is actually being passed: the reader has to go and look at what `c` consists of.

> **REMEMBER**
> React doesn't change the page — it compares what it should look like with what it does look like, and brings the two into line. Which is why a component never contains commands like "add a row", only a description of the result.

Conditions in the markup

```
{temp === null ? <p>no data</p> : <p>{temp}°</p>}   // if-else
{isLoading && <Spinner />}                           // only if
{items.length > 0 && <List items={items} />}         // careful!
```

The third line is a common trap. If `items.length` equals zero, a **"0"** appears on the screen: zero isn't rendered as "nothing", it is printed as a number. The safe forms are `items.length > 0 &&` (as here) or the ternary operator.

WHAT TO TELL THE AI

> **PROMPT**
> ```
> Make a CityCard component: it takes name (a string),
> temp (a number or null) and onRemove (a function of the name).
> If temp is null, show "no data".
> Prop types as a separate interface, export it.
> ```

> **PROMPT**
> ```
> This list is rendered with map and no key. Add a key based on
> the record's identifier, not the index, and explain why.
> ```

> **PROMPT**
> ```
> Split this component into three smaller ones. Don't change the
> logic, only the structure. Show what props each one has.
> ```

WHAT TO LOOK FOR

No `key`, or `key={index}`. The most frequent mistake in generated lists.

`{items.length && ...}`. It will show “0” instead of nothing.

A two-hundred-line component. It breaks into parts — otherwise it can be neither read nor reused.

Logic inside the markup. Three nested ternary operators in JSX read badly. Computations move higher up, into variables.

`{...props}` **everywhere.** Easy to write, hard to read: it isn’t clear what is being passed.

Markup without semantics. The same `div`s instead of `button` and `article` (article 8.2) — React doesn’t fix that.

CHEAT SHEET

- A component is a capitalized function that returns markup.
- `{ }` in JSX is an exit into JavaScript; `className` instead of `class`.
- One root element is returned, otherwise wrap it in `<>…</>`.
- Lists through `map` `key` is mandatory and must be stable.
- `key={index}` breaks on sorting and deletion.
- `{x.length && ...}` will print “0”. Write `x.length > 0 &&`.

10.3 Props and state

where the data comes from, and why the counter counts wrong

WHAT IT IS

Two sources of data in a component, and they mustn’t be confused:

- **props** — came from outside, from the parent. They can’t be changed.
- **state** — lives inside the component. It changes, and on every change the component is drawn again.

Types give you more here than anywhere else in the chapter: props are the boundary between components, and TypeScript watches it strictly.

WHAT IT LOOKS LIKE IN CODE

```
const [count, setCount] = useState(0);
```

`useState` returns a pair: the current value and a function that changes it. The square brackets unpack a two-element array.

Assigning directly doesn’t work. `count = 5` redraws nothing: React learns about a change only through `setCount`.

What the types catch

Here are the compiler’s real messages for four typical mistakes:

A required prop is missing:

```
<CityCard name="Oslo" temp={4} />
```

```
error TS2741: Property 'onRemove' is missing in type
'{ name: string; temp: number; }' but required in type 'CityCardProps'.
```

The wrong type:

```
<CityCard name="Oslo" temp="4" onRemove={() => {}} />
```

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

The wrong handler signature:

```
<CityCard name="Oslo" temp={4} onRemove={(id: number) => {}} />
```

```
error TS2322: Type '(id: number) => void' is not assignable
to type '(name: string) => void'.
```

A forgotten emptiness check:

```
function T({ temp }: { temp: number | null }) { return
<p>{temp.toFixed(1)}</p>; }
```

```
error TS18047: 'temp' is possibly 'null'.
```

All four are classic AI mistakes, and none of the four makes it as far as the run. The third especially: an AI often remembers a handler with a different signature and substitutes it silently.

A trap you have to see

Three buttons, the same job — add one three times:

```
<button onClick={() => { setCount(count + 1); setCount(count + 1);
setCount(count + 1); }}>
  three times in a row
</button>

<button onClick={() => { setCount(c => c + 1); setCount(c => c + 1);
setCount(c => c + 1); }}>
  three times with a function
</button>
```

Measured in the browser:

```
start:                            count: 0 | renders: 1
after one +1:                     count: 1 | renders: 2
after "three times in a row":     count: 2 | renders: 3
after "three times with function": count: 5 | renders: 4
```

Three calls in a row gave **+1**, not +3. Three calls with a function gave **+3**.

The reason: inside the handler `count` is the value **as of the render**, and calls to `setCount` don't change it. Three times in a row it runs

`setCount(0 + 1)`. Writing `setCount(c => c + 1)` asks React to take the current value itself.

Note the render counter too: three calls produced **one** redraw. React collects the changes and applies them in one go — that is called batching.

> **REMEMBER**
> If the new value depends on the old one — always a function: `setCount(c => c + 1)`. That rule closes a whole class of errors that show up only during fast repeated actions and therefore aren't caught by ordinary checking.

Where state should live

The rule: **in the nearest common parent of the components that need it.**

If two sibling components need the state, it is lifted to the parent and passed back down as props — the value itself and the function that changes it:

```
<CityCard name={c.name} temp={c.temp} onRemove={handleRemove} />
```

That is the answer to "why are there so many props": data goes top-down, events bottom-up.

WHAT TO TELL THE AI

> **PROMPT**
> The city list's state is currently in CityCard. Lift it to App, pass the value and the handler down. Describe the prop types with an interface.

> **PROMPT**
> This handler has several setState calls in a row that depend on the previous value. Rewrite them with the functional form setX(prev => ...).

> **PROMPT**
> Describe this component's props with an interface: what's required, what isn't, what types the handlers have. Don't touch the implementation.

WHAT TO LOOK FOR

A prop changed inside the component. `props.items.push(...)` — props can't be changed, they are the parent's data.

`setState` in a row depending on the old value. Taken apart above.

Duplicated state. The same thing held both in the parent and in the child — guaranteed to go out of sync.

State where an ordinary variable would do. If the value doesn't affect what is visible, `useState` isn't needed — it causes extra redraws.

Props with no types, or with `any`. All the benefit from article 9.5 is lost.

Ten props on one component. A sign the component does too much (article 2.5, on the number of arguments).

CHEAT SHEET

- Props come from above and are immutable; state lives inside.
- Direct assignment redraws nothing — only `setX` does.
- New value depends on the old one — the functional form `setX(prev => ...)`.
- Three `setCount(count + 1)` in a row give +1, not +3.
- Several changes give one redraw: React batches.
- Shared state belongs in the nearest common parent.

10.4 Effects

where React breaks beginners most often

WHAT IT IS

`useEffect` is a way to do something that isn't drawing: request data, subscribe to an event, set a timer. It runs **after** the component has been drawn.

It is the hardest place in React and the most frequent place for AI mistakes. One forgotten line turns an application into an endless loop of requests.

WHAT IT LOOKS LIKE IN CODE

```
useEffect(() => {
  // what to do
  return () => { /* how to clean up after yourself */ };
}, [dependencies]);
```

Three parts, and each one matters:

The body — what to run after the render.

The returned function — the cleanup: unsubscribe, cancel the timer, abort the request. It runs before the effect's next run and when the component is removed.

The dependency array — when to repeat. And that is the source of all the trouble.

The dependency array decides everything

Written as	When it runs
no array	**after every render**
`[]`	once, when the component appears
`[userId]`	on appearance and every time `userId` changes

The difference between the first two rows, measured:

```
useEffect(() => {              // no array
  setData([...data, "x"]);
});
```

```
useEffect(() => {              // empty array
  setData(["x"]);
}, []);
```

```
no dependency array:   effect ran 49 times
with empty array [] :  effect ran 1 times
```

Forty-nine times — and only because I put a limiter in. Without one the loop would be endless: the effect changes the state → the component redraws → the effect runs again.

> **CAREFUL**
> An effect with no dependency array that changes state is an endless loop. If it goes to the network, the application starts hammering the server dozens of times a second. One glance checks it: does `useEffect` have a second argument?

Cleanup

```
useEffect(() => {
  const controller = new AbortController();
  fetch(`/api/subs/${userId}`, { signal: controller.signal })
    .then(r => r.json())
    .then(setData)
    .catch(e => { if (e.name !== "AbortError") console.error(e); });
  return () => controller.abort();
}, [userId]);
```

Why the returned function is needed: the user switched city quickly — the first request is still in flight and a second one goes out. Without cancellation the responses can arrive out of order, and the previous city's data ends up on the screen.

That is a race condition, and an AI forgets about it almost every time: the generated code has a `fetch` but neither an `AbortController` nor a returned cleanup function.

When an effect isn't needed

The main misconception is that everything belongs inside a `useEffect`. In fact it is only needed for reaching outside — the network, timers, subscriptions, direct work with the DOM.

Not needed:

```
// bad: extra state and an extra redraw
const [total, setTotal] = useState(0);
useEffect(() => { setTotal(items.length); }, [items]);

// good: just compute it
const total = items.length;
```

If a value can be computed from what you already have, compute it right during the render. State and an effect here only add a step and a redraw.

WHAT TO TELL THE AI

PROMPT
```
This useEffect has no dependency array and changes state
inside. Fix it and explain what was wrong.
```

PROMPT
```
Add request cancellation with AbortController and a returned
cleanup function. Don't show an AbortError to the user — that
isn't an error, it's a cancellation.
```

PROMPT
```
Are useState and useEffect needed here? Could this value just
be computed during the render?
```

That last question is worth asking every time you see a `useEffect` that only sets state from other data.

WHAT TO LOOK FOR

No dependency array. Checked instantly, breaks everything.

An array that is there but incomplete. The effect uses a variable that isn't in the dependencies — it will work with a stale value. A linter with the `react-hooks/exhaustive-deps` rule catches this automatically.

No cleanup on subscriptions and timers. A `setInterval` without `clearInterval` keeps running after the component is removed.

`fetch` **without cancellation.** A race condition on fast switching.

An effect instead of a computation. Extra state, an extra redraw.

A request in an effect with no error handling. The server answered `500` — on screen, an endless loading state.

CHEAT SHEET

- `useEffect` is for reaching outside: the network, timers, subscriptions.
- Dependency array: none — every time, `[]` — once, `[x]` — when `x` changes.
- No array plus a state change = an endless loop.
- Return a cleanup function for subscriptions, timers and requests.
- `fetch` without an `AbortController` gives a race condition.
- If the value can be computed from data you have, no effect is needed.

10.5 Stating interface tasks

how to put into words what you see with your eyes

WHAT IT IS

The interface is the only thing in this book that you conceive **visually** and have to describe in text. Hence a particular difficulty: the picture is in your head, the AI gets words, and the discrepancy only shows up when you look at the result.

This article is about narrowing that gap.

WHAT TO TELL THE AI

Four parts, as everywhere

The same parts as in article 3.2, but with their own content:

Part	For an interface that means
Context	what data there is, where it comes from, what is already written
Goal	what the user sees and what they can do
Constraints	what not to touch, which components to reuse
Criterion	checkable states of the screen

Describe states, not appearance

The chapter's key technique. Any screen with data has at least four states, and by default an AI builds only one — the happy one.

BETTER

```
A subscriptions list screen. Four states:
```

```
1. Loading — a skeleton of three cards
2. Empty — the text "No subscriptions yet" and an "Add" button
3. Data — the list of cards
4. Error — the error text and a "Retry" button

All four must be implemented, not only the third.
```

```
DON'T DO THIS
Make a subscriptions list
```

The second request will give you only the third state. Loading, emptiness and the error turn up later — as bugs, for users.

Name the structure, not the pixels

```
DON'T DO THIS
A card 320 pixels wide, 18 of top margin, a 17px heading
```

```
BETTER
A card: the city's name, the temperature large underneath,
a "Remove" button at the bottom. Cards in a row, and in a
column on a narrow screen. Spacing from the existing
variables, don't introduce new values.
```

Pixels are magic numbers (article 8.3): fitted to one screen, they fall apart on another.

Give a signature

For React the signature is the props interface, and it works just as it does in articles 6.5 and 9.5:

```
PROMPT
A component to this interface:

interface SubsListProps {
  subs: City[] | null;        // null = still loading
  error: string | null;
  onRemove: (name: string) => void;
  onRetry: () => void;
}

All four states are visible from the types — implement them.
```

That is the most compact form of a task: the types describe the possibilities by themselves.

> **TIP**
> If the interface already exists somewhere — in a mockup, on another screen, in a screenshot — show it and say "make it like this". Describing in words what can be shown is extra work and an extra source of discrepancies.

Ask in parts

A whole screen in one request is five tasks in one (article 3.16). Better in steps:

1. the card component with prop types;
2. the list with loading and empty states;
3. error handling and retry;
4. responsiveness.

After each step, look at it. That is cheaper than taking apart a finished screen in which something is wrong.

WHAT TO LOOK FOR

Only the happy state is built. Easily checked: turn off the network and see what is on the screen.

Invented spacing and sizes. If the project has variables or a spacing scale, new values are a mismatch.

The component doesn't reuse the existing ones. An AI will happily write its own button instead of the one already in the project. That is reason 2 from article 3.1 — it simply hasn't seen it.

Invented wording. "Sorry, something went wrong" instead of text in the voice and style the rest of the interface uses.

Accessibility forgotten. The same `div`s instead of `button`s, no labels on fields (article 8.2). React doesn't fix that.

> **CHEAT SHEET**
> - Describe **states**, not appearance: loading, empty, data, error.
> - By default an AI will build only "data".
> - Structure instead of pixels; spacing from existing variables.
> - The props interface is the most compact form of a task.
> - If there is a mockup or a similar screen, show it instead of describing it.
> - Ask in parts and look after each one.

10.6 AI mistakes in React, and a checklist

what to check beyond "it opened and it works

WHAT IT IS

The chapter's closer, and the end of the whole frontend block. React adds its own mistakes to those from chapters 8 and 9 — and almost all of them are **invisible to a quick check**: the screen opened, the data showed up, the button presses.

TEN AI MISTAKES IN REACT

1. **No `key`, or `key={index}`** in a list. Deleting from the middle makes the data slide onto the neighbours (article 10.2).
2. **A `useEffect` with no dependency array** that changes state — an endless loop. Measured: 49 runs instead of one (article 10.4).
3. **An incomplete dependency array.** The effect works with a stale value.
4. **No cleanup.** A `setInterval`, a subscription or a `fetch` without cancellation outlives the component.
5. **`setState` in a row depending on the old value.** Three calls give +1 instead of +3 (article 10.3).
6. **State instead of a computation.** An extra `useState` + `useEffect` where one line would do.
7. **Only the happy state is built.** No loading, emptiness or error (article 10.5).
8. **`{items.length && ...}`** — on an empty list it prints "0".
9. **Props with no types, or with `any`.** Everything TypeScript gives you is lost (article 9.5).
10. **Secrets in frontend code.** Everything that made it into the build is visible to the user (article 10.1).

A REACT CODE CHECKLIST

Six steps on top of the frontend checklist (article 8.5).

1. Type checking

```
npx tsc --noEmit
```

Catches props: missing ones, wrong types, a handler with a different signature.

2. A linter with the hook rules

```
npx eslint .
```

The `react-hooks/exhaustive-deps` rule catches incomplete dependency arrays — the thing you almost can't see with your eyes.

3. The console

F12. React writes warnings there about `key`, about state after a component was removed, about wrong props. There should be no red and no yellow.

4. The network when the screen opens

The Network tab (article 8.4). How many requests go out on opening? One or two is normal. Dozens and a rising counter is exactly the loop from point 2.

5. The four states

- **Loading** — throttle the network in the developer tools.
- **Empty** — delete all the records.
- **Error** — turn off the network and reload.
- **Data** — the ordinary case.

The first three almost nobody checks, and that is exactly where the bugs live.

6. Fast repeated actions

Press the button five times in a row. Switch tabs quickly. This catches both the `setState` from point 5 and the race condition from point 4.

> **REMEMBER**
> The main difference between checking React and checking ordinary code: here it isn't enough to look at the result — you have to look at **every state** and at **fast actions**. The errors don't live in what you see straight away.

WHAT TO TELL THE AI

PROMPT

```
Check your React code against the list:
- is there a key, and is it not the index
- does every useEffect have a dependency array, and is it
  complete
- is there cleanup for subscriptions, timers and requests
- are there setState calls in a row depending on the old value
- are the loading, empty and error states implemented
- is there any in the props
```

```
Which of these are there? If none, say so.
```

PROMPT

```
This screen is built only for the "data arrived" case.
Add loading, an empty list, and an error with a "Retry"
button. Don't rewrite the existing markup.
```

CHEAT SHEET

- Ten mistakes: `key`, dependencies, cleanup, `setState` in a row, extra state, a single screen state, `length &&`, `any` in props, secrets in the build.
- Checklist: `tsc` · linter with the hook rules · console · network · four states · fast presses.
- The `exhaustive-deps` rule catches what the eye can't see.
- Dozens of requests when a screen opens means an endless loop in an effect.
- Check loading, emptiness and the error, not only the happy case.
- Press the button five times in a row — that catches two classes of error at once.

11 Flutter and Dart

Mobile applications. The one chapter in the book whose examples were not run — and the first article says so outright. So the emphasis here is on what doesn't go out of date: how a project is arranged, how to read a widget tree, and where an AI goes wrong.

11.1 The anatomy of a Flutter project

and an honest warning about this chapter

WHAT IT IS

Flutter is a way to write an application once and build it for Android, iOS, the web and the desktop. The language is Dart.

Before we start, one warning that applies to the whole chapter.

> **CAREFUL**
> **The examples in this chapter were not run.** In every other chapter of this book each listing was executed and each number measured. Here the code is shown to be read, not as a tested recipe: Flutter changes its API from version to version, and a listing that ran on one version may not build half a year later. So the chapter is written at the level of reading: how a project is arranged, how to read someone else's code, where an AI goes wrong. Check the exact syntax and method names against the documentation — including against me.

That is unpleasant, but more honest than pretending. It is also exactly the situation this whole book is about: **confident text is not the same as checked text** (article 1.6). Treat this chapter the way you would treat an AI's answer about a rare library — read it, but verify.

WHAT IT LOOKS LIKE IN CODE

```
weather_app/
├── lib/                    ← your code, and only it
│   ├── main.dart           entry point
│   ├── screens/            screens
│   ├── widgets/            reusable pieces of interface
│   ├── models/             data descriptions
│   └── services/           network, database, API work
├── test/                   tests
├── android/                Android platform files
├── ios/                    iOS platform files
├── web/                    files for the web build
├── pubspec.yaml            dependencies and assets
├── pubspec.lock            exact versions — commit this
└── .dart_tool/             housekeeping, don't touch
```

The structure is familiar: **`lib/` is what you read**, everything else is scaffolding. Exactly the same rule as `src/` in a React project (article 10.1).

The `android/` and `ios/` folders are real native projects. You look into them rarely: when you need permissions, icons, app signing. That is chapter 12's subject.

`pubspec.yaml`
The counterpart of `package.json`, but also a list of assets:

```yaml
name: weather_app
environment:
  sdk: ^3.5.0

dependencies:
  flutter:
    sdk: flutter
  http: ^1.2.0

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^4.0.0

flutter:
  uses-material-design: true
  assets:
    - assets/icons/
```

Two differences from other ecosystems worth knowing about:

Assets are declared explicitly. Putting an image in a folder isn't enough — it has to be written into `assets`. Forget, and the application

won't find the file at run time, and the error comes when the screen opens rather than at build time. An AI forgets this regularly.

Indentation is significant. YAML, like Python, is sensitive to spaces, and a wrong indent breaks the whole file.

The commands

```
flutter pub get          # install dependencies
flutter run              # run on a device or an emulator
flutter analyze          # static analysis — the counterpart of tsc and
mypy
flutter test             # tests
flutter build apk        # a build (chapter 12)
```

`flutter analyze` is the most important of these for working with an AI. It is the same line of defence as `mypy --strict` in Python (article 6.3) and `tsc --noEmit` in TypeScript (9.3): a check before the run. In Dart types are mandatory, so the analyzer catches a lot.

WHAT TO TELL THE AI

PROMPT

```
Show this project's lib/ structure: what's in each folder,
one line each. Don't show android/, ios/ or .dart_tool.
```

PROMPT

```
I've added an image to assets/icons/. What needs to go into
pubspec.yaml for it to work? Show the exact indentation.
```

PROMPT

```
My Flutter version is 3.38, Dart 3.10. Write code only for
those, and tell me if anything you propose appeared in a
later version.
```

That last request is especially apt here: Flutter changes its API between major versions more often than most frameworks, and there is more training data for the old versions (article 3.1, reason 3).

WHAT TO LOOK FOR

Code for old versions. The commonest sign is outdated widget and parameter names. If `flutter analyze` complains about unfamiliar arguments, the version is probably the reason.

An asset not declared in `pubspec.yaml`. The error appears when the screen opens, not at build time.

No `pubspec.lock` in the repository. The versions will drift apart (article 5.6).

Everything in `main.dart`. A thousand-line file instead of screens and widgets.

Edits in `android/` and `ios/` with no explanation. Those folders are partly generated; changes there are easily lost when the tooling is updated.

CHEAT SHEET

- The examples in this chapter were not run — check them against the documentation.
- Read only `lib/` `android/` and `ios/` are for builds and permissions.
- `pubspec.yaml` holds dependencies **and** the asset list; indentation matters.
- Putting an image in a folder isn't enough — it has to be declared in `assets`.
- `flutter analyze` is the check-before-the-run line, like `mypy` and `tsc`.
- Flutter often breaks compatibility between versions: name yours.

11.2 Dart in half an hour

what you need to know in order to read it

WHAT IT IS

Dart resembles Java, JavaScript and TypeScript at the same time. If you have read chapter 9, half of it is already familiar: types, `async/await`, classes.

This article covers what differs and what turns up in every file of a Flutter project.

WHAT IT LOOKS LIKE IN CODE

```
class City {
  final String name;
  final double? temp;

  const City({required this.name, this.temp});

  String get report => temp == null
      ? '$name: no data'
      : '$name: ${temp! > 0 ? '+' : ''}$temp°';
}

void main() {
  final cities = <City>[
    const City(name: 'Oslo', temp: 4),
    const City(name: 'Tromsø'),
  ];

  for (final c in cities) {
    print(c.report);
  }
}
```

What matters here

`final` instead of `const` and `let`. `final` means assign once. `const` means the value is known at compile time. In Flutter `const` turns up constantly: it lets widgets avoid being recreated on a redraw.

`double? temp` — a type with a question mark, the same as `number | null` in TypeScript. Dart is as strict about emptiness as TypeScript with `strict`: you can't reach for `temp` directly until you have checked.

`temp!` — the assertion "there is definitely something here". The direct analogue of `as` from article 9.5 and the same source of trouble: it is a promise to the compiler, not a check. Get it wrong and the application dies with `Null check operator used on a null value`.

`required` — a mandatory named parameter. Without it the argument can be left out.

`$name` and `${expression}` — interpolation into a string, like f-strings in Python.

`get report => ...` — a computed property: accessed like a field, without parentheses.

Named parameters

The main syntactic difference you have to get used to:

```
City(name: 'Oslo', temp: 4)          // named, in curly braces
```

In Flutter nearly every parameter is named, and that explains why the code looks so verbose:

```
Padding(
  padding: const EdgeInsets.all(16),
  child: Text('Hello'),
)
```

The upside is that while reading you can always see what each value means.

The rest, briefly

Written as	What it means
`late`	"there will be a value, but later" — a promise, checked at run time
`?.`	safe access, as in JS (article 9.1)
`??`	a default value
`..`	a cascade: several operations on one object
`=>`	the short form of a one-expression function
`List<City>`	a list of cities; `<>` as in generics (article 9.4)

`late` is worth a separate mention. It is a way of saying "I will assign this later". If you didn't assign it and then reach for it, you get a run-time error. An AI uses `late` to get around the initialization requirement — and that is the same gag as `as` and `!`.

> **REMEMBER**
> Three notations in Dart switch off the emptiness check: `!`, `late` and a type cast. Meet them in generated code and ask why a check wouldn't do. That is the direct analogue of `any`, `as` and `ts-ignore` from article 9.6.

WHAT TO TELL THE AI

> **PROMPT**
> Explain this class line by line: what final means, what required means, what the exclamation mark after temp does.

> **PROMPT**
> This uses `late` and `!`. Rewrite it without them, with proper null checks. If there's a place where they can't be avoided — explain why instead of rewriting it.

PROMPT

```
Write a data model for this API's response: here is a sample
JSON. Make fields that may be absent honestly nullable.
Add a fromJson factory with checks rather than casts.
```

WHAT TO LOOK FOR

`!` at every turn. The application will die wherever the data turned out not to be there.

`late` instead of proper initialization. The same problem, only later.

Code for old Flutter versions. Outdated widget and parameter names.

`dynamic`. The exact analogue of `any` from TypeScript: it switches checking off.

JSON parsing with no checks. `json['main']['temp'] as double` fails if the server answered with an error — exactly as in articles 2.12 and 9.5.

CHEAT SHEET

- `final` means assign once, `const` means known at compile time.
- `double?` may be empty; Dart won't let you reach for it unchecked.
- `!`, `late` and a cast switch off the emptiness check.
- Parameters in Flutter are nearly always named: `City(name: ..., temp: ...)`.
- `$name` and `${expression}` interpolate into a string.
- `dynamic` is `any`, with all that follows.

11.3 Widgets

the tree, composition, and why a margin is a widget too

WHAT IT IS

In Flutter **everything is a widget**. A button, text, a margin, alignment, the screen itself. An interface is assembled not out of properties, as in CSS, but out of widgets nested inside one another.

That is the main difference from everything earlier in the book, and the main source of bafflement when reading: the code looks like a deep staircase of brackets.

WHAT IT LOOKS LIKE IN CODE

```
Scaffold(
  appBar: AppBar(title: const Text('Weather')),
  body: Padding(
    padding: const EdgeInsets.all(16),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text(city.name, style: Theme.of(context).textTheme.titleLarge),
        const SizedBox(height: 8),
        Text(city.temp == null ? 'no data' : '${city.temp}°'),
        const Spacer(),
        ElevatedButton(
          onPressed: () => onRemove(city.name),
          child: const Text('Remove'),
        ),
      ],
    ),
  ),
)
```

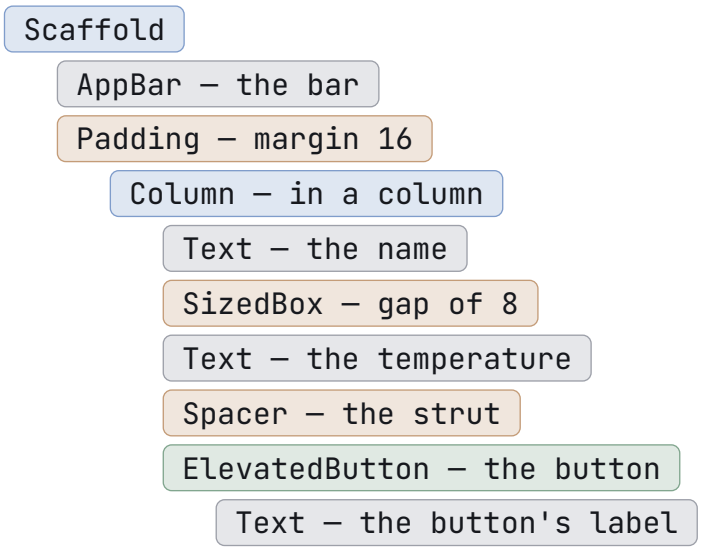


Fig. 11.1 · The same thing as a tree. The margin and the gap are nodes just like the text and the button.

How to read it

Read by nesting, not by lines. The indentation in the code repeats the tree's structure, so the most useful thing while reading is to watch the indentation level.

`child` is one child, `children` is a list. Those two words let you reconstruct the whole structure: where it branches, where it is a chain.

Margins and gaps are widgets too. That is the main difference from CSS:

What you need	Widget
a margin around	`Padding`
a gap between	`SizedBox(height: 8)`
take all the free space	`Expanded`, `Spacer`
align	`Align`, `Center`
in a column / in a row	`Column`, `Row`
stacked on top of each other	`Stack`

Which is why the tree is so deep: what would be two lines of properties in CSS is two levels of nesting here.

`const` in front of a widget. Not a formality: such a widget is created once and isn't recreated on a redraw. An AI often leaves it out, and the analyzer complains about that.

`Theme.of(context)` — take a value from the application's theme: a colour, a font, a size. The counterpart of CSS variables (article 8.3): better than writing sizes in as numbers.

> **TIP**
> When the tree becomes unreadable, that is a signal to pull a piece out into its own widget, not a reason to put up with it. The request "pull the contents of Column out into a separate CityInfo widget" takes one line and gives the code its readability back.

Row, Column and overflow

The most frequent visual breakage in Flutter: the content doesn't fit, and a yellow-and-black stripe with an overflow message appears on the screen.

The cause is almost always the same: text longer than the screen was put into a `Row`. The cure is wrapping it in `Expanded` or `Flexible` — they say "take the space that's there, and no more".

WHAT TO TELL THE AI

PROMPT

```
Explain this widget tree: what each part is responsible for.
One line per nesting level, no code.
```

PROMPT

```
Pull the contents of Column out into a separate CityInfo
widget in lib/widgets/. Pass the parameters through the
constructor. Don't change the behaviour.
```

PROMPT

```
With a long city name this Row overflows. Fix it with
Expanded, and explain why that rather than a fixed width.
```

WHAT TO LOOK FOR

A ten-level tree in one file. It breaks into widgets.

No `const` where the widget doesn't depend on data. Needless recreation on every redraw.

Sizes as numbers instead of the theme. `fontSize: 17` instead of `Theme.of(context)` — the same magic numbers as in article 8.3.

A `Container` for one thing. If you only need a margin, that's `Padding` only alignment, `Align`. `Container` can do everything at once and therefore hides the intent.

Logic inside `build`. Network requests and computations in the build method: it is called on every redraw, which is to say often.

Fixed sizes for one screen. Phones differ; `MediaQuery` and `Expanded` exist precisely for that.

CHEAT SHEET

- In Flutter everything is a widget, margins and gaps included.
- `child` is one, `children` is a list; the tree follows from them.
- Read by nesting, not by lines.
- `const` in front of a widget saves a recreation on redraw.
- Sizes and colours come from the theme, not from numbers.
- A yellow-and-black stripe means overflow; the cure is `Expanded`.

11.4 State

stateless, stateful, and which to choose

WHAT IT IS

Widgets come in two kinds: those that simply draw the data handed to them, and those that hold something of their own and redraw when it changes.

The idea is the same as props and state in React (article 10.3), and the rules for deciding are the same. Only the syntax differs — and in Flutter it is wordier.

WHAT IT LOOKS LIKE IN CODE

Stateless — no data of its own

```
class CityCard extends StatelessWidget {
  final String name;
  final double? temp;
  final void Function(String) onRemove;

  const CityCard({
    super.key,
    required this.name,
    this.temp,
    required this.onRemove,
  });

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Column(
        children: [
          Text(name),
          Text(temp == null ? 'no data' : '$temp°'),
          TextButton(
            onPressed: () => onRemove(name),
            child: const Text('Remove'),
          ),
        ],
      ),
    );
  }
}
```

Everything comes from outside through the constructor — the direct analogue of props. The `final` on the fields is no accident: they can't be changed, just like props in React.

`super.key` is the same `key` as in React (article 10.2), and it is there for the same reason: so that when the list changes Flutter knows which widget is which.

Stateful — it has its own

```
class Counter extends StatefulWidget {
  const Counter({super.key});

  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return TextButton(
      onPressed: _increment,
      child: Text('pressed: $_count'),
    );
  }
}
```

Two classes instead of one is a Flutter peculiarity you have to get used to: the widget itself and its state live separately.

`setState` is the only way to announce a change. Assigning `_count++` without the `setState` wrapper changes the value, but **the screen doesn't redraw**. That is exactly the same mistake as assignment instead of `setX` in React.

A leading underscore in Dart means "this file only". Which is why the state is usually `_CounterState` and the fields `_count`.

The lifecycle

```
@override
void initState() {
  super.initState();
  _loadData();        // runs once, when it appears
}

@override
void dispose() {
  _controller.dispose();   // cleanup: controllers, subscriptions,
timers
  super.dispose();
}
```

Method	When	The React equivalent
`initState`	once, on appearance	`useEffect(..., [])`
`build`	on every redraw	the component's body
`dispose`	on removal	the function returned from `useEffect`

`dispose` is mandatory for everything that has to be closed: text field controllers, scroll controllers, animations, subscriptions, timers. A forgotten `dispose` is a leak, and the analyzer won't always warn about it.

> **REMEMBER**
> The rule for choosing is the same as in React: state lives in the nearest common parent of the widgets that need it. If two sibling screens need the data, it belongs in neither of them.

When there is a lot of state

Passing data down five levels of constructors is awkward, and there are state-management libraries for that: `provider`, `riverpod`, `bloc` and others.

Choosing between them is an architectural decision, not a matter of taste: it determines how the whole project is arranged. It is worth making yourself rather than leaving to an AI (article 1.1), and you must name your choice in your requests — otherwise you will get code for whichever library had more of the training data.

WHAT TO TELL THE AI

> **PROMPT**
> `The project uses provider, version 6. State is held in a ChangeNotifier. Don't add new state-management libraries.`

PROMPT
```
This widget is stateful but has no state of its own —
everything comes through the constructor. Make it stateless.
```

PROMPT
```
Check whether every controller in this file has a dispose.
A list of what isn't being closed.
```

WHAT TO LOOK FOR

Stateful with no state. An extra class and extra code. The sign: the `State` has no fields, only a `build`.

Assignment without `setState`. The value changes, the screen doesn't.

`setState` with heavy work inside. Only field changes go inside; requests and computations go before it.

No `dispose`. Controllers and subscriptions outlive the widget.

A network request in `build`. `build` is called often; the request belongs in `initState` or in the data layer. The direct analogue of the mistake in article 10.4.

The state library chosen for you. `bloc` appeared in the answer although the project uses `provider` — and the project ends up with two systems at once.

CHEAT SHEET

- Stateless — everything from outside through the constructor, the equivalent of props.
- Stateful — two classes: the widget and its state.
- Change fields only inside `setState`, or the screen won't redraw.
- `initState` once, `build` often, `dispose` for cleanup.
- Controllers and subscriptions must be closed in `dispose`.
- Choose the state-management library yourself and name it in the request.

11.5 Stating mobile interface tasks

what to name besides the appearance

WHAT IT IS

The techniques from article 10.5 work here too: describe states rather than pixels, give a signature, ask in parts.

But a mobile application has things a website doesn't — and if you don't mention them, the AI won't ask.

WHAT TO TELL THE AI

Six things specific to mobile

1. Versions. Flutter breaks compatibility more often than most frameworks:

```
PROMPT
Flutter 3.38, Dart 3.10. Code for those versions only.
If you propose something from a later one, say so.
```

2. The state library. Name it and the project won't grow a second one (article 11.4).

3. Screen size and orientation. Phones differ, and "it looks fine" on one means nothing:

```
PROMPT
Check the layout on a narrow screen (360 logical pixels)
and in landscape orientation. Don't use fixed heights.
```

4. What happens with no network. A mobile app loses its network constantly — that is the norm, not the exception:

```
PROMPT
The network can drop at any moment. What do we show:
while loading, on a network error, on an empty response,
on successful data. All four are needed.
```

5. Permissions. The camera, location, notifications, files — all of them require asking the user, and a refusal has to be handled:

```
PROMPT
I need location. Handle three cases: granted, denied,
denied permanently. In the last case, an offer to open
the settings.
```

The third case is always forgotten, and it is the nastiest: the application asks endlessly for a permission the system will never show again.

6. Platform differences. Android and iOS behave differently: the back button, the safe areas at the screen's edges, the look of dialogs.

```
PROMPT
We're building for Android and iOS. Flag the places where
the behaviour will differ.
```

What not to do

Don't ask for "make it look nice". That is the anti-pattern from article 3.16, and in Flutter it is especially destructive: you will get a ten-level tree with eyeballed sizes.

Don't describe things in pixels. `Theme.of(context)` and the ready-made spacing widgets exist precisely for that (article 11.3).

Don't ask for a whole screen. Four steps are cheaper: data model → card widget → list screen with its states → error handling.

> **TIP**
> The key technique for a mobile interface is a screenshot. If you have a mockup, a similar screen, or even a photo of another application's screen, show it and say "make it structurally similar". Describing in words what can be shown is an extra source of discrepancies.

A check worth ordering straight away

PROMPT

```
Once you've written it: list what I should check by hand on
a device. Include the narrow screen, no network, and a
denied permission.
```

That list is a draft of the checklist from article 11.6, written for this particular task.

WHAT TO LOOK FOR

Only the happy case is built. As in React: "the data arrived" is there, loading, errors and emptiness are not.

No handling of a denied permission. The application hangs on the request.

Fixed sizes. It will fall apart on a different phone.

Strings not pulled out. Labels written straight into the widgets — add a second language and everything has to be rewritten.

Platform differences ignored. It works on Android and behaves strangely on iOS.

CHEAT SHEET

- Name your Flutter and Dart versions — here that matters especially.
- Name the state library, or you will get a second one.
- Ask for the narrow screen and landscape orientation to be checked.
- The network drops constantly: all four screen states are needed.
- Permissions are three cases, and "denied permanently" is always forgotten.
- If there is a mockup or a similar screen, show it instead of describing it.

11.6 AI mistakes in Flutter, and a checklist

and how to check what I couldn't

WHAT IT IS

The chapter's closer. And here, something no other chapter has: **a list of what you have to check yourself**, because I couldn't (article 11.1).

AI MISTAKES IN FLUTTER

1. **Code for an old version.** This stack's main problem. Outdated widget and parameter names; `flutter analyze` usually complains.
2. **`!`, `late`, `dynamic`.** Three ways to switch off the emptiness check — the direct analogue of `any`, `as` and `ts-ignore` from article 9.6.
3. **Assignment without `setState`.** The value changes, the screen doesn't.
4. **No `dispose`.** Controllers and subscriptions outlive the widget.
5. **A network request in `build`.** `build` is called often.
6. **Stateful with no state.** An extra class.
7. **No `const`** where the widget doesn't depend on data. Needless recreation.
8. **An asset not declared in `pubspec.yaml`.** The error comes when the screen opens, not at build time.
9. **Only the happy case is built.** No loading, error or emptiness (article 11.5).
10. **A denied permission isn't handled.** "Denied permanently" especially.
11. **A second state library added.** Because the request didn't name the first one.
12. **Fixed sizes.** Overflow on a narrow screen.

A FLUTTER CODE CHECKLIST

1. The analyzer

```
flutter analyze
```

The first and most important step — here it stands in for `mypy` and `tsc`. It catches outdated APIs, forgotten `const`s, unused things, and some of the emptiness errors.

2. Run it

```
flutter run
```

Hot reload shows an edit at once, but **not every change is picked up**: edits in `initState` and in declarations often need a full restart. If a change "didn't apply", restart completely before looking for a bug in the code.

3. A narrow screen and long text
Run it on a small device or narrow the window. Type in a long name. The yellow-and-black overflow stripe is the commonest finding.

4. The four states
Loading, empty, error, data. For the error, turn off the network on the device.

5. Permissions
Deny the permission. Then deny it permanently. The application has to keep working and explain what to do.

6. Both platforms
If you build for Android and iOS, check on both. The back button, the safe areas and the dialogs behave differently.

HOW TO CHECK THIS CHAPTER

The examples here weren't run, so — how to satisfy yourself that what is written is right:

PROMPT
```
Here is an extract from a Flutter handbook: <extract>.
I'm on Flutter 3.38. Is this current for my version?
If anything is outdated or inaccurate, say exactly what,
with a link to the documentation.
```

And the most reliable way is simply to run it. Create an empty project and check whatever looked doubtful:

```
flutter create test_app && cd test_app && flutter run
```

REMEMBER
That is the book's central idea applied to the book itself: **confident text is not the same as checked text**. The chapter is written carefully but it was not run — which means the same rules apply to it as to an AI's answer about a rare library. Read it, use it, verify it.

WHAT TO TELL THE AI

PROMPT
```
Check this Flutter code against the list:
- is the API current for Flutter 3.38
```

```
- are there !, late or dynamic, and why
- does every state change go through setState
- do the controllers and subscriptions have a dispose
- are there requests in build
- is there a const where the widget is static
- are loading, error and emptiness implemented

Which of these are there? If none, say so.
```

CHEAT SHEET

- This stack's main mistake is code for an old version. Name yours.
- `!`, `late`, `dynamic` are gags, not solutions.
- Changes only through `setState` close controllers in `dispose`.
- `flutter analyze` is the first checking step, standing in for `mypy` and `tsc`.
- "It didn't apply" — a full restart first, then look for the bug.
- Check: a narrow screen, the four states, a denied permission, both platforms.

12 Building and shipping

The last step, and the one that wrecks the most projects. The code works on your machine and nowhere else. This is about how a build differs from a run, why a green build doesn't mean a working program, and what to ask an AI before you ship.

12.1 What a build is and why it breaks

five reasons behind "it works on my machine

WHAT IT IS

A build is the turning of your sources into something that runs for other people. While you develop, a developer tool runs your program: it reads your files as they are, substitutes settings from your machine, reloads what changed on the fly. That is convenient and nothing like what the user will get.

During a build something else happens: the code is checked, rewritten, glued into a few files, setting values are substituted in, the surplus is thrown away. The result is an **artifact** — one file or folder you can hand over: an `.apk` for Android, a `dist/` folder for a website, a `.whl` or a container for the server side.

The gap between "it ran on my machine" and "the artifact works" is the most frequent and most galling breakage in a project, because it happens at the end, when everything already looked finished.

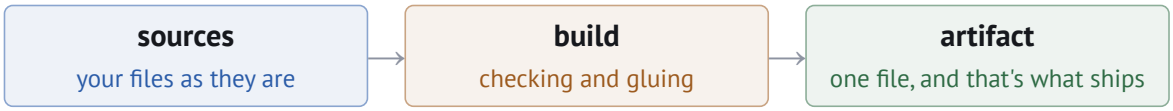


Fig. 12.1 · Three states of one project. This chapter's errors live on the arrows, not in the code.

> **REMEMBER**
> Running during development and building are two different programs reading the same files differently. "It works in development" doesn't mean "it builds", and "it builds" doesn't mean "it works".

WHAT IT LOOKS LIKE IN CODE

Below are five reasons for breakage, and all five were checked on live projects: the real command output is shown.

1. The build passed and the code is broken

Here is a file with two blatant type errors — a string where a number belongs and a number where a string belongs:

```
import { CityCard } from './components/CityCard'

export default function App() {
  const temp: number = "hot"
  return <CityCard name={temp} />
}
```

Run the build directly:

```
$ npx vite build
vite v7.3.7 building client environment for production...
✓ 28 modules transformed.
dist/index.html                   0.20 kB │ gzip:  0.17 kB
dist/assets/index-D40P3igU.js  222.62 kB │ gzip: 69.34 kB
✓ built in 1.69s
```

It built. In under two seconds, with a tick. Now the type check, on the same file:

```
$ npx tsc -b
src/App.tsx(4,9): error TS2322: Type 'string' is not assignable to type
'number'.
src/App.tsx(5,20): error TS2322: Type 'number' is not assignable to type
'string'.
```

The reason is that the bundler doesn't check types — it **throws them away**. Types are for the editor and the checker; they aren't in the built file (article 9.1). Which is why in a normal project the build command is two commands:

```
"build": "tsc -b && vite build"
```

Check first, then glue. And this is exactly where the mistake appears that both people and AIs make: the build fails, and it is "fixed" by taking `tsc -b` out of the command. The build turns green, the check is switched off.

> **CAREFUL**
> If an AI proposes editing the build command rather than the code, stop and read what is being removed. Removing the type check, disabling a linter rule, adding `--force` or `--legacy-peer-deps` — that isn't a fix, it is silencing the signal (article 2.14).

2. Settings are baked in

Environment variables aren't read while the program runs — they are **substituted into the text** of the artifact. The settings file:

```
VITE_API_URL=https://api.example.com
API_KEY=secret-abc-123
```

The code that reads them:

```
const API = import.meta.env.VITE_API_URL
const KEY = import.meta.env.API_KEY
```

And here is what ended up in the built file — a real fragment from `dist/`:

```
const mh="https://api.example.com",dh=void 0;
```

One line, two different disasters.

The first: the address got into the artifact as text. The built file is served to anyone's browser, which means everything in it is public. Had there been a key for a paid API in place of the address, it would be sitting right there in plain text.

The second: `API_KEY` became `void 0`, that is, "nothing". The bundler substitutes only variables with the right prefix (`VITE_` in this example); the rest silently become empty. No error, no warning: the application builds, starts, and behaves oddly — knocking on `undefined/api/cities`, for instance.

You can check this by hand, and it is worth doing at least once in your life: build the project and search the built file for your own secret.

```
$ grep -c "secret-abc-123" dist/assets/*.js
0
$ grep -c "api.example.com" dist/assets/*.js
1
```

The rule is simple: **what is meant to stay secret doesn't go into code that travels to the user's device.** Keys live on the server, and the application talks to its own server, which talks to the paid API.

3. A path that depends on where you started from

An innocent line:

```
with open("weather/cities.json", encoding="utf-8") as f:
```

From the project root it works:

```
$ python3 -m weather.main
{'Oslo': 4, 'Tromso': -3}
```

From any other folder it doesn't:

```
  File ".../weather/main.py", line 5, in load_cities
    with open("weather/cities.json", encoding="utf-8") as f:
         ~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'weather/
cities.json'
```

A relative path is counted not from the file with the code but **from the folder the program was started in**. During development that is always the project root, so the mistake is invisible. On a server the program is started by a service, and its folder is its own.

The right way is to count from the file's own location:

```
from pathlib import Path

DATA = Path(__file__).parent / "cities.json"

def load_cities():
    with DATA.open(encoding="utf-8") as f:
        return json.load(f)
```

After that edit, running from `/tmp` gives the same thing as running from the project root. Checked: `{'Oslo': 4, 'Tromso': -3}`.

4. A file that didn't make it into the artifact

The same project, paths now fixed, built into a package:

```
$ python3 -m build --wheel
Successfully built weather-0.1.0-py3-none-any.whl
```

No error, no warning. Look at what's inside:

```
weather/__init__.py
weather/main.py
weather-0.1.0.dist-info/METADATA
weather-0.1.0.dist-info/WHEEL
weather-0.1.0.dist-info/RECORD
```

`cities.json` isn't in there. Install the package and run it:

```
FileNotFoundError: [Errno 2] No such file or directory:
'/tmp/venv-weather/lib/python3.13/site-packages/weather/cities.json'
```

A build tool builds code, and everything else — images, fonts, dictionaries, email templates, data files — **has to be declared separately**. Three lines in the project's settings:

```
[tool.setuptools.package-data]
weather = ["*.json"]
```

And the built package now has what it needs:

```
adding 'weather/__init__.py'
adding 'weather/cities.json'
adding 'weather/main.py'
```

This is a disease common to every stack: in Flutter assets are declared in `pubspec.yaml` (article 11.1), on the frontend images must be either imported or put in a special folder, into a container files have to be copied with an explicit command. Forget to declare it and the artifact is built without the file, and it fails not at build time but when the screen that needs that file is opened.

> **TIP**
> Running your own package from the project root isn't enough: that way the project folder lands in the search path anyway, and Python finds the file in the sources rather than in the installed package. That is exactly why mistake number four goes unnoticed for so long. Check from a different folder.

5. Dependency versions drift

The project's settings said this:

```
"react": "^19.1.1",
"vite": "^7.1.7"
```

And this is what got installed:

```
$ npm ls react vite --depth=0
+-- react@19.3.1
`-- vite@7.3.7
```

The `^` sign means "this version or newer within compatibility". That is, the same project installed by you today and on a server a month from now will get **different versions** of the libraries. Usually nothing happens. Sometimes something does, and working out what changed starts from scratch.

With Python it is the same, and vividly so: one `requests` line in the dependency list drags in five packages, each with its own version:

```
certifi==2026.7.22
charset-normalizer==3.5.2
idna==3.20
requests==2.34.2
urllib3==2.8.0
```

Four of the five you didn't choose. They are the source of the surprises.

The cure is a **file with exact versions**, created automatically on install: `package-lock.json`, `pubspec.lock`, `poetry.lock`, a `requirements.txt`

produced by `pip freeze`. It records exactly what you had when everything worked:

```
node_modules/react -> 19.3.1
node_modules/vite  -> 7.3.7
```

That file must be **committed to the repository**, and dependencies installed with a command that respects it: `npm ci` rather than `npm install`. Otherwise it is pointless.

WHAT TO TELL THE AI

PROMPT

```
The project builds for me but fails after installation.
Don't propose fixes at random. First list how running in
development differs from the built artifact in this project:
file paths, environment variables, assets, dependency
versions. One point per line.
```

A request like that gives you a list of places to look instead of a random edit.

PROMPT

```
Here is my build command and my environment settings file.
List which values will end up in the artifact in plain text
and be visible to the user. Separately: what will become
undefined, and why.
```

PROMPT

```
List every file the project reads that isn't code: data,
images, fonts, templates. For each one say whether it's
declared in the build settings. If it isn't, write the line
that needs adding.
```

PROMPT

```
The build fails with this error: <text>.
I want two options: a code fix and a build-settings fix.
For each, say what will stop being checked. I'll choose.
```

That last technique is worth remembering. Any “build fix” should come with its price: without that question, switching a check off looks like a solution.

WHAT TO LOOK FOR

The check vanished from the build command. It was `tsc -b && vite build`, it is now `vite build`. The build is green, the types aren’t checked.

Suppression flags appeared. `--force`, `--legacy-peer-deps`, `--no-verify`, `--ignore-scripts`, `|| true` at the end of a command. Each of them switches something off, and that something has to be said out loud.

A secret in the application's code. An API key, a password, a token in a file that is built and travels to the device. Checked by searching the built file.

Relative paths to data files. `open("data/x.json")` instead of a path from `__file__`.

Undeclared assets. An image, a font or a data file is in the project folder but isn't mentioned in the build settings.

The exact-versions file in `.gitignore`. The classic "optimization": `package-lock.json` looks like junk because it is large and keeps changing. Without it there is no reproducible build.

Building on your machine as the only way. As long as the project builds only for you and only by hand, nobody — you in six months included — will be able to build it.

CHEAT SHEET

- A build isn't a run: different code, different settings, different files.
- A green build doesn't mean working code: the bundler throws types away rather than checking them.
- Editing the build command instead of the code is nearly always switching a check off.
- Environment settings are baked into the artifact: secrets don't belong there.
- A variable without the right prefix silently becomes `undefined`.
- A relative path is counted from the launch folder, not from the code file.
- Everything that isn't code has to be declared in the build settings.
- Check the installed package from a different folder, or you'll find the sources.
- Commit the exact-versions file and install with `npm ci`, not `install`.

12.2 Android and iOS

versions, signing, permissions, identifier

WHAT IT IS

Besides its code, a mobile application has a **passport**: who released it, which version it is, what it is allowed to do and what the system

calls it. Those four things live not in the code but in the settings of the platform folders — and they are exactly why a build that ran beautifully on your own phone isn't accepted by the store.

An AI helps less here than with code: the requirements change every year while the training data stays the same. The concepts themselves, though, don't change for decades — and those are the ones worth knowing.

> **CAREFUL**
> Like chapter 11, this article is written at the level of reading: the settings fragments here are illustrative rather than copied from a particular project. Store requirements change every year. File names and the meaning of the fields are stable; check the exact syntax against your version's documentation.

WHAT IT LOOKS LIKE IN CODE

1. The version is two different numbers

```
version: 1.2.0+34
```

One line in `pubspec.yaml`, and the build turns it into two values:

Part	Android	iOS	Who sees it
`1.2.0`	versionName	Version	the user, in the store
`34`	versionCode	Build	the store only

The first is for people, shown in the update's description. **The second is for the store**, and it has to grow with every upload. If it didn't, the upload is rejected with a message that this version already exists.

That is the most frequent and most foolish waste of time when shipping: the build takes twenty minutes and the rejection comes because of a number nobody incremented.

2. The identifier is forever

```
Android:  com.lunario.weather      (applicationId)
iOS:      com.lunario.weather      (bundle identifier)
```

This is the application's internal name in the system and in the store. There is one rule and it is hard: **after the first release it cannot be changed**. You can only publish a different application and ask users to reinstall — losing every review, rating and install.

Project templates put `com.example.<name>` here. You can't publish with `com.example`: the stores don't accept such identifiers. Change it before the first release, and preferably on day one.

3. Permissions are declared in two places, differently

Android — a list in the manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/
>
<uses-permission android:name="android.permission.INTERNET"/>
```

iOS — not a list but **explanations for the user**:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>Needed to show the weather in your city.</string>
```

The difference is fundamental. On Android you declare what the application needs. On iOS you are obliged to write **why** — in text the user will see in the system dialog. No text and the application won't pass review, and on a device it will crash at the moment of the request.

And one more thing people often misunderstand: a declaration in the manifest isn't a permission. The permission is granted by the user, while the application is running, and they can refuse (article 11.5). The declaration only gives you the right to ask.

4. Signing

The artifact that goes to the store is signed with a cryptographic key. The signature answers one question: **is this an update to the same application, or a different one?**

Platform	What signs it	Where it is kept
Android	a key in a keystore file	with you, not in the repository
iOS	a developer certificate and a profile	in your Apple account

There is one thing you need to know about the Android key, and it is unpleasant: **lose the key and you can't release an update.** Ever. The application stays in the store and cannot be updated — only republished under a new identifier. The store's signing service partly saves the situation, but it has to be switched on in advance, before the first release.

Hence two rules:

- **The key and its passwords do not go into the repository.** The keystore file, the passwords and the paths to them go into a file that is in `.gitignore`. That is exactly the conversation about `.env` from article 5.7, only with higher stakes.
- **A backup of the key in two places,** and neither of them your laptop.

The debug signature the tools use for development builds is no good for the store. The sign of trouble: the artifact is called `app-debug.apk`, or the build log shows `signingConfig debug`.

> **REMEMBER**
> Four things have to be decided **before** the first release; afterwards it is too late: the application identifier, the signing scheme and where the key is kept, the name in the store, and the set of permissions. Everything else — code, design, prices, description — can be changed by updates as often as you like.

What else will cut users off

```
minSdk 26      ← below this Android version the app won't install
```

The minimum system version is a quiet decision with loud consequences. Raised it to make a library work and you cut off some of your users. That is a normal trade, but it has to be a conscious one: an AI raises the minimum version in passing, with one line in the settings, and doesn't mention it in the explanation.

WHAT TO TELL THE AI

The key technique for this article is to **demand an account of the consequences, not edits**. Platform settings change rarely and cost a lot.

> **PROMPT**
> ```
> Propose what to change to make this library work.
> For each edit in android/ and ios/, write separately what it
> changes for the user: minimum system version, new
> permissions, app size. Flag edits touching minSdk
> explicitly.
> ```

> **PROMPT**
> ```
> Here is my application's permission list: <list>.
> The application only does <description>.
> Which permissions are unnecessary? For each one that is
> needed, write the iOS explanation text — one sentence.
> ```

That second request solves two problems at once: it removes the surplus permissions that get releases rejected, and it writes the very explanation texts people forget.

PROMPT

```
I'm preparing an update. What has to be raised in the
version, and where exactly is it recorded in the project?
Show the line before and after.
```

PROMPT

```
Check whether signing files and passwords have got into the
repository: keys, keystores, profiles, properties files.
A list of what you find and the lines for .gitignore.
```

WHAT TO LOOK FOR

Permissions "just in case". An application with a list of cities has acquired the camera and contacts. An AI adds permissions when a library could theoretically use them. Stores reject you for that, and users uninstall.

No explanation text for iOS. The permission is declared, the explanation isn't: rejected at review.

The version wasn't raised. Upload rejected, twenty minutes of building wasted.

The identifier is still `com.example`. The template wasn't edited.

A debug signature. `app-debug.apk` instead of a signed artifact.

Signing files in the repository. The keystore, passwords, profiles — all of it goes into the git history, and from there it isn't simply removed.

`minSdk` raised with no explanation. One line, and some of your users can no longer install the application.

A permission declared but refusal unhandled. The most frequent code mistake on this subject — the same one as in article 11.5.

CHEAT SHEET

- `1.2.0+34`: the first is for people, the second is for the store and has to grow.
- The application identifier can't be changed after release — decide at once.
- Android declares permissions, iOS demands an explanation — in text.
- A manifest declaration ≠ a permission: the user grants it and can refuse.
- Lose the signing key and there are no more updates. A copy in two places.
- Keys and passwords go in `.gitignore`, not in the repository.
- `minSdk` is a decision about who won't be able to install the app.
- Ask an AI not for a settings edit but for an edit with its price to the user.

12.3 Shipping and updates

why the old version lives longer than you think

WHAT IT IS

Shipping isn't the finish line but the start of a condition in which **your program exists in several versions at once**. The server updated, the applications on people's phones didn't. The website updated, the browser is holding the old file. The new version is in the store and half the people are on last year's.

So the main question with every update isn't "how do I put it out" but **"what happens to the people still on the old one"**. Neither an AI nor the publishing tools ask that question — only you do.

WHAT IT LOOKS LIKE IN CODE

Three ways of shipping, three different speeds

What we ship	How it reaches people	Can it be rolled back
a website	instantly, at the next opening	yes, instantly
a server, an API	instantly, for everyone at once	yes, if prepared for
an app in a store	after review, when people choose to	no

The difference in the last column is the important one. A website rolls back in a minute. An application in a store **cannot be rolled back**: you can only release the next version and wait for people to install it. That is why mobile builds are checked more carefully and rolled out to a fraction of users rather than everyone at once.

A website: the cache, and how the build gets round it

Build the project twice in a row without changing a line:

```
$ npx vite build
dist/assets/index-CxiKQytF.js  222.62 kB | gzip: 69.34 kB
$ npx vite build
dist/assets/index-CxiKQytF.js  222.62 kB | gzip: 69.34 kB
```

The file name is the same. Now change one word in the text and build again:

```
$ npx vite build
dist/assets/index-Cfu8g7G7.js  222.62 kB | gzip: 69.34 kB
```

The name changed. Those letters in the name are a fingerprint of the contents: different contents, different name. That is done for exactly one purpose: so that a browser holding files of its own doesn't hand the user old code. The old file in the cache bothers nobody — nobody will ever come for it again.

But one file in this scheme doesn't change its name:

```
dist/index.html                  0.20 kB
dist/assets/index-Cfu8g7G7.js  222.62 kB
```

`index.html` is the file that says which of the hashed files to load. **It must not be cached.** If the hosting caches `index.html` for a day, the user will open the old page for a day, pointing at the old code, and no amount of publishing will fix it.

Hence the hosting rule worth knowing by heart: **files with a hash in the name — cache for a long time; `index.html` — don't cache.** It is the one setting worth checking by hand the first time you publish a site.

> **TIP**
> "I updated the site but I'm still getting the old one" is the cache nine times out of ten, not a failed deploy. Check by opening it in a private window: if it's new there, the deploy worked and the cache is to blame.

A server: the update hits people you weren't thinking about

A server updates instantly and for everyone. And here is where the thing nobody thinks about surfaces: **old applications on phones have no "update now" button.**

Rename a field in the API response and applications released six months ago stop showing the weather. Make a field required in a request and old versions stop sending data. Formally you changed

your server; in reality you broke programs that are already on people's devices.

The rule that follows:

- **adding fields is fine**, old clients simply won't notice them;
- **removing and renaming is not**, as long as anyone using them is still out there;
- if the change is unavoidable, a `/api/v2/` appears alongside while `/api/v1/` keeps working (article 7.3).

The second thing that breaks on a server is **the database**. Updating the database and updating the code don't happen at the same moment, and the order matters: first the new field is added, then the code that uses it is deployed. The other way round gives you a few minutes in which the code looks for something that isn't there yet.

And third: rolling back code is one command, **rolling back a database change is not**. A deleted column doesn't come back with the rollback. So database changes that delete something are deployed separately and later, once it is clear the new version is alive.

The store: what happens after the upload

```
uploaded  →  review  →  available to some people  →  available to
everyone
```

Review takes from hours to days and can end in rejection. The most frequent reasons for rejection aren't technical: permissions without an explanation (article 12.2), a demo that doesn't work for the reviewer, no privacy policy when the application collects anything at all.

A staged rollout — giving the new version to a small share of users first — is the only defence against the fact that a release can't be rolled back. If crashes start pouring in during the first day, the rollout is stopped and most people simply stay on the old, working version.

> **CAREFUL**
> Don't ship on a Friday. That isn't a joke or a superstition: a mobile app can't be rolled back, store review takes hours, and you are the one who will be fixing it. Release when you have a day to watch the result.

WHAT TO TELL THE AI

Shipping is the area where what is most useful from an AI is **lists of checks, not actions**.

PROMPT

```
I'm changing an API response: the "temp" field becomes
"temperature". My mobile apps in the store don't update
immediately. What breaks for people on the old version?
Propose a way to make this change without breaking them.
```

PROMPT

```
I'm adding a required field to the database. Write the order
of operations: what to deploy first, what second, and what
happens if they're deployed the other way round.
Separately: which part of this can't be rolled back.
```

That second request is worth asking every time the database changes. The answer to "what can't be rolled back" is precisely the list of things to do carefully.

PROMPT

```
Here are my hosting's caching settings: <settings>.
The build files have a hash in the name, index.html doesn't.
What's configured wrongly and what will it lead to?
```

PROMPT

```
Draw up a shipping checklist for my project: <description>.
Split it into "before shipping", "right after" and "a day
later". Only what applies to my stack.
```

WHAT TO LOOK FOR

An API change with no thought for old clients. A field renamed, no version added. It works for you and is broken for everyone who hasn't updated.

A deletion in the database deployed together with the code. The rollback becomes impossible at exactly the moment you need it.

`index.html` **is cached.** Deploys stop reaching users, and the cause is hunted in the code.

No way to find out what broke. After shipping you need logs and error reports, or users will report the breakage — or won't.

Shipping done by hand and from memory. While it's five terminal commands you remember, one day you will run them in a different order.

The version isn't visible in the application. When a user complains, the first question is which version they have. If there is nowhere to look, the conversation doesn't happen.

A migration not tested on a copy of the data. Running a structure change straight against the real database is the most expensive possible way to find out it has a bug.

CHEAT SHEET

- After shipping there are always several versions: think about the people on the old one.
- A website rolls back instantly, a server if you prepared, an app never.
- A hash in the file name is cache protection; `index.html` must not be cached.
- “The browser has the old one” — check a private window before searching the code.
- In an API you can add; you can’t remove or rename.
- The database field first, then the code; deletions separately and later.
- Code has a rollback, the database doesn’t.
- Roll out to a fraction of users, and don’t ship on a Friday.

12.4 A checklist before shipping

once per project and once per release

WHAT IT IS

Two lists. The first is worked through once, at the start of a project, and costs dearly if forgotten. The second runs before every release, and is worth keeping right in the repository, next to the code.

ONCE PER PROJECT

1. The application identifier. Not `com.example`. It can’t be changed after release (article 12.2).

2. The signing key and its backup. In two places, neither of them your working computer.

3. The `.gitignore` — checked by hand, not taken on trust. Here is a real `.gitignore` from a fresh template of a popular bundler:

```
node_modules
dist
dist-ssr
*.local
```

There is no line for `.env` in it. The check: create a `.env` file with a key and do an ordinary `git add -A`:

```
$ git ls-files | grep -c "\.env"
1
```

The secret has gone into the repository. Not because somebody made a mistake, but because that is how the template is built: `*.local` covers `.env.local`, not `.env`. Hence the rule: after the first commit, look with your own eyes at **what exactly** landed in the repository:

```
git ls-files
```

Once, at the start, while the history is short. Removing a file from the history afterwards is far harder than not adding it.

4. The exact-versions file is committed — `package-lock.json`, `pubspec.lock`, `poetry.lock` (article 12.1).

5. The build is tested from a clean clone. Not from your own folder where everything already works, but like this:

```
git clone <your repository> /tmp/trial
cd /tmp/trial && npm ci && npm run build
```

That is the only way to find out that the project builds somewhere other than your machine. Everything that was only on your disk and never committed shows up here.

6. The version is visible in the application itself. A line on the "about" screen or in the site's footer. Without it a conversation with a user about a breakage is impossible.

7. The hosting's caching settings. Hashed files for a long time, `index.html` never (article 12.3).

8. Logs and errors are collected somewhere. Otherwise you learn about breakages from users.

BEFORE EVERY RELEASE

1. The checks aren't switched off. Whatever checks the build is still in the build command: `tsc -b`, `mypy`, `flutter analyze`. Whether they were switched off since last time is something to look up in the change history, not to remember.

2. The tests have been run. At least the ones that exist (chapter 4).

3. A build from scratch. Delete `dist`, `build` and the dependency folder and build again. Old artifacts are good at hiding breakage.

4. A search for secrets in the artifact. Literally:

```
grep -r "sk-\|secret\|password\|api_key" dist/
```

If something is found, that is a leak, not a nitpick (article 12.1).

5. The assets are there. Images, fonts and data files are declared in the build settings and present in the artifact.

6. The version is raised. For a store, the number has to grow.

7. You ran the artifact, not the sources. Open the built folder, install the built package, put the built `.apk` on a phone. Checking the sources checks something other than what the user will get.

8. API and database changes worked through per article 12.3. What was added, what was renamed, what can't be rolled back.

9. The rollback is written down. One line: the command or the number of the previous release. It has to be written before shipping, because afterwards there won't be time.

> **REMEMBER**
> The main item in both lists is the fifth of the first: **build the project from a clean clone**. On its own it closes most of this chapter's errors: uncommitted files, undeclared assets, drifting versions, forgotten environment settings.

WHAT TO TELL THE AI

PROMPT

```
Draw up a shipping checklist for my project.
Stack: <list it>. Shipped to: <where>.
Split it into "once per project" and "before every release".
Only what applies to my stack, no general advice.
Format it as a list I'll put in the repository.
```

The resulting file is worth keeping in the project and adding to every time something breaks. A checklist grown out of your own mistakes works better than any ready-made one.

PROMPT

```
Here is my .gitignore: <contents>.
What that's usual for this stack isn't covered?
Separately: which of the already-committed files look like
secrets.
```

PROMPT

```
I've built the project. Write the commands to check the
artifact before shipping: searching for secrets, checking
the assets are there, running the built version. For my OS:
<which>.
```

WHAT TO LOOK FOR

The checklist is in your head. Until it is written down, it is different every time.

“I’ll check later”. Shipping is the one step after which “later” means “at the users”.

An edit made straight in the artifact. A file patched in `dist/` or on the server for a quick fix. The next build wipes it and the breakage comes back — now with no explanation.

Shipping with no way back. You don’t know how to restore the previous version until you have tried.

CHEAT SHEET

- Two lists: once per project, and before every release.
- A template `.gitignore` doesn’t cover `.env` — check `git ls-files` yourself.
- Build from a clean clone: that is the chapter’s central check.
- Before a release, build from scratch rather than on top of the old build.
- Search the artifact for secrets with `grep`, not with your eyes.
- Run and check the artifact, not the sources.
- The way back is written down before shipping, not after.
- Your own checklist in the repository beats any ready-made one.

PART III

Cross-cutting topics

What belongs to no single stack and is decided once per project: how to break an idea into parts, where to keep the data, what to put in the settings. And a workshop — complete examples from idea to working program, in which everything so far comes together.

13 Architecture in plain terms

The decisions taken on day one and not redone afterwards. How to break an idea into parts, where to keep the data, what to put in the settings — and how to explain all of it to an AI in one paragraph, so it doesn't invent the structure anew in every answer.

13.1 How to break an idea into parts

three questions and one rule about direction

WHAT IT IS

Architecture sounds grand, but in practice it is the answer to one question: **what depends on what**. Not "which folders to create" — the folders will appear later, by themselves — but who is allowed to touch whom.

While the project is small this feels unnecessary: one file, everything works. The problem isn't size but that without an answer to that question **every new AI answer puts code wherever it was convenient at that moment**. A month later the project has three places that talk to the database, two different ways of reading settings, and a two-hundred-line function nobody wants to touch.

WHAT IT LOOKS LIKE IN CODE

Three questions for any idea

Before asking for code, answer three questions. Not the AI — yourself.

1. What is stored? List the things: a city, a user, a note, an order, a subscription. Those are nouns, and tables and data models come out of them.

2. What happens? List the actions: add a city, refresh the weather, send a notification. Those are verbs, and functions come out of them.

3. What is visible from outside? A screen, a page, a bot command, an API address. That is what people talk to the program through.

Three half-page lists, and the project already has a skeleton. The technique works for a two-hundred-line bot and for a twenty-screen application alike.

The rule about direction

Those three lists are then laid out in layers, and between the layers there is a single rule worth remembering:

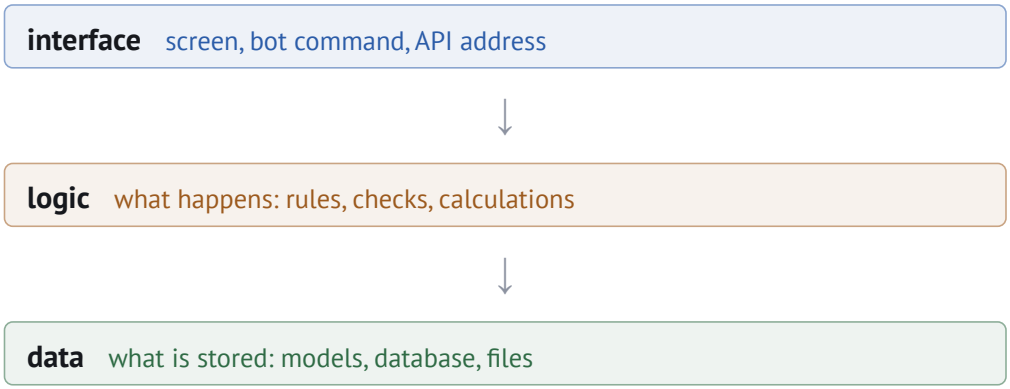


Fig. 13.1 · The arrows only go down. The data layer doesn't know there is a screen above it — which is why it can be checked on its own.

The upper layer knows about the lower one. The lower one doesn't know about the upper. The screen talks to the logic, the logic to the data. Never the other way.

That isn't theory or a matter of taste: breaking the rule is visible at once, because the program stops starting. Here are two files that know about each other:

```
# app/models.py
from app.services import fetch_weather

class City:
    def refresh(self):
        self.temp = fetch_weather(self.name)
```

```
# app/services.py
from app.models import City

def all_cities():
    return [City("Oslo"), City("Bergen")]
```

Running it:

```
ImportError: cannot import name 'fetch_weather' from partially
initialized
module 'app.services' (most likely due to a circular import)
```

A circular import is a message saying the layers have been mixed up. The data climbed upwards, into the logic. The cure isn't a clever import inside a function (that is what the internet advises, and what an AI often proposes) but reversing the dependency: the model knows nothing, the service knows about the model.

```
# app/models.py — imports nothing
class City:
    def __init__(self, name, temp=None):
        self.name = name
        self.temp = temp
```

```
# app/services.py — knows about the model
from app.models import City

def all_cities():
    return [City(n, fetch_weather(n)) for n in ("Oslo", "Bergen")]
```

```
[City(Oslo, 4), City(Bergen, 4)]
```

> **REMEMBER**
> A circular import isn't a mysterious Python error but **a signal that the architecture is wrong**. If an AI proposes getting round it by moving the import inside a function, it is treating the symptom. The right question is: which of these two files should know about the other?

The smaller the better — up to a point

It makes sense to split down to the level where **each piece can be explained in one phrase**. "Takes a city, returns a temperature." "Holds the user's list of cities." As soon as the explanation needs the word "and" three times, it is time to split further.

There is a limit in the other direction too: twenty files of ten lines isn't architecture, it is scatter. The sign that you overdid it: understanding one action means opening five files in a row.

WHAT TO TELL THE AI

The main mistake is asking for the whole application. Instead, get the breakdown **as text, before any code**:

> **PROMPT**

```
The idea: <describe it in two or three sentences>.
Don't write code. Break it into parts:
1. What is stored — a list of entities with their fields.
2. What happens — a list of actions.
3. What's visible from outside — screens or addresses.
For each action, say which entities it touches.
```

The answer to such a request reads in a minute and is corrected in two. Code that would have to be rewritten takes an evening.

PROMPT

```
Here is my breakdown: <the three lists>.
Propose a layer structure: what depends on what.
Draw with arrows who imports whom.
If an arrow comes out pointing both ways, say so.
```

PROMPT

```
I got an ImportError mentioning a circular import.
Don't propose moving the import inside a function.
Explain which of the two modules shouldn't know about the
other, and how to reverse the dependency.
```

WHAT TO LOOK FOR

The application in one piece. You asked for "make an app" and got one four-hundred-line file with everything mixed together.

Circular imports. Reverse the dependency rather than hiding the import.

A database query right in a screen's handler. The interface reached over the logic's head straight into the data. It works — until the first change in the database.

The same action in three places. Three places where the temperature is formatted, and all three have to be remembered.

A file called `utils` that keeps growing. Everything that found no home gets put there. That isn't a layer, it is a cupboard to sweep things into.

The AI did the breakdown and you didn't read it. The project's structure is worth understanding whole — it is the one part that can't be delegated (article 1.1).

CHEAT SHEET

- Architecture is the answer to "what depends on what".
- Three lists before any code: what is stored, what happens, what's visible from outside.
- Three layers: interface → logic → data. The arrows only go down.
- A circular import is a signal of mixed-up layers, not a Python error.
- A piece of the right size is explained in one phrase without the word "and".
- Ask the AI for the breakdown as text before code — text is cheaper to fix.
- A `utils` folder is where you sweep what you didn't sort out.

13.2 Folder structure

decisions you can't redo later

WHAT IT IS

The folder structure is the architecture from article 13.1 written into the file system. Its main value isn't beauty but that **it answers the question "where do I put this new code" without a discussion**. Until there is an answer, every new piece lands wherever the cursor happened to be.

For working with an AI the structure has a second, equally important role: it is what you name in the request. "Add it to `app/services/weather.py`" is a request you can get a predictable answer to. "Add a weather function" is an invitation to invent the structure again.

WHAT IT LOOKS LIKE IN CODE

Two ways of arranging things, and both are right

By kind of code — everything of the same kind together:

```
app/
├── models/        what is stored
├── services/      what happens
├── routes/        what's visible from outside
└── utils/
```

By feature — everything belonging to one subject together:

```
app/
├── weather/       the weather model, logic and screen
├── users/         the user model, logic and screens
└── shared/
```

Choosing between them isn't a matter of taste but the answer to **"what changes as a whole more often"**:

	Arranged by kind	Arranged by feature
Good when	the project is small, few subjects	many subjects, independent
An edit touches	several folders	usually one
Easy to see	what's in the project at all	how one subject works
Breaks down when	there get to be many subjects	the subjects start to interweave

For a first project, arranging by kind makes more sense: everyone understands it, AI included, and it involves fewer decisions. Moving to arrangement by feature makes sense once the project has more than five or six independent subjects.

What you shouldn't do is mix the two. A `models/` folder next to a `weather/` folder with its own models inside isn't a compromise but two answers to one question. After that, new code can go in both places.

What is expensive to redo

Not all decisions are equal. These three are worth thinking about on day one:

1. The boundary between the layers. If the logic and the database work are mixed together from the start, separating them later means rewriting nearly everything. Nobody has ever asked to go the other way, from separated to mixed.

2. The root package's name. It goes into every import in the project. Changing it means changing every file. Not `app` and not `src` if the project might one day become a package: something of your own, short and clear.

3. Where the data lives. The decision "we'll just put it in JSON for now" lives longer than planned (article 13.3).

The rest — folder names, file order, where the tests are — is redone in half an hour and isn't worth long deliberation.

File size

A guideline that works: **one file, one reason to change**. If you have to open a file both when the API's response format changes and when a screen's appearance changes, there are two different files inside it.

The numeric guideline is 300–500 lines. Not a rule but a signal: once it reaches a thousand, two files are probably hiding in there.

With an AI that has a practical consequence: **a file that fits into a request as a whole is far easier to discuss**. An edit in a two-hundred-line file is a precise request and a precise answer. An edit in a thousand-line file is a paraphrase and a hope.

> **TIP**
> It is worth writing the project's structure down once as text and keeping it in the repository — in `README.md` or a separate `ARCHITECTURE.md`. Then you can paste it whole into any request, and the AI stops proposing "create a `helpers.py` file" where you already have `services/`.

What shouldn't be there

```
app/
├── utils.py          ← 900 lines of everything
├── helpers.py        ← the same thing, but different
├── main_v2.py        ← the previous version, might come in handy
├── test.py           ← not a test, a "let's try it"
└── new/              ← we're rewriting, but let's keep the old one for
now
```

Every line here is a deferred decision. `utils` and `helpers` are "I'll sort it out later". `main_v2` and `new/` are "I'll delete it later". Neither ever happens, but every request to an AI now has to explain which files are the real ones.

The old doesn't live next to the new: git exists for that (article 5.1).

WHAT TO TELL THE AI

PROMPT

```
Here is my project's structure:
<folder tree>

I'm adding a feature: <which>.
Say which files the code will go into and which files will
have to change. Name new files by my scheme.
Don't write the code yet.
```

That request is worth making a habit: it takes a minute and removes the commonest cause of mess — code put in the wrong place.

PROMPT

```
Project structure: <tree>.
Rules: models imports nothing, services imports models,
routes imports services.
Check the file you just wrote against those rules.
```

PROMPT

```
I have a 900-line file: <contents or description>.
Propose which files to split it into and on what basis.
One phrase per file saying what it's responsible for.
Don't move the code yet.
```

PROMPT

```
The project has a utils.py with everything in it.
Distribute its contents across the existing folders.
For each function, say where and why.
If something really is shared, leave it, but name it more
precisely.
```

WHAT TO LOOK FOR

The AI proposed a new folder next to an existing one. `helpers/` while `services/` is alive, `db/` while `models/` is alive. A sign that the structure wasn't named in the request.

Two arrangements mixed. `models/` and `weather/` in one folder.

`utils` is growing. Every function put there is a deferred decision.

Files with versions in the name. `main_v2`, `app_new`, `index_final`. Git exists for that.

Tests mixed in with the code. A separate `tests/` folder alongside, not a `test.py` in every folder.

One file grows with every request. If the AI adds a fourth unrelated function to the same file, it is time to split.

CHEAT SHEET

- The folder structure answers "where do I put this new code".
- By kind or by feature — choose by what changes as a whole more often.
- Don't mix the two arrangements: you get two answers to one question.
- Expensive to redo: the layer boundary, the root package's name, where the data lives.
- One file, one reason to change; 300–500 lines as a signal.
- A file that fits into a request is many times easier to discuss.
- Write the structure down as text in the repository and paste it into requests.
- `utils`, `main_v2`, `new/` are deferred decisions, not folders.

13.3 Where to keep the data

a file, SQLite or a real database

WHAT IT IS

Four options, and the choice is nearly always between the second and the third:

Where	When it's the right answer
in memory	the data is only needed while the program runs
in a file (JSON, CSV)	settings, a cache, an export, a couple of hundred records
SQLite	one process, up to hundreds of thousands of records, searching needed
a database server	several processes, a network, access rights, growth

The decision is taken on day one, in one line, and almost never revisited — because by the time it starts getting in the way, the whole project rests on it. So it is worth knowing not only how the options differ but **exactly when the difference becomes noticeable**.

WHAT IT LOOKS LIKE IN CODE

A file versus SQLite: the numbers

The commonest scenario: "we'll just put it in JSON and redo it later". Keeping a list in a file means that on every write you read the whole file, append, and write the whole thing back.

```
with open("data.json", encoding="utf-8") as f:
    rows = json.load(f)
rows.append(record)
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(rows, f)
```

Here is how long adding records takes — measured, on the same machine:

Records	JSON, s	SQLite, s
500	0.5	0.5
1,000	1.7	1.0
2,000	6.1	1.5
4,000	26.3	3.6
8,000	89.1	7.7

What to look at isn't the difference in the last row but **how the columns grow**. Doubling the number of records roughly doubles SQLite's time and roughly quadruples JSON's: every new record forces a re-read of everything already accumulated.

And the main thing: **at five hundred records there is no difference** — half a second either way. Which is exactly why the wrong decision is invisible at the moment it is taken. It becomes visible six months later, when there are eight thousand records and the program stops to think for a minute and a half.

Searching

```
finding one city among 50,000
  JSON (load the file and scan):  19.9 ms
  SQLite with an index         :   0.02 ms
```

A thousand times. The reason is simple: a file has to be read whole and parsed before anything can be found in it, while a database reads only the page it needs (article 7.6).

Two processes writing at once

This is the difference no optimization gets round. Two processes each add 300 records to one JSON file:

```
run 1: 300 of 600 written (A=300, B=0)
run 2: 300 of 600 written (A=300, B=0)
run 3: 300 of 600 written (A=300, B=0)
```

Half the data vanished in all three runs. And in each run one of the processes crashed:

```
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

That is reading the file at exactly the moment the other process had emptied it but hadn't yet written the new contents.

The same test with SQLite:

```
SQLite — expected 600, written 600 (A=300, B=300), lost 0
```

A database can do what a file fundamentally cannot: **put simultaneous writes in a queue**. That is the main reason databases exist — not speed, but the fact that data isn't lost.

> **CAREFUL**
> If your program can have **two running instances at once** — a web server with several worker processes, a bot plus a maintenance script, two tabs with the same application — a file as storage is unsuitable. Not "slow", but losing data. Silently.

When a file is enough

A file is excellent storage when **one process writes, and rarely**:

- settings a human edits by hand;
- a cache of an external API's responses;
- exports and data exchange;
- a reference list updated once a month;
- a small bot only you use.

The sign that a file will do: you can open it in a text editor and that is useful.

When SQLite stops being enough

SQLite is not a toy database. It is a full SQL database that lasts a very long time, and for one application on one server it is most often the right answer. Moving to a server database (PostgreSQL, MySQL) makes sense once at least one of these appears:

- **several machines** that need the same data;
- **many simultaneous writers** (SQLite admits one writer at a time);
- **access rights** — different applications with different access to different tables;
- **a volume** at which you need tuning, scheduled backups, replication.

The other side: a server database is another program to install, configure, update and back up. For a bot used by ten people that is work with no payoff.

> **REMEMBER**
> The honest order of choice: **in memory → a file → SQLite → a database server**. Take the next level when the previous one stopped coping for an understandable reason, not "for future growth". But jumping from a file straight to SQLite is nothing to fear: it is one line of `sqlite3.connect`, not new infrastructure.

The move people make too late

The most expensive part of moving from a file to a database isn't the database but the fact that **the code talks to the storage from ten places**. If reading and writing are gathered into one file from the start — `storage.py`, `repository.py`, whatever — the move takes an evening, because one implementation changes and not the whole project.

That is perhaps the most useful architectural habit in the whole chapter: **all data access through one door**.

WHAT TO TELL THE AI

By default an AI proposes what appeared most in its training data — usually PostgreSQL and an ORM. For a two-hundred-line bot that is surplus. So name the choice yourself and ask the AI for the justification:

PROMPT

```
Project: <description>. I expect <how many> records.
Writing is done by <one process / several>.
The data is needed by <just me / several users>.
Which of these is enough: a file, SQLite or a database
server? Name the sign by which the chosen one will stop
being enough.
```

That last line is the valuable one: it turns the choice into a checkable condition.

PROMPT

```
Put all data access through one storage.py module:
the functions add_city, get_city, list_cities, delete_city.
SQLite inside. The rest of the code mustn't know about the
database.
```

PROMPT

```
The data is currently in a JSON file and access is smeared
across the project. Find every place where that file is read
or written. A list with line numbers. Don't change the code
yet.
```

PROMPT

```
Move the storage from JSON to SQLite.
Don't change storage.py's external functions or their
signatures. Separately, write a migration script for the
data already accumulated, and say what to do if it fails
halfway.
```

WHAT TO LOOK FOR

A JSON file as the main storage for a server or a bot. That is data loss, not slow operation.

Data access from ten places. Moving to a database turns into rewriting the project.

PostgreSQL for a single-user project. Configuration, backups and updates for a load that doesn't exist.

No check that the write succeeded. With a file, a write error often goes unnoticed.

A data migration with no backup. A migration script is the prime candidate for failing halfway.

Passwords and keys in the same database in plain text. The storage was chosen and the question of what is in it was never asked (article 5.7).

No answer to "what happens if the file is opened twice". Worth asking before the answer arrives as missing data.

CHEAT SHEET

- The order of choice: memory → a file → SQLite → a database server.
- At 500 records there is no difference; at 8,000 it is 89 seconds against 7.7.
- Write time to a file grows quadratically: every write re-reads everything.
- Finding one of 50,000: 19.9 ms in a file against 0.02 ms in an indexed database.
- Two processes on one file — half the data is lost, and that is measured.
- A file is good where one process writes, and rarely.
- SQLite lasts a long time; a database server is for when there is more than one machine.
- All data access through one module. That is the insurance for the move.

13.4 Configs and environments

three disasters from one small file

WHAT IT IS

An environment is the set of settings the same code is started with. There are usually three: **your machine**, **a test server**, **a production**

server. They differ not in code but in values: the database address, API keys, whether debugging is on.

Settings are moved out of the code for two reasons: so secrets aren't kept in the repository (article 5.7), and so one program can work in different places. The mechanism is the same everywhere — **environment variables** — and they hold exactly one subtlety that breaks projects: **everything that comes from the environment is a string**.

WHAT IT LOOKS LIKE IN CODE

A settings file that looks fine

```
import os

DATABASE_URL = os.environ.get("DATABASE_URL", "sqlite:///test.db")
PORT = os.environ.get("PORT", 8000)
DEBUG = os.environ.get("DEBUG", False)
```

Three lines, three disasters. And on your machine, where no variables are set, everything is flawless:

```
database: sqlite:///test.db
port    : 8000 int
debug   : False bool
  -> debug mode is off
port + 1 = 8001
```

Now the same thing with the variables — the way it starts on a server:

```
PORT=8080 DEBUG=false DATABASE_URL=postgresql://prod/db python3 app.py
```

```
database: postgresql://prod/db
port    : '8080' str
debug   : 'false' str
  -> debug mode is ON
port + 1 = TypeError: can only concatenate str (not "int") to str
```

One at a time.

1. `DEBUG=false` turned debugging on. The string `'false'` is non-empty, and therefore truthy. The `if DEBUG:` check fires. On a production server that means detailed error pages, tracebacks facing outwards, and everything normally seen only by developers. A one-line mistake; the consequence is a leak.

2. The port became a string. The `TypeError` is actually the lucky case: at least it is loud. Worse is when the string passes on silently.

3. The type depends on whether the variable is set. On your machine `PORT` is a number, on the server a string. The same code, different types. You cannot check that locally at all.

And separately, **the quietest of the three**: `DATABASE_URL` with the default `sqlite:///test.db`. The variable wasn't set on the server — the application didn't crash, didn't complain, and started writing to a local file. You find out when the data is looked for and isn't there.

> **CAREFUL**
> A default value for a setting the program shouldn't work without isn't a convenience but **a way of hiding an error**. For a database address, keys and passwords there are no defaults: with no value the program is obliged to fail at once and say what it is missing.

How it should be

```
from pydantic import Field
from pydantic_settings import BaseSettings


class Settings(BaseSettings):
    database_url: str
    api_key: str
    port: int = 8000
    debug: bool = False
    timeout: float = Field(default=5.0, gt=0)


settings = Settings()
```

The whole difference is in the types. `port: int` means the string `'8080'` becomes a number at startup or the program doesn't start. `debug: bool` means `false` becomes a real `False`. `database_url` and `api_key` have no defaults — which means without them there is no startup.

Running it with no variables:

```
pydantic_core._pydantic_core.ValidationError: 2 validation errors for
Settings
database_url
  Field required [type=missing, input_value={}, input_type=dict]
api_key
  Field required [type=missing, input_value={}, input_type=dict]
```

Both missing variables are named at once, at startup. Not one at a time, not mid-run, not an hour after the deploy.

With the variables:

```
database: postgresql://prod/db
port    : 8080 int
debug   : False bool
port+1  : 8081
```

`DEBUG=false` became `False`, as any common sense expects. And if a value turns out to be junk:

```
port
  Input should be a valid integer, unable to parse string as an integer
  [type=int_parsing, input_value='eight', input_type=str]
```

The setting is named, the value is named, the reason is named. That is the difference between "the program crashed" and "the program explained".

> **REMEMBER**
> Settings are checked **once at startup**, not at the moment of use. A program that dies forty minutes into its run because of an empty API key is a program that didn't check its settings at startup.

What goes into the repository and what doesn't

```
.env           ← the real values. In .gitignore. Never committed.
.env.example   ← the same names, empty or sample values. Committed.
```

`.env.example` is a small file with a big payoff: it is **the list of what has to be set** for the project to start. Without it a new person (or you yourself in six months on another machine) learns about a missing variable by trial and error.

```
DATABASE_URL=postgresql://user:password@localhost/dbname
API_KEY=
PORT=8000
DEBUG=false
```

And a reminder from article 12.4: in project templates the `.env` line may not be in `.gitignore`. That is checked once, with `git ls-files`, and preferably on day one.

Three environments, one codebase

The rule is simple: **the code doesn't know where it is running**. There are no `if environment == "production"` scattered across the project; there are settings that differ from place to place.

The one justified exception is a single variable such as `ENV=production` and a few decisions based on it in one place: the log level, whether to show detailed errors, whether to switch the API documentation on.

WHAT TO TELL THE AI

PROMPT

```
Make a settings module with pydantic-settings.
I need: database_url (string, required),
api_key (string, required), port (int, default 8000),
debug (bool, default false).
Don't put defaults on the required ones.
Alongside it, a .env.example with all the names.
```

PROMPT

```
Find every use of os.environ and os.getenv in the project.
For each, say: the value's type, whether there's a default,
and what happens if the variable isn't set.
As a table. Don't change the code yet.
```

That request is worth running on any project more than a month old: environment lookups spread unnoticed.

PROMPT

```
Check this config for three mistakes: a string where a
number belongs, the string 'false' where a boolean belongs,
a default where there shouldn't be one. List what you find.
```

PROMPT

```
The project should fail at startup if the required settings
aren't set. At the moment it fails later. Add a check at
startup, with a clear message listing everything missing at
once.
```

WHAT TO LOOK FOR

`os.environ.get` **with a default for a database address or a key.** The quietest of this article's mistakes.

A setting used as a number or a flag without conversion. What comes from the environment is a string, always.

`DEBUG=true` **as a string.** Any non-empty string is truthy, `'false'` and `'0'` included.

Settings read in ten places. One settings module; the rest of the code asks it.

`.env` **in the repository.** Checked with `git ls-files`.

No `.env.example`**.** The project can't be started without guessing the list of variables.

Keys written into the code "temporarily". The temporary lasts longest, and git remembers everything (article 5.7).

`if production:` **all over the project.** The code knows where it is running — which means the test server isn't testing what goes into production.

CHEAT SHEET

- Everything that comes from the environment is a string. Always.
- `DEBUG=false` without conversion turns debugging on.
- A default for a required setting hides an error rather than helping.
- Typed settings name every missing variable at once.
- Settings are checked at startup, not at the moment of use.
- `.env` into `.gitignore`, `.env.example` into the repository.
- One settings module; the rest of the code reads it, not the environment.
- The code shouldn't know which server it is running on.

13.5 How to describe the architecture to an AI

twenty lines that save you an evening

WHAT IT IS

An AI has no memory of your project. In each new conversation it knows only what it can see right now — and if it can't see the structure, it **invents its own**. Hence answers like "create a `helpers.py` file" while a `services/` folder is alive and well, and a fourth way of reading settings.

The cure is one short text — an **architecture brief**. Article 3.14 talked about the context file in general: what the project is, what the commands are, how to run it. This is about the part of it that answers "how is it put together", and that is worth writing separately, because it is the part an AI breaks most often.

WHAT IT LOOKS LIKE IN CODE

The template

The brief consists of six blocks. No more are needed: a long text reads worse and goes out of date faster.

```
1. WHAT IT IS        — one sentence on what the program does
2. STACK             — language, framework, database, versions
3. STRUCTURE         — the folder tree, one line of explanation per
folder
4. DEPENDENCIES      — who is allowed to import whom
5. WHERE THINGS LIVE — data, settings, secrets, logs
6. WHAT NOT TO DO    — decisions that are not up for discussion
```

The sixth block is the most underrated. Without it, in every third answer an AI proposes adding a library, switching databases, or "refactoring this while we're here".

A filled-in example

```
WHAT IT IS
A Telegram bot: subscribe to the weather in chosen cities,
a morning digest.

STACK
Python 3.12, aiogram 3, SQLite, requests.
Run with: python -m bot. One server, systemd.

STRUCTURE
bot/
├── handlers/   commands and buttons, only input parsing and the reply
├── services/   logic: weather, subscriptions, the digest
├── storage.py  all database access, no other DB calls anywhere
├── models.py   the City and Subscription dataclasses
└── config.py   settings via pydantic-settings

DEPENDENCIES
handlers -> services -> storage -> models
No reverse imports. models imports nothing from the project.

WHERE THINGS LIVE
Data: SQLite, the file data/bot.db
Settings: environment variables, see .env.example
Secrets: only in .env, never in the repository
Logs: stdout, picked up by systemd

WHAT NOT TO DO
- Don't add an ORM or other libraries without a separate conversation.
- Don't talk to the database from handlers directly.
- Don't change the table schema without a migration script.
- Don't add async where there isn't any.
```

A little over twenty lines. Pasted at the start of a request when the task is large, and sitting in `ARCHITECTURE.md` or the context file the rest of the time.

Why this works better than it looks

The brief removes exactly the uncertainty an AI would otherwise close with a guess. It doesn't guess where database access lives — it is written down. It doesn't choose between SQLite and PostgreSQL — written down. It doesn't invent a name for a new file — there is a scheme.

The side effect matters more than the main one: **to write such a brief you first have to know it**. If the "dependencies" block won't write

itself because everything in the project imports everything, the brief has already paid for itself, just differently.

> **TIP**
> You write the brief, not the AI. You can ask it to **assemble a draft** from the code, but checking and correcting it is yours to do: a brief with things in it that aren't in the project is worse than no brief at all — the AI will confidently refer to what doesn't exist.

A check that takes a minute

The fastest way to find out whether the AI has understood the architecture:

PROMPT

```
Retell in your own words how my project is put together:
what depends on what and where things live. Five lines.
If anything is missing from the description for you to start
work, say exactly what.
```

The retelling shows up the discrepancies at once. The second part of the question is no less useful: what the AI is missing is usually the hole in the brief.

When to update the brief

Not when the code changes but **when a decision changes**. Added a library, switched storage, introduced a new layer, dropped a rule — the brief gets rewritten the same day.

The sign that the brief is out of date: the AI does it "wrong", but exactly as written.

WHAT TO TELL THE AI

PROMPT

```
Here is my project's tree and a few key files.
Assemble a draft architecture brief in six blocks:
what it is, stack, structure, dependencies, where things
live, what not to do.
Where you aren't sure, write "?" — don't invent.
```

The instruction about `?` is mandatory: without it the gaps are filled with plausible fiction (article 1.6).

PROMPT

```
Here is the project's architecture brief: <text>.
The task: <what needs doing>.
First say which files you'll touch and whether that breaks
the dependency rules. Write the code after I say yes.
```

PROMPT

```
Here is the brief, and here is the code you just wrote.
Check your code against the brief: dependencies, file
locations, the ban on new libraries.
List the violations.
```

That last technique — **the AI checking itself against your rules** — works noticeably better than the same rules stated at the start of the conversation: the check runs against finished code rather than against an intention.

WHAT TO LOOK FOR

A brief written by the AI and not read by you. It may describe a project other than yours.

No "what not to do" block in the brief. The most useful block, and the most often forgotten.

An out-of-date brief. The AI follows what is written, and what is written is no longer true.

A three-page brief. You can't paste that into a request or read it with your eyes; anything longer than a screen stops working.

No versions. "Python and a Telegram bot" is both aiogram 2 and aiogram 3, and they are incompatible.

The brief exists but isn't used in requests. A file in the repository won't insert itself into the conversation — unless it is the context file the tool picks up by itself (article 3.14).

CHEAT SHEET

- An AI has no memory of the project: describe no structure and you get an invented one.
- Six blocks: what it is, stack, structure, dependencies, where things live, what not to do.
- The "what not to do" block saves the most time.
- Twenty lines is the limit; nobody reads longer, the AI included.
- The AI can assemble the draft, but you check and correct it.
- "Retell the architecture in your own words" is a one-minute check.
- Update the brief when a decision changes, not when the code changes.
- Ask the AI to check its own code against the brief — that works.

13.6 Questions before starting a project

ten of them, half an hour, an evening saved

WHAT IT IS

The chapter's closer and its one practical outcome: **a list of questions worth answering before the first line of code**. Not because that is the proper way, but because each of them costs dearly if you answer it a month later.

Half an hour with this list turns a request to an AI from "make me a bot" into a mini-spec (article 3.15), which can produce a predictable result.

THE TEN QUESTIONS

1. What does the program do, in one sentence? If the sentence won't form, the project hasn't been thought up yet, rather than "not written yet".

2. What is stored? A list of nouns. Tables and models come out of it (article 13.1).

3. Where is it stored? Memory, a file, SQLite, a database server — and the sign by which the chosen one will stop being enough (article 13.3).

4. Who uses it, and how many of them? Just you, ten acquaintances, or unknown people from the internet. Everything depends on the answer: authentication, load, handling other people's input, the cost of a mistake.

5. What can't be lost? Data splits into two piles: what can be gathered again and what can't. The second pile needs an answer about backups — on day one, not after the first loss.

6. Which external services are needed, and what happens when they don't answer? Every external API fails to answer one day. The options: show old data, show an error, retry later. The choice has to be made in advance, or chance makes it.

7. What changes most often? The message texts? The calculation rules? The appearance? What changes often should live separately and change without editing code.

8. How many environments, and where are the secrets? One machine, or a machine plus a server; how the keys get there (article 13.4).

9. How does it start, and who restarts it if it dies? "I do, by hand" is a fine answer for an evening's script and a poor one for a bot people use.

10. How will I know it broke? Logs, a notification, a liveness check. Without an answer to this, breakages are reported by users — or not reported, and they simply leave.

> **REMEMBER**
> Half the questions are about failures: what gets lost, what to do when an external service is unavailable, who restarts it, how you find out about a breakage. That isn't pessimism: **the difference between a script and a program is that a program keeps working when something has gone wrong.**

WHAT TO TELL THE AI

This list is convenient to work through with an AI — it asks good clarifying questions when asked outright to (article 3.11):

PROMPT

```
The idea: <describe it in two or three sentences>.
Ask me one question from the list at a time and wait for the
answer before moving on to the next:
1) what it does, in one sentence
2) what is stored
3) where it's stored
4) who uses it and how many
5) what can't be lost
6) external services and behaviour when they fail
7) what changes most often
8) environments and secrets
9) starting and restarting
10) how I find out it broke

Don't write code. At the end, gather the answers into a
mini-spec.
```

The resulting mini-spec is what a normal project starts from. The architecture brief (article 13.5) is assembled from it too.

PROMPT

```
Here are my answers to the ten questions: <text>.
Which decisions follow from them by themselves, and which
have I not made yet? The latter as a list, with the options
and one line of consequences per option.
```

WHAT TO LOOK FOR

The questions were asked and the answers weren't written down. A week later all that remains is a general impression.

The AI answered the questions. It can propose options, but the choice is yours (article 1.1): these are precisely the decisions that aren't redone later.

"I'll work that out later" for questions 3, 5 and 8. Storage, backups and secrets cost the most "later".

A mini-spec written but never used in requests. Then it isn't a spec, it is a diary.

CHEAT SHEET

- Half an hour on ten questions saves an evening of rewriting.
- If the sentence about what the program does won't form, the project hasn't been thought up.
- For every choice, record the sign by which it will stop working.
- Half the questions are about failures, and that is the main difference between a program and a script.
- What changes often is moved out of the code.
- The human answers; the AI asks the questions and gathers them into a mini-spec.
- The answers go into the repository and become the brief from article 13.5.

14 Workshop

Five projects from "I want to build" to a working program. Everything so far comes together here: the questions before starting, the mini-spec, a bad request against a good one, review, a real bug and its fix. The code in these cases was run, and the mistakes in them are the ones that actually happened.

14.1 Case — a Telegram bot with an external API

the full cycle, two real bugs

WHAT IT IS

The first case, and at the same time a model for the rest: the idea, the questions, the mini-spec, the request, the plan, the code, the review, a bug, the fix. Nothing new in technique — all of it has appeared in the book already; what is new is that they come one after another, as they do in life.

The idea in one phrase: **a bot that sends the morning weather for your chosen cities.**

STEP 1. THE TEN QUESTIONS

Half an hour with the list from article 13.6. Briefly, the answers only:

Question	Answer
what it does	sends the weather for chosen cities every morning
what is stored	subscriptions: who to which city
where it's stored	SQLite, few records, one process writing
who uses it	me and a dozen acquaintances
what can't be lost	the subscription list; the weather can, it refreshes
external services	one weather API; no answer — show the previous data
what changes most	the message texts

environments	my machine and one server; keys in `.env`
starting	systemd, restarts by itself
how I find out it broke	logs in the journal plus a message to myself if the digest fails

The sixth row is the one the list exists for. Without it the bot stays silent one morning and it isn't clear whether it broke or the weather simply didn't change.

STEP 2. THE REQUEST

How not to do it:

DON'T DO THIS

```
Write a telegram bot in python that shows the weather.
```

What arrives: one file, a synchronous `requests` inside an async handler, the API key as a string in the code, the city list in a global variable. It will work — right up to the first restart.

How to do it (and this is the mini-spec from article 3.15):

BETTER

```
A bot on aiogram 3, Python 3.12. Storage: SQLite.
Features: /add <city>, /list, /remove <city>, a morning digest.

Structure:
bot/handlers/  only input parsing and the reply
bot/services/  logic and the call to the weather API
bot/storage.py all database access, nowhere else
bot/config.py  settings via pydantic-settings

Rules: handlers don't talk to the database directly;
no new libraries without my agreement;
the API key only from an environment variable.

Write the plan first: which files and what's in each.
Code after I say yes.
```

Those last two lines are the technique from article 3.10. The plan reads in a minute, and rewriting it is cheaper than rewriting code.

STEP 3. STORAGE AND THE FIRST BUG

Start at the bottom, with the data layer (article 13.1). Here is what arrived:

```
def connect():
    db = sqlite3.connect(DB)
    db.execute("""
        CREATE TABLE IF NOT EXISTS subscriptions (
            user_id INTEGER NOT NULL,
            city    TEXT    NOT NULL
        )
    """)
    db.commit()
    return db


def add_city(db, user_id: int, city: str) -> None:
    db.execute("INSERT INTO subscriptions VALUES (?, ?)", (user_id,
city))
    db.commit()
```

The code reads well, works, and looks right. Write the tests — not for appearances but by the list from article 3.9: the ordinary case, the edges, the thing people forget.

```
def test_add_and_list(db):
    storage.add_city(db, 1, "Oslo")
    assert storage.list_cities(db, 1) == ["Oslo"]


def test_cities_are_per_user(db):
    storage.add_city(db, 1, "Oslo")
    storage.add_city(db, 2, "Bergen")
    assert storage.list_cities(db, 1) == ["Oslo"]


def test_same_city_twice(db):
    storage.add_city(db, 1, "Oslo")
    storage.add_city(db, 1, "Oslo")
    assert storage.list_cities(db, 1) == ["Oslo"], "the city shouldn't
be added twice"
```

Running it:

```
E       AssertionError: the city shouldn't be added twice
E       assert ['Oslo', 'Oslo'] == ['Oslo']
E         Left contains one more item: 'Oslo'

1 failed, 3 passed in 0.03s
```

The third test failed. Add a city twice and you get two subscriptions, which means two messages every morning. You don't find that by hand: who would add the same city twice? A user would — a month later, having forgotten they already had.

The fix goes where the cause is, not where the symptom is. Not a check in the code before the insert (which you can forget to call from somewhere else) but **a rule in the table itself**:

```
            user_id INTEGER NOT NULL,
            city    TEXT    NOT NULL,
            UNIQUE (user_id, city)
```

```
    db.execute("INSERT OR IGNORE INTO subscriptions VALUES (?, ?)",
               (user_id, city))
```

```
4 passed in 0.02s
```

> **REMEMBER**
> A rule written into the table's schema always works. A check written into the code works only where nobody forgot it. If a condition matters, it belongs in the database.

STEP 4. THE EXTERNAL API AND THE SECOND BUG

The first version of the weather call looked like this:

```
def fetch(city: str) -> float:
    r = requests.get(API_URL, params={"q": city, "appid":
settings.api_key})
    return r.json()["main"]["temp"]
```

Two problems in two lines, and both typical (article 2.14): parsing the response without checking — and **no timeout**.

Let's check the second. Start a local server that has "hung" — it accepts the request and says nothing:

```
class Slow(BaseHTTPRequestHandler):
    def do_GET(self):
        time.sleep(30)
        ...
```

```
no timeout:     still hanging after 12 s  (we aborted it ourselves)
with timeout=2: requests.Timeout after 2.0 s
```

Twelve seconds is how long we agreed to wait before aborting the check. A real hung server holds the connection as long as it likes, and **the whole morning digest** hangs with it: one unreachable city and the other subscribers get nothing.

`requests` has no default timeout. None at all. That is a property of the library an AI remembers about half the time.

The corrected version — with a timeout, with a check, and with the answer to "what do we do if it doesn't respond" from the mini-spec:

```
def fetch(city: str) -> float | None:
    try:
        r = requests.get(API_URL, params={...}, timeout=5)
        r.raise_for_status()
    except requests.RequestException:
        return None                      # the caller shows the previous
data
    data = r.json()
    main = data.get("main")
    if not isinstance(main, dict) or "temp" not in main:
        return None
    return main["temp"]
```

Checked against the local server — a real city, an invented city, and the server switched off:

```
Oslo            -> 4.2
Madeupville     -> None
server is down  -> None
```

Not one crash: the function has an answer for all three cases.

One detail is left, and it is the very one step 2 started with: `fetch` is synchronous and the bot is asynchronous. Called straight from a handler or the digest, it stops the whole bot for the length of the request — nobody gets a reply until the weather arrives. So it is called in a separate thread:

```
temp = await asyncio.to_thread(fetch, city)
```

The other route is to rewrite `fetch` with `aiohttp`. Both are honest; what matters is that a synchronous request never sits inside an `async def` without such a wrapper.

STEP 5. WHAT THIS CASE DOESN'T HAVE

What was run in this case is the data layer (four tests) and the behaviour against a hung server (a local server, real measurements). The handlers and a run against the real Telegram and a real weather API are not shown here: that is the next step, and it is done with the checklist from article 6.6, not on trust.

That is exactly the honesty the book demands of an AI (article 1.6), and it applies to itself the same way.

CHEAT SHEET

- The order: questions → mini-spec → plan → code from the bottom up → tests → run.
- Start with the data layer: everything else stands on it.
- The "the same thing twice" test finds a bug you don't find by hand.
- A condition that matters to the data lives in the table's schema, not in the code.
- `requests` has no default timeout — always set one.
- One hung request in a digest stops the whole digest.
- A synchronous request from an asynchronous bot goes through `asyncio.to_thread`.
- "What do we do if the service doesn't answer" is decided in the mini-spec, not at the moment of failure.

14.2 Case — a web API with a database

and why a 500 instead of a 409 is a bug

WHAT IT IS

A continuation of the first case. The bot from article 14.1 acquires neighbours: a website and a mobile application. Keeping the city list in three places won't do, which means we need **one API that everyone reads from**.

The idea in one phrase: **a service that stores a user's cities and serves them over HTTP.**

STEP 1. THE MINI-SPEC

The questions are the same (article 13.6), so here is only what differs from the bot:

Question	Answer
who uses it	the bot, the website, the app — programs, not people
what is stored	cities and the last known temperature
where it's stored	SQLite: one process, few records (article 13.3)
what can't be lost	the city list
how I find out it broke	the response codes in the logs

From the "programs use it" row follows the main requirement: **the responses have to be predictable**. A program can't guess what "something went wrong" means — it needs a response code that tells it what to do.

STEP 2. THE REQUEST

```
BETTER
An API on FastAPI, Python 3.12, storage in SQLite.

Endpoints:
GET    /cities        the list of cities
POST   /cities        add {name, temp}
DELETE /cities/{name} remove

Requirements:
- request and response schemas with pydantic, city name 1–60 characters
- all database access in one place
- response codes: 201 on creation, 204 on deletion,
  404 if there's nothing to delete, 409 if the city already exists
- no ORM, the queries are written by hand

The list of files and what's in each first. Code after "yes".
```

The line about response codes is the important one. Without it you get code that answers 200 to everything and reports errors as text inside the response.

STEP 3. WHAT ARRIVED

The structure came out right, the schemas are in place, the queries are parameterized (that matters — substituting a string opens an SQL injection, article 7.8). The add handler:

```
@app.post("/cities", response_model=CityOut, status_code=201)
def add_city(city: CityIn):
    with closing(db()) as conn:
        conn.execute("INSERT INTO cities VALUES (?, ?)", (city.name,
city.temp))
        conn.commit()
    return city
```

Start it and check by hand — exactly as article 4.7 advises: not only the happy path.

```
$ curl -X POST localhost:8102/cities -d '{"name":"Oslo","temp":4}'
201

$ curl -X POST localhost:8102/cities -d '{"name":"Oslo","temp":4}'
Internal Server Error
HTTP 500
```

The second time, a 500. The reason is in the logs:

```
sqlite3.IntegrityError: UNIQUE constraint failed: cities.name
```

What happened: in the table's schema `name` is declared the primary key — and rightly so, there shouldn't be duplicates (the lesson from article

14.1). But the handler doesn't know that, the exception flies out, and FastAPI turns it into a 500.

> **REMEMBER**
> **A 500 is always a bug.** It means "the program didn't allow for this case". An expected situation — a duplicate, something missing, no permission — has a code of its own: 409, 404, 403. If a user's action can produce a 500, the case wasn't thought through; it isn't "the user did something odd".

The fix is four lines:

```
        try:
            conn.execute("INSERT INTO cities VALUES (?, ?)", (city.name,
city.temp))
        except sqlite3.IntegrityError:
            raise HTTPException(409, f"city {city.name} is already
added")
```

STEP 4. ACCEPTANCE

Check every case from the mini-spec one after another — the happy ones and the unhappy ones:

```
first POST  : 201
second POST : {"detail":"city Oslo is already added"} | HTTP 409
GET         : [{"name":"Oslo","temp":4.0}]
DELETE Oslo : 204
DELETE again: 404
```

And the data validation nobody wrote — it came from the schema:

```
$ curl -X POST localhost:8103/cities -d '{"name":"","temp":4}'
422
```

```
{"detail":[{"type":"string_too_short","loc":["body","name"],
 "msg":"String should have at least 1 character","ctx":
{"min_length":1}}]}
```

The response names the field, names the rule, names the violation. That is the point of the schemas from article 7.3: the checks are written once, next to the description of the data, rather than in every handler.

The five lines of output above are five checks, each matching a line of the mini-spec. Such a run takes a minute and replaces the question "but does it actually work?".

STEP 5. WHAT IS DELIBERATELY ABSENT

No authentication. Anyone who knows the address can add a city. For a service on your own machine that is a conscious decision; for a

service on the internet it is a hole. The decision is written in the mini-spec rather than forgotten, and that is the whole difference.

No pagination. There are dozens of cities. When there are thousands it will appear, and the sign was named in advance (article 13.3).

No ORM. There are five queries and all of them are simple. An ORM is another library, another layer and another source of baffling errors; at five queries it doesn't pay for itself. At fifty, the answer would be different.

CHEAT SHEET

- If programs use the API, the main requirement is predictable codes.
- List the codes in the mini-spec: without that you get 200 for everything.
- A 500 is always a bug: a case wasn't allowed for.
- A duplicate is 409, nothing to delete is 404, malformed data is 422.
- A rule in the table's schema catches the duplicate, but the response code is the handler's responsibility.
- Data validation is written once in the schema, not in every handler.
- Acceptance is running through every line of the mini-spec, unhappy cases included.
- What isn't in the project should be a decision, not forgetfulness.

14.3 Case — a React site for that API

works in curl, doesn't work in the browser

WHAT IT IS

The third case continues the second: the API from article 14.2 is already running, and now it needs a website. The task looks simple — show a list — and it is exactly where an error surfaces that is in no React guide, because it isn't about React.

STEP 1. THE REQUEST

BETTER

```
React 19 + TypeScript, built with Vite.
One screen: a list of cities with temperatures.
Data: GET http://localhost:8102/cities,
the response [{name: string, temp: number | null}].

All four states are needed: loading, error, empty list,
data. Describe the response type explicitly, don't use
any or as.
Simple styles, no component libraries.
```

The requirement about four states comes from article 10.5, and it isn't there for appearances: it is what saves us at the next step.

STEP 2. WHAT ARRIVED

```
type City = { name: string; temp: number | null }

export default function App() {
  const [cities, setCities] = useState<City[]>([])
  const [error, setError] = useState<string | null>(null)
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    fetch('http://localhost:8102/cities')
      .then((r) => {
        if (!r.ok) throw new Error(`HTTP ${r.status}`)
        return r.json()
      })
      .then(setCities)
      .catch((e) => setError(String(e)))
      .finally(() => setLoading(false))
  }, [])

  if (loading) return <p className="state">Loading…</p>
  if (error) return <p className="state err">Couldn't load: {error}</p>
  if (cities.length === 0) return <p className="state">No cities yet</p>

  return (
    <main>
      <h1>Weather</h1>
      <ul>
        {cities.map((c) => (
          <li key={c.name}>
            <span>{c.name}</span>
            <b>{c.temp === null ? 'no data' : `${c.temp}°`}</b>
          </li>
        ))}
      </ul>
    </main>
  )
}
```

The code is right. The effect's dependencies are empty, the `key` is in place, the four states are there, there is no `any`. It builds:

```
✓ built in 1.82s
```

STEP 3. A BUG THAT ISN'T IN THE CODE

Open it in the browser:

```
Couldn't load: TypeError: Failed to fetch
```

Meanwhile the API is alive and answering perfectly well:

```
$ curl -s -o /dev/null -w "%{http_code}" localhost:8102/cities
200
```

Works in `curl`, doesn't work in the browser. The reason is in the developer console (article 8.4), and only there:

```
Access to fetch at 'http://localhost:8102/cities' from origin
'http://localhost:5190' has been blocked by CORS policy:
No 'Access-Control-Allow-Origin' header is present on the requested
resource.
```

This is **CORS** — the browser rule that a page at one address can't read responses from another unless that other one explicitly allows it. `curl` has no such rule: it exists only in the browser and only to protect the user.

Two conclusions that make this article worth its place.

First: the error on the screen doesn't name the cause. The browser deliberately withholds the details from the page's code — `fetch` gets only `TypeError: Failed to fetch`. You have to look in the console, not in the React code. Requests of the form "fix TypeError: Failed to fetch" are useless: there is no error in the code you showed.

Second: the fix belongs on the server. CORS is configured where the data comes from:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5190"],   # the site's address, not
"*"
    allow_methods=["GET", "POST", "DELETE"],
    allow_headers=["Content-Type"],
)
```

After restarting the API the header appears:

```
$ curl -D- -H "Origin: http://localhost:5190" localhost:8102/cities
access-control-allow-origin: http://localhost:5190
```

And the page comes to life — no errors in the console:



Fig. 14.1 · Three cities came from article 14.2's API. The screenshot was taken in the browser after CORS was configured.

> **CAREFUL**
> Almost any AI asked about CORS will propose `allow_origins=["*"]`. It works, and for a learning project it is tolerable, but it means "my responses may be read from anywhere". If the API serves anything beyond public data, the addresses are listed by name. The difference between `"*"` and a list is the difference between "I have nothing to hide" and "I didn't think about it".

STEP 4. WHAT TO CHECK WITH YOUR EYES

A built site isn't the same as a site in development (chapter 12). The minimum checks before calling this case closed:

- **The four states.** Switch the API off — the error line should appear, not an empty screen. That was the check at step 3, just an involuntary one.
- **A narrow window.** Shrink it to phone width.
- **The built version.** `vite build` and open the result, not only the development mode.
- **The API address isn't hard-wired.** It is different on the server; so an environment variable — and remember it ends up in the artifact in plain text (article 12.1).

CHEAT SHEET

- “Works in curl, doesn’t work in the browser” is almost always CORS.
- The real cause is visible only in the browser console, not on the screen.
- `TypeError: Failed to fetch` is not an error in the React code.
- CORS is fixed on the server that serves the data, not on the site.
- `allow_origins=["*"]` isn’t a solution but a refusal to decide.
- The four screen states paid for themselves on the very first day.
- Check the built version and a narrow window, not just development mode.

14.4 Case — a mobile application

the same API, a different conversation

WHAT IT IS

A fourth consumer of the same API: a Flutter application. In substance the screen is the one from article 14.3 — a list of cities with temperatures — but the conversation with the AI comes out different, and the differences are worth seeing side by side.

> **CAREFUL**
> Like chapter 11, this case is written at the level of reading. There are no listings and no measurements here — there is planning, there are requests, and there is an account of what will have to be checked in the answers. This case’s code is the one place in the book you have to check yourself.

STEP 1. WHAT CHANGES IN THE MINI-SPEC

Three lines compared with article 14.2 — and each one drags work behind it:

The API’s address. `localhost` in an application on a phone points at the phone itself, not at your machine. During development you need the machine’s address on the local network; in production, a real domain. It is the first thing people trip over, and an AI doesn’t warn you because it doesn’t know where you are running things.

The network drops. For a website that is rare; for a phone it is normal (article 11.5). Which means the last data has to be kept on the device and shown with a “updated at such-and-such” note, rather than an empty screen.

Updates aren't instant. A website updates when the page is reloaded, an application when the user agrees to it (article 12.3). Which means the API can't break compatibility: the `temp` field stays `temp` forever.

STEP 2. THE REQUEST

```
BETTER
Flutter 3.38, Dart 3.10. State with provider 6,
don't add other state libraries.

One screen: a list of cities with temperatures.
Data: GET http://<address>/cities,
the response [{"name": String, "temp": double?}].

I need:
- a City model with fromJson, no casts,
  temp honestly nullable
- four states: loading, error, empty, data
- the API address in a setting, not hard-wired
- the layout checked at 360 wide and with a long name

The list of files and what's in each first. Code after
"yes".
```

Compare it with the request from article 14.3: half the lines match word for word. Four states, explicit types, the address in a setting — these aren't properties of React or Flutter but properties of **the task "show data from the network"**. Which is why a good request carries over between stacks almost in full.

Three things differ, and all three are about the platform: the versions (here you must name them), the state library, and checking on a narrow screen.

STEP 3. WHAT TO CHECK IN THE ANSWER

Since it can't be run, what is left is what can be read. In order of importance:

1. The API version. The stack's main problem (article 11.2). Outdated widget and parameter names; `flutter analyze` usually finds them.

2. `!`, `late`, `dynamic` in the JSON parsing. The server can answer with an error, and `json['temp'] as double` will kill the application. The model should have an honest `double?` and a check, not a cast.

3. The four states — are they all there? The most frequent: "the data arrived" exists, the rest doesn't.

4. Requests in `build`. Going to the network belongs to the data layer, not to the build method (article 11.4).

5. `dispose` on the controllers. Forgotten cleanup.

6. Fixed sizes. Overflow on a narrow screen.

That is the same list as in article 11.6 — here it is simply applied to a specific task.

STEP 4. A CONVERSATION WORTH STARTING EARLY

PROMPT

```
I'm going to release this application to a store.
List what has to be decided before the first release and
what can't be changed after it.
```

The answer to that question is taken apart in article 12.2, and it is exactly why the mobile case is worth planning longer than the others: a website is redone in an evening, and an application identifier never is.

STEP 5. AN HONEST CONCLUSION

Of the API's four consumers (the bot, the website, the application, the script) this one is the most expensive: its own stack, its own build, its own release process, no possibility of rolling back. And its screen is the same as the website's.

The question worth asking yourself before starting rather than after: **do you need this application, or is a website open on a phone enough?** For a list of cities with temperatures the answer is usually "the website is enough". An application is justified when you need something a website doesn't have: working without a network, notifications, the camera, background work.

And that is the architectural decision from chapter 13 — the very one an AI will take for you if you don't take it yourself.

CHEAT SHEET

- A good request carries over between stacks almost word for word.
- Only the platform-specific part differs: versions, state library, narrow screen.
- `localhost` on a phone is the phone itself, not your machine.
- A mobile application loses its network constantly: it needs the previous data.
- Old versions of an application live a long time — the API can't break compatibility.
- Read the answer against the list in article 11.6: version, `!/late`, states, build.
- Before a mobile case, answer honestly whether you need it: a website is cheaper in every way.

14.5 Case — an evening's script

the most dangerous genre

WHAT IT IS

The last case is the smallest and the most dangerous. A twenty-line task: **sort out a folder of photos by renaming the files after the date they were taken**.

The danger with scripts like this isn't their complexity but that nobody takes them seriously. No mini-spec is written, no tests, no review — "it's only a rename". And it is run straight against the real folder.

STEP 1. WHAT ARRIVED

DON'T DO THIS

```
Write a python script that renames all the files in the
photos folder by modification date, format YYYY-MM-DD.
```

```python
import os
from datetime import datetime
from pathlib import Path

folder = Path("photos")

for f in folder.iterdir():
    date = datetime.fromtimestamp(f.stat().st_mtime)
    new_name = date.strftime("%Y-%m-%d") + f.suffix
    f.rename(folder / new_name)
```

Nine lines, readable in one go, doing exactly what was asked. It is hard to imagine what could break here.

STEP 2. WHAT IT DID

The folder held four files:

```
IMG_1001.jpg  "first photo"    taken 14 March
IMG_1002.jpg  "second photo"   taken 14 March
IMG_1003.jpg  "third photo"    taken 15 March
notes.txt     "not a photo"
```

After running it:

```
2026-03-14.jpg  "second photo"
2026-03-15.jpg  "third photo"
2026-03-15.txt  "not a photo"
```

There are three files now. `first photo` is nowhere.

Two photos from the same day got the same name, and the second quietly wrote over the first: `rename` in Python doesn't ask permission and doesn't warn. `notes.txt`, which nobody asked it to touch, was renamed along the way.

No error, no warning, no exit code. The script ran successfully.

> **REMEMBER**
> A script that changes files **has no undo**. No history like git's, no recycle bin, no "previous version". So the cost of a mistake here is higher than in an application, even though there is twenty times less code (article 1.5).

STEP 3. HOW IT SHOULD HAVE BEEN ASKED

BETTER

```
A Python 3.12 script: rename the photos in a folder by the
file's modification date into the format YYYY-MM-DD.

Required:
- by default change NOTHING, only show the plan
  "old name -> new"; carry it out on an --apply flag
- if several files share a date: -2, -3 and so on
- if a target name is already taken by a file not in this
  list — stop and do nothing
- touch only .jpg/.jpeg/.png/.heic, skip the rest and list
  what was skipped
- take the folder as an argument, don't hard-wire it
```

Six lines of requirements for nine lines of code. It looks excessive right up until the first lost photograph.

The first is the important one. **A preview mode by default** turns a dangerous script into a safe one: first you see what will happen, then you allow it.

STEP 4. WHAT CAME OUT

```
$ python3 rename.py photos
IMG_1001.jpg -> 2026-03-14.jpg
IMG_1002.jpg -> 2026-03-14-2.jpg
IMG_1003.jpg -> 2026-03-15.jpg
skipped (not a photo): notes.txt

nothing changed. files to rename: 3
repeat with --apply to carry it out
```

The whole plan is visible, and so is the main thing: the two files from 14 March get **different** names. Now it can be allowed:

```
$ python3 rename.py photos --apply
...
renamed: 3
```

The result:

```
2026-03-14.jpg    "first photo"
2026-03-14-2.jpg  "second photo"
2026-03-15.jpg    "third photo"
notes.txt         "not a photo"
```

Four files went in, four came out. Nothing was lost, `notes.txt` wasn't touched.

STEP 5. A RULE FOR THE WHOLE GENRE

Scripts that change something outside themselves — files, a database, emails, messages — differ from other code in one way: **a mistake can't be rolled back**. So there is a short list for them worth applying every time:

1. **A preview mode by default.** Execution on an explicit flag.
2. **On a copy first.** `cp -r photos photos-trial` takes a second.
3. **Never overwrite silently.** Name clash — stop.
4. **Filter by type.** "All the files in the folder" is almost never what was meant.
5. **A report at the end.** How many processed, how many skipped and why.
6. **The folder is an argument, not a constant.** One day someone forgets to change the constant.

Not one item takes any effort. All six together are the difference between a tool and a game of roulette.

CHEAT SHEET

- The most dangerous code is the shortest: nobody takes it seriously.
- `rename` overwrites silently: two files with one date means one file fewer.
- A script that changes files has no undo. No history, no recycle bin.
- A preview mode by default is the main protection, and it costs one line of requirements.
- “All the files in the folder” almost never means what you meant.
- The first run goes against a copy of the folder, not the real one.
- Six lines of requirements for nine lines of code is a normal proportion.

Appendices

The reference part. You look things up here rather than reading it straight through: the request templates in one place, the review checklist, the common errors decoded, git and terminal commands, a glossary of the English terms.

A Request templates

everything in the book, in one place

WHAT IT IS

The templates from chapters 3, 13 and 14, gathered together. Put your own text in the angle brackets. Each one names the article where it is explained why it is built that way.

WORKING WITH CODE

A new function (article 3.4):

```
PROMPT
Project: <stack, versions>. File: <path>.
I need a function <name>: it takes <what>, returns <what>.
Edge cases: <empty / network error / no data>.
Constraints: don't add new libraries, match the file's style.
Show only the changed fragment.
```

Fix a bug (article 3.5):

```
PROMPT
What I do: <steps>.
What I expect: <result>. What I get: <result>.
The error in full: <text with the traceback>.
Versions: <language, libraries>.
Tell me the cause first. The fix after I say yes.
```

Refactoring (article 3.6):

```
PROMPT
Here is the code: <fragment>.
The behaviour mustn't change. I need: <what exactly to improve>.
Don't touch: <files, functions>.
Show a diff rather than the whole file, and explain each change in one
line.
```

Explain this code (article 3.7):

PROMPT
```
Explain this code line by line, to someone who knows the
basics but doesn't know this library.
Separately: what could break here and with what data.
```

Review (article 3.8):

PROMPT
```
Review this: <code>.
Look for: unhandled errors, edge cases, leaked secrets,
unnecessary dependencies, places where the behaviour isn't obvious.
Format: a list, one line per finding, with the line number.
Don't rewrite the code.
```

Tests (article 3.9):

PROMPT
```
Write tests for <function/module>, pytest.
Cover: the ordinary case, the edges, empty input, an error.
No mocks where they can be avoided.
One assertion per test, and the test's name describes the case.
```

TECHNIQUES

The plan first (article 3.10):

PROMPT
```
The task: <description>.
Don't write code. The plan first: which files you'll touch,
what changes in each, in what order.
Code after I say yes.
```

Let the AI ask the questions (article 3.11):

PROMPT
```
Before writing any code, ask me the questions whose answers
would change the solution. One at a time.
If you have no questions, say what assumptions you're making.
```

Limit the edit zone (article 3.13):

PROMPT
```
Change only the file <path> and only the function <name>.
Don't touch anything else. If the solution needs something
else touched, say so, but don't do it.
```

THE WHOLE PROJECT

The ten questions (article 13.6):

PROMPT
```
The idea: <two or three sentences>.
Ask me one question at a time and wait for the answer:
what it does in one sentence; what is stored; where it's stored;
who uses it and how many; what can't be lost;
```

```
external services and behaviour when they fail; what changes most;
environments and secrets; starting and restarting; how I find out it
broke.
Don't write code. At the end, gather the answers into a mini-spec.
```

Breaking it into parts (article 13.1):

```
PROMPT
The idea: <description>. Don't write code. Break it into parts:
1) what is stored — entities with their fields
2) what happens — actions
3) what's visible from outside — screens or addresses
For each action, say which entities it touches.
```

The architecture brief (article 13.5):

```
PROMPT
Here is the project tree and the key files.
Assemble a draft brief in six blocks: what it is, stack,
structure, dependencies, where things live, what not to do.
Where you aren't sure, write "?" — don't invent.
```

Checking code against the brief (article 13.5):

```
PROMPT
Here is the brief and the code you wrote.
Check your code against the brief: dependencies,
file locations, the ban on new libraries.
Violations as a list.
```

BEFORE A RELEASE

What breaks in the build (article 12.1):

```
PROMPT
The project builds for me but fails after installation.
List how running in development differs from the artifact
in this project: paths, environment variables, assets, versions.
One point per line.
```

A fix with its price (article 12.1):

```
PROMPT
The build fails: <error>. I want two options:
a code fix and a build-settings fix.
For each, say what will stop being checked.
I'll choose.
```

Compatibility on an update (article 12.3):

```
PROMPT
I'm changing <a field / a schema / an endpoint>.
Old clients don't update immediately.
What breaks for them and how do I make this change without
breaking them? Separately: which part can't be rolled back.
```

A shipping checklist (article 12.4):

```
PROMPT
Draw up a shipping checklist for my project.
Stack: <list it>. Shipped to: <where>.
Split it into "once per project" and "before every release".
Only mine, no general advice.
```

B A code review checklist

twelve points on one screen

WHAT IT IS

The list you read any generated piece of code against before it goes into the project. The details are in chapters 2 and 4; here there is only the order.

FIRST, THREE QUESTIONS

1. **Do I understand what every line does?** If not, don't paste it — ask (article 2.1).
2. **Does this solve my problem or a similar one?** The commonest substitution: the problem the AI understood has been solved, not the one you set (article 3.1).
3. **What happens if this breaks?** The answer decides how much time to spend on the remaining points (article 1.5).

THE TWELVE POINTS

1. Errors. Is there handling where something could go wrong: the network, a file, parsing data, division. An empty `except` is a red flag.

2. Edge cases. An empty list, `None`, zero, a negative number, a very long string, two identical values.

3. External data is validated. An API response, user input, a file's contents — none of it is parsed without a check (article 7.3).

4. Secrets. No keys, passwords or tokens in the code. Not "temporarily", not "as an example" (article 5.7).

5. New dependencies. A library appeared that nobody asked for? Ask why. Often it can be left out.

6. The edit zone. Only what was asked for has been changed. Pieces rewritten "while we were there" are grounds for reverting and asking again (article 3.13).

7. Files and paths. Relative paths to data, hard-wired addresses and `/home/user/...` in the code (articles 12.1, 13.4).

8. Repetition. The same action has appeared in a third place — time to extract it.

9. Names. The names say what's inside. `data2`, `temp_fix`, `handle` usually mean the author didn't know either.

10. Layer dependencies. A database query from a screen's handler, an import pointing upwards (article 13.1).

11. What was deleted. In the diff, look not only at what was added but at the minuses: what disappeared along with the edit (article 2.13).

12. The checks are intact. Weren't the linter, the types or the tests switched off just to make it "pass" (article 2.14)?

THEN, RUN IT

Reading doesn't replace running, and running doesn't replace reading. The minimum:

- run it and walk through **your** scenario by hand;
- walk through the **unhappy** scenario: turn off the network, pass an empty value;
- run the tests if there are any, and read what they actually check (article 4.6);
- look at the whole of the changed files in `git diff` before committing.

> **REMEMBER**
> The main review question isn't "is this code correct" but **"will I be able to fix this in a month"**. Code that works but isn't understood is a debt, and you are the one who repays it.

C Common Python errors

what is written and what it means

WHAT IT IS

Real Python 3.13 messages — all obtained by running, not from memory. Read them from the bottom up: the last line says **what** happened, the lines above say **where** (article 2.12).

NAMES AND TYPES

`NameError: name 'cuty' is not defined` The name doesn't exist. Usually a typo, or a variable declared further down the file than it is used.

`AttributeError: 'str' object has no attribute 'push'` The object has no such method. A frequent sign that the AI mixed up languages: `push` belongs to arrays in JavaScript; Python lists have `append`.

`TypeError: can only concatenate str (not "int") to str` A string was added to a number. Almost always the value came from an environment variable or from input and stayed a string (article 13.4).

`TypeError: 'NoneType' object is not subscriptable` A key or index was used on `None`. Which means a function above returned `None` instead of data — and that wasn't checked.

`TypeError: f() missing 1 required positional argument: 'b'` Called with too few arguments. Often after a signature was edited and not every call site was updated.

`UnboundLocalError: cannot access local variable 'n' where it is not associated with a value` Inside a function, a value was assigned to a variable declared outside. Python treats it as local. The cure is passing it as an argument, not `global`.

DATA

`KeyError: 'temp'` The dictionary has no such key. The commonest error when parsing API responses: the key is there when all is well and disappears when the server returns an error.

`IndexError: list index out of range` An index past the end of the list. Check the list isn't empty — and remember the last element is `[-1]`, not `[len(xs)]`.

`ValueError: invalid literal for int() with base 10: '-2.5'` The string doesn't turn into a number. `int()` doesn't understand decimal notation — you want `float()` and then rounding.

`ZeroDivisionError: division by zero` Division by zero. Usually the denominator is the length of an empty list or a counter that hasn't had time to grow.

`json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)` Something that isn't JSON was parsed as JSON: an empty string, an

HTML error page, a truncated file. What to print is the raw response text, not the parsed data.

`UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd8 in position 96`
A binary file was opened as text, or text in another encoding. For binary, the `"rb"` mode; for text, an explicit `encoding=`.

FILES AND MODULES

`FileNotFoundError: [Errno 2] No such file or directory: 'cities.json'`
The file isn't there **where the program was started from**. A relative path is counted from the working directory, not from the file with the code (article 12.1).

`ModuleNotFoundError: No module named 'rekvests'` The library isn't installed, or the name is misspelled. If the name looks strange, it may not exist at all (article 4.8).

`ImportError: cannot import name 'x' from partially initialized module 'y' (most likely due to a circular import)`
Two modules import each other. That isn't a Python error but a signal of mixed-up layers (article 13.1).

SYNTAX AND EXECUTION

`IndentationError: expected an indented block after function definition on line 1`
The indentation was lost. Usually when pasting code in pieces.

`SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?`
One equals sign instead of two in a condition. Python says so right in the error text.

`RecursionError: maximum recursion depth exceeded` A function calls itself with no stopping condition. The traceback shows `[Previous line repeated 996 more times]`.

`sqlite3.IntegrityError: UNIQUE constraint failed: cities.name` A table rule was broken: a record with that value already exists. That isn't a breakage but a guard doing its job — catch it and return a clear response (article 14.2).

> **TIP**
> When handing an error to an AI, copy the **whole** traceback, not the last line. The last line says what broke; the lines above say what path led there, and that is what the cause is found from.

D Common JS, TypeScript and Dart errors

three languages, the same rakes

WHAT IT IS

The JavaScript and TypeScript messages were obtained by running (Node 22, TypeScript 5.8). The Dart messages are given from memory and were not run — like the whole of chapter 11, they need checking against the documentation.

JAVASCRIPT AT RUN TIME

`TypeError: Cannot read properties of undefined (reading 'name')` The most frequent error in the whole frontend. A field was read from something that isn't there: the data hasn't arrived yet, the key is called something else, the array is empty. The cure is a check or `?.`, but first you have to understand **why** it is `undefined` there.

`TypeError: Cannot read properties of undefined (reading 'map')` The same thing in React: a list was rendered before the data arrived. A sign that the "loading" state wasn't built (article 10.5).

`TypeError: Cannot set properties of null (setting 'textContent')` The element wasn't found. The script ran before the browser built the page, or the selector is wrong.

`TypeError: xs.push2 is not a function` The method doesn't exist. A typo — or an invented method (article 4.8).

`TypeError: d is not iterable` Iterating over something that isn't an array: `undefined`, an object, `null`.

`TypeError: Assignment to constant variable.` A value was assigned to a `const`. Changing an object's contents is allowed, reassigning the variable isn't.

`ReferenceError: cuty is not defined` The name doesn't exist. Unlike in Python, in the browser this sometimes means the script didn't load.

`SyntaxError: Unexpected end of JSON input` `JSON.parse` on an empty string. Usually the server's response is empty because the request never arrived.

`SyntaxError: await is only valid in async functions and the top level bodies of modules`
An `await` in an ordinary function. Either make the function `async` or work through `.then`.

`TypeError: Failed to fetch` The request didn't happen. There is no error in the code: the real cause is in the browser console, most often CORS (article 14.3).

TYPESCRIPT AT CHECK TIME

```
error TS2741: Property 'temp' is missing in type '{ name: string; }' but required in type 'City'
```

A field is missing. Either the field should be optional (`temp?`), or it really was forgotten.

`error TS2322: Type 'string' is not assignable to type 'number'` The value is the wrong type. The classic: a number that arrived as a string.

```
error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'
```

The same thing in a function argument.

`error TS18048: 'd' is possibly 'undefined'` The value may be absent and is being used directly. The right answer is a check, not a `!` (article 9.5).

`error TS18047: 'c.temp' is possibly 'null'` The same for `null`. It appears exactly where the types are described honestly — and that is a sign of good code, not of a problem.

`error TS2339: Property 'b' does not exist on type '{ a: number; }'` The field isn't in the type. Either a typo, or the type is described incompletely — and then the type is what needs fixing, not gagging it with `as any`.

DART AND FLUTTER

> **CAREFUL**
> The messages in this section are given by meaning rather than copied from a particular run. The meaning of the errors is stable; check the exact wording against your version.

`Null check operator used on a null value` A `!` was put where the value turned out to be empty. The direct analogue of `as` in TypeScript and the same kind of gag (article 11.2).

`LateInitializationError: Field 'x' has not been initialized` The field was declared `late` and nobody remembered to assign it.

`setState() called after dispose()` The widget is already gone and a network response arrives and tries to update the screen. A sign the subscription wasn't closed in `dispose` (article 11.4).

`RenderFlex overflowed by N pixels` That yellow-and-black stripe: the content doesn't fit. The cure is `Expanded` or `Flexible` (article 11.3).

`The method 'X' isn't defined for the type 'Y'` The method isn't there. In Flutter that most often means **code for another version**: the stack's main disease.

WHAT THEY HAVE IN COMMON

Three quarters of the list is one and the same error in three languages: **something that isn't there was used**. `None`, `null`, `undefined`, a missing key, an empty array.

So the technique is the same in every stack: not "add a check where it crashed" but go higher and answer **why it is empty there** — and handle it where the data appears.

CHEAT SHEET

- Read the last line: it says what happened.
- `undefined (reading '...')` — the data hasn't arrived or the key is different.
- `Failed to fetch` — the cause is in the browser console, not in the code.
- `possibly 'undefined'` in TypeScript is a sign of honest types, not a problem.
- `!` in Dart and `as` in TypeScript are the same gag.
- Nearly everything in the list is one error: using something that isn't there.

E Git — rescue scenarios

what to type when everything is broken

WHAT IT IS

Not a git course but a list of situations. Every command was run in a real repository, and the output is real. The basics of git are in article 5.1.

EVERY DAY

```
git status            # what is changed right now
git diff              # what exactly is changed, line by line
git diff --staged     # what will go into the commit
git add .             # take everything changed
git commit -m "..."   # record it
git log --oneline     # the history as a short list
```

Before asking an AI for a large edit, it is worth making a commit. That is one command, and it turns any failure into "roll back and try differently".

I BROKE A FILE AND WANT IT BACK AS IT WAS

```
$ git diff --stat
 app.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git restore app.py      # restore one file
$ cat app.py
third version
```

To bring **everything** back to the last commit:

```
git restore .
```

> **CAREFUL**
> `git restore` throws away uncommitted changes **with no way back**. That isn't the same as reverting a commit: a commit can be found in the history, an unrecorded edit can't.

I DELETED A FILE

```
$ rm app.py
$ git restore app.py
$ ls
app.py
```

As long as the file was in the last commit, it comes back.

I COMMITTED THE WRONG THING

The commit hasn't been pushed — undo the commit, keep the changes:

```
$ git reset --soft HEAD~1
$ git status --short
M  app.py
```

The changes are there, the commit isn't. You can fix it and commit again.

The commit has been pushed — don't rewrite history, add an opposite commit instead:

```
$ git revert --no-edit HEAD
[main 2127961] Revert "third version"
 1 file changed, 1 insertion(+), 1 deletion(-)

$ cat app.py
second version
```

`revert` creates a new commit that undoes the old one. The history stays truthful and nothing breaks for anyone.

I URGENTLY NEED TO SET MY CHANGES ASIDE

```
$ echo "draft" >> app.py
$ git stash          # hide them and return to a clean state
$ cat app.py
second version

$ git stash pop      # take them back out
$ cat app.py
second version
draft
```

Handy when an AI has done half the work and you urgently need to check something else.

I COMMITTED A SECRET

```
$ git rm --cached .env
$ echo ".env" >> .gitignore
$ git add .gitignore && git commit -m "removed .env from the repository"
$ git ls-files
.gitignore
app.py
```

`.env` is gone from the working copy and from future commits. **But:**

```
$ git log --oneline --all -- .env
c07fe2e removed .env from the repository
1736a45 added .env
```

The file is still in the history, and the key comes out of it with one command. So the order of operations is: **change the key first**, and clean the repository afterwards. If the repository is public, treat the key as compromised from the moment it was pushed (article 5.7).

WHEN IT BROKE — AND IT USED TO WORK

```
git log --oneline              # find the suspicious commit
git show <hash>                # see what changed in it
git restore --source=<hash> <file>   # take the file's version from that
commit
```

If there are many commits and it isn't clear which is to blame, there is `git bisect`, which halves the history (the technique from article 4.4, only automatic).

BRANCHES

```
git switch -c trial    # a new branch from where you are
git switch main        # go back
git merge trial        # bring the work in
git branch -d trial    # delete the branch
```

For working with an AI a branch is useful in one specific case: **a large rework** that you can't tell in advance will survive to the end. In a branch you can abandon it without clearing up after yourself.

I DID RESET --HARD AND LOST A COMMIT

A commit that "disappeared" after `reset` has not been deleted yet. Git remembers where `HEAD` stood every time:

```
$ git reset --hard HEAD~1
$ cat app.py
first version
$ git reflog
ee9fbbe HEAD@{0}: reset: moving to HEAD~1
585928f HEAD@{1}: commit: second version
ee9fbbe HEAD@{2}: commit (initial): first version
$ git reset --hard HEAD@{1}
$ cat app.py
second version
```

`reflog` is the last line of defence: it keeps entries for several weeks. But it only rescues what was committed. Uncommitted changes after `reset --hard` cannot be brought back by anything — one more reason to commit before every task you give an AI.

WHAT NOT TO DO

`git push --force` to a shared branch. It rewrites the history for everyone.

`git reset --hard` without understanding it. It throws away both commits and edits.

An "everything at once" commit once a week. Such a commit can't be partially reverted — and that is exactly what you need when an AI broke one of twenty changes.

Committing without reading the diff. `git diff` before a commit is thirty seconds that catch a debug print left in by accident and a forgotten key.

F The terminal — the necessary minimum

twenty commands that are enough

WHAT IT IS

As much terminal as you need to run your own programs, read logs and understand what an AI is proposing. The examples are Linux and macOS; on Windows the same things are available through PowerShell or WSL.

WHERE AM I AND WHAT'S HERE

```
pwd                 # which folder am I in
ls                  # what's here
ls -la              # everything, hidden files included, with sizes and
dates
cd project          # go into a folder
cd ..               # up one level
cd                  # home
```

Hidden files are the ones starting with a dot: `.env`, `.gitignore`. Without `-la` you can't see them, and that is exactly why they get forgotten.

LOOK AT A FILE

```
cat app.py          # the whole thing
head -20 app.py     # the first 20 lines
tail -20 app.py     # the last 20
tail -f logs/app.log  # the last ones, then live as they arrive
wc -l app.py        # how many lines
```

`tail -f` is the key command when debugging a running program: leave it in a separate window and watch what gets written while you press buttons.

FIND

```
$ grep -rn "error" logs/
logs/app.log:1:error
logs/app.log:3:error again

$ find . -name "*.py"
./app.py

$ grep -c "error" logs/app.log
2
```

`grep -rn` is recursive, with line numbers. It is the command you use to find where a function is used in the project, where a setting is read from, and whether a key got into the code:

```
grep -rn "sk-\|password\|api_key" .
```

FILES

```
cp file copy            # copy
cp -r folder copy       # copy a folder
mv old new              # move or rename
rm file                 # delete, with no recycle bin
mkdir -p a/b/c          # create folders, intermediate ones included
du -sh .                # how big a folder is
```

> **CAREFUL**
> `rm` has no recycle bin and no confirmation. Before running an `rm` with a wildcard, or one out of somebody else's script, replace it with `ls` and the same argument and see what would be deleted. That is the same thought as article 14.5: show first, act afterwards.

RUNNING AND STOPPING

```
python3 app.py          # run it
Ctrl+C                  # stop a running program
Ctrl+D                  # end input (leave python, leave ssh)
./script.sh             # run a script
chmod +x script.sh      # allow it to be run
```

If `./script.sh` answers `Permission denied`, the `chmod +x` is missing.

CHAINING COMMANDS

```
command1 && command2    # the second only if the first succeeded
command1 | command2     # the first's output into the second's input
command > file          # output into a file (overwrite)
command >> file         # append to the end
command 2>&1            # errors to the same place as normal output
```

`&&` is what every build command is made of: `tsc -b && vite build` means "check the types, and only build if that worked" (article 12.1).

THE EXIT CODE

```
$ python3 -c "import sys; sys.exit(3)"
$ echo $?
3
```

`$?` is how the last command ended. Zero means "successfully", anything else doesn't. Both `&&` and the checks in a build run on this: a program can print "done" and still return an error.

ENVIRONMENT VARIABLES

```
PORT=8080 python3 app.py      # set for this run only
export API_KEY=abc            # set for the whole terminal session
echo $API_KEY                 # look at it
env | grep API                # list the matching ones
```

A variable set with `export` lives until the window is closed and doesn't land in any file — which is what makes it good for keys (article 13.4).

WHAT ELSE COMES IN HANDY

```
history | grep docker    # what did I type about docker
Ctrl+R                   # search the command history
curl -s -o /dev/null -w "%{http_code}\n" localhost:8000/   # the
response code
df -h                    # how much disk space there is
ps aux | grep python     # which processes are running
kill <number>            # stop a process
```

The `curl` command from that list is worth memorizing: it answers "is my server alive at all?" without a browser and without guessing (article 14.3).

G Glossary

the jargon that turns up every day

WHAT IT IS

The words that arrive in AI answers, in error messages and in documentation. Not a dictionary but a list of what gets in the way of reading until you know it.

CODE AND ITS PARTS

Word	What it means
variable	a named place a value is kept in
function, method	a named piece of code; a method is a function belonging to an object
argument, parameter	what you pass into a function; what it is called inside
return	hand a value back out of a function
class, instance	a class (the description); an instance (an object made from it)

property, field	a value belonging to an object
array, list	an ordered collection of values
dictionary, map, object	key–value pairs
string, integer, float, boolean	text, a whole number, a decimal, true/false
null, None, undefined	"nothing"; in JS also "no value was set"
scope	where a name exists
statement, expression	an instruction; something that has a value
loop, iterate	repeat; go through the elements one at a time
nested	one thing inside another
mutable, immutable	changeable; unchangeable

WHAT HAPPENS WHILE IT RUNS

Word	What it means
runtime	the time the program is running (as opposed to the build)
exception, throw, raise	an error object; sending it up out of the code
catch, handle	intercept an error and deal with it
stack trace, traceback	the chain of calls leading to the crash
debug	look for the cause of an error
log	a record of an event; "the logs" are those records
crash	the program dies
hang, freeze	the program stops responding
race condition	two processes interfering with each other
memory leak	memory taken and never given back
timeout	a limit on how long to wait
async, await, promise	asynchrony: "wait for the result without blocking everything"
thread, process	two ways of running things at the same time
cache	stored answers kept so they needn't be computed again

DATA AND THE NETWORK

Word	What it means

API	the way one program talks to another
endpoint	one specific address of an API
request, response	what is sent; what comes back
payload, body	the contents of a request or a response
header	a line of metadata on a request or a response
status code	the response code: 200, 404, 500
JSON	a text format for data: pairs and lists
schema, validation	a description of data; checking against it
parse, serialize	turn text into data; turn data into text
query	a request to a database
index	a pointer in a database that speeds up searching
migration	a change to the database's structure
transaction	a group of changes, all or nothing
CRUD	create, read, update, delete
CORS	the browser rule about requests to another address

FRONTEND

Word	What it means
frontend, backend	what the user sees; what runs on the server
DOM	the tree of a page's elements in the browser
component	a piece of interface with its own code
props	data passed into a component from outside
state	data a component holds itself
render	draw, build the interface
hook	a React function named `useSomething`
effect	something done on the side: a request, a subscription, a timer
layout	how the elements are arranged
responsive	adapting to the width of the screen
viewport	the visible area of the screen
bundle	the built file holding the page's whole code

PROCESS AND TOOLS

Word	What it means
repository, repo	a project with its history of changes
commit	one recorded change
branch, merge	a parallel line of work; bringing it back together
diff	the difference between versions
pull request	a proposal to merge changes in
rollback, revert	going back to how it was
dependency	somebody else's library your code needs
lock file	a file with the exact versions of the dependencies
environment	the set of settings a program runs with
build	turning sources into something that runs
artifact	what comes out of a build
deploy	putting it on the server
rollout	releasing a new version gradually
linter	a program that looks for suspicious places in code
refactoring	reworking code without changing its behaviour
legacy	old code that is frightening to change
technical debt	deferred decisions, and their interest

ABOUT AI

Word	What it means
prompt	a request to the model
context, context window	what the model can see now; how much fits
token	the chunk of text the model counts volume in
hallucination	something invented: a function, a link, a fact that doesn't exist
temperature	how varied the answers are
fine-tuning	further training a model on your own data
agent	a mode in which the model runs commands itself
system prompt	a standing instruction set before the conversation

> **TIP**
> If an unfamiliar word turns up in an AI's answer, ask right there: "what is a `race condition` in this code, in one paragraph". That is cheaper than looking it up separately, and the explanation will be about your code rather than in general.

Index

Page numbers point to where the term is explained or put to work.

.env 24, 141, 142, 143, 144, 298, 337
acceptance checklist 114, 115, 117, 189, 204, 224
agent 58, 128, 129, 130, 131, 132, 142
anti-patterns in requests 258
any (TypeScript) 118, 157, 165, 215, 223, 225, 298
API 6, 265, 308, 315, 318, 319, 320
architecture 283, 285, 287, 294, 300, 301, 302
artifact 263, 265, 266, 268, 279, 280, 281
asynchrony 41, 42, 43, 44, 158, 161, 162
backend 167, 169, 170, 176, 180, 182, 188
backup 186, 188, 272, 278, 293, 295, 304
box-sizing 199, 200
branch 111, 117, 217, 251, 338, 344
build 228, 263, 264, 268, 269, 275, 279
cache 40, 55, 64, 93, 275, 277, 278
circular import 47, 151, 285, 286, 287, 332
class 33, 34, 35, 154, 194, 197, 230
commit 125, 126, 127, 128, 336, 337, 338
component 58, 226, 231, 232, 234, 235, 343
config 7, 148, 150, 151, 226, 295, 299
context file 58, 61, 90, 91, 92, 135, 163
context window 131, 133
CORS 317, 318, 319, 334, 343
CSS 191, 192, 196, 198, 200, 202, 252
Dart 244, 246, 247, 248, 249, 250, 258
database 177, 179, 184, 185, 190, 293, 295
dataclass 34, 40, 152, 153, 154, 157, 301
decorator 19, 38, 39, 40, 41, 61, 72
dependencies 44, 47, 90, 138, 139, 236, 237
deploy 143, 144, 145, 227, 275, 276, 277
developer tools 200, 201, 203, 204, 242
diff 51, 52, 53, 70, 72, 87, 89
DOM 191, 192, 193, 194, 201, 203, 227
dry run 323, 324, 325
endpoint 92, 313, 328, 343
environment variables 6, 20, 21, 54, 90, 141, 143
environments 90, 138, 140, 141, 143, 268, 299
exact versions 23, 140, 226, 228, 229, 267, 344
exception 37, 38, 159, 161, 164, 165, 168
f-string 153, 163, 181, 182, 189, 190, 248
FastAPI 40, 167, 168, 313, 314
Flutter 244, 245, 246, 247, 250, 260, 261
folder structure 287, 289, 290, 291, 302, 328
frontend 200, 202, 203, 204, 206, 228, 241
generics 216, 217, 218, 219, 249
git 124, 126, 127, 335, 336, 337, 338
hallucination 117, 119, 120, 164, 224, 344
HTML 191, 192, 193, 195, 227, 275, 277
HTTP 168, 170, 177, 179, 190, 316, 317
index in a database 183, 185
JavaScript 207, 208, 209, 210, 213, 214, 216
key in a list 143, 180, 231, 254, 271, 272, 331
layers 176, 177, 178, 179, 284, 286, 287
linter 209, 224, 225, 227, 237, 242, 243

migration 13, 64, 65, 185, 186, 187, 188
mini-spec 63, 64, 65, 92, 94, 95, 305
mutable default argument 29, 30, 55, 56, 164, 165, 166
mypy 156, 157, 158, 162, 165, 174, 216
N+1 queries 182, 183, 184, 185, 189, 190
pathlib 45, 152, 153, 154, 163, 266, 322
permissions 190, 258, 259, 261, 271, 272, 273
plan before code 302, 308, 327
PostgreSQL 184, 186, 293, 294, 295, 296, 298
print debugging 109, 111
props 230, 231, 232, 234, 235, 241, 242
pydantic 46, 168, 174, 222, 297, 299, 301
pytest 78, 79, 85, 112, 113, 114, 165
Python 148, 149, 151, 161, 163, 164, 165
React 58, 226, 227, 229, 241, 242, 315
red flags 54, 55, 56, 163, 329
refactoring 70, 71, 72, 73, 79, 95, 114
request template 326, 327, 329
response code 104, 170, 171, 172, 312, 313, 343
review 2, 4, 75, 76, 77, 276, 307
rollback 11, 123, 124, 125, 126, 128, 276
script that changes files 322, 323, 325
secrets 104, 106, 141, 142, 143, 265, 279
setState 234, 241, 242, 243, 255, 257, 262
shipping 274, 276, 277, 278, 280, 281, 329
signing key 245, 274, 278
SQL injection 181
SQLite 93, 291, 292, 293, 294, 295, 301
state 34, 135, 235, 237, 255, 256, 257
storage choice 64, 90, 293, 294, 295, 308, 309
Telegram bot 45, 46, 58, 61, 63, 139, 303
tests 78, 79, 80, 112, 113, 114, 121
timeout 102, 158, 159, 160, 212, 217, 310
token 21, 54, 140, 141, 142, 143, 188
traceback 47, 49, 50, 51, 68, 69, 70
type annotations 14, 25, 26, 75, 155, 157
TypeScript 213, 214, 215, 216, 224, 333, 335
useEffect 40, 235, 236, 237, 238, 241, 256
useState 232, 235, 237, 241, 316
validation 74, 172, 173, 176, 195, 196, 297
virtual environment 22, 90, 138, 139, 140, 144, 145
Vite 226, 227, 228, 264, 267, 268, 275
widget 250, 251, 252, 253, 257, 259, 260