

// two answers to the same question

Vibe Coding and Spec-Driven Development;

Where the phrase came from;

"There's a new kind of coding I call **vibe coding**, where you fully give in to the vibes, embrace exponentials, and **forget that the code even exists.**"

// Andrej Karpathy, February 2025

- ▶ He was describing a mode he was enjoying
- ▶ Not proposing a methodology

The one idea;

Both modes are correct. For **different tasks**. The skill is telling which task you are on — before you start, not afterwards.

- ▶ These two hours are an argument, not a verdict

The two hours;

Part 1 — the argument

- ▶ What vibe coding is genuinely good at, and what happened when the industry adopted it
- ▶ Comprehension debt and haunted codebases
- ▶ The counter-trend: spec-driven development
- ▶ Choosing a mode, deliberately

Part 2 — the mechanism

- ▶ What a spec changes about what the model is optimising for
- ▶ Plan-then-implement against prompt-run-repeat
- ▶ One feature, built both ways
- ▶ Measuring comprehension debt on code you wrote an hour ago

What it is genuinely good at;

- ▶ **Prototypes** — the point is to learn something, then delete it
- ▶ **Exploring an unfamiliar stack** — you do not know what to specify yet
- ▶ **Throwaway tooling** — a script you will run twice
- ▶ **Demos** — where "it works on stage" is the entire requirement

The pattern: being wrong is **cheap**, because the artefact is disposable.

Case 1: a to-do app in one prompt;

PROMPT

Build me a to-do app. I can add a task, mark it done, delete it, and the list survives a page refresh.

- ▶ A working app, in minutes, from one sentence — and the tools **are** this good
- ▶ You now own a file layout you did not choose, dependencies you did not pick, and lines you have not read

// the brief is deliberately small – it comes back in part 2

What you actually have;

It runs — you added a task and refreshed

checked

The list is stored somewhere, in a shape the model chose

unread

Whatever the user types goes from the form into that storage — validated?

unread

What the code does with input it was never demoed on

unknown

- ▶ Running is the **minimum**, not the evidence
- ▶ Every layer below the top was decided by the model and read by nobody

Then the industry adopted it;

US developers using AI coding tools daily	92%
...who say they trust the output	29%
...who always review before committing	48%
Major issues vs human-written code	1.7×
Samples with an OWASP Top-10 vulnerability	~45%
Sprint capacity on AI-traceable bugs, a quarter in	a fifth to a third

// treat these as direction, not decimal points

A word on those numbers;

- ▶ Industry surveys, varying rigour, heavily recycled
- ▶ The **direction** is not in dispute. The precision is fiction

Being able to say "I believe the trend, not the decimal" is itself an engineering skill.

Two words worth knowing;

Comprehension debt — the future cost of understanding code a machine wrote and nobody read

borrowed

Haunted codebase — a working system the team no longer understands

the bill

- ▶ Technical debt, but what you borrowed against is **understanding**
- ▶ AI did not invent this. It accelerated it

Predict: when does a vibe-coded project hurt?;

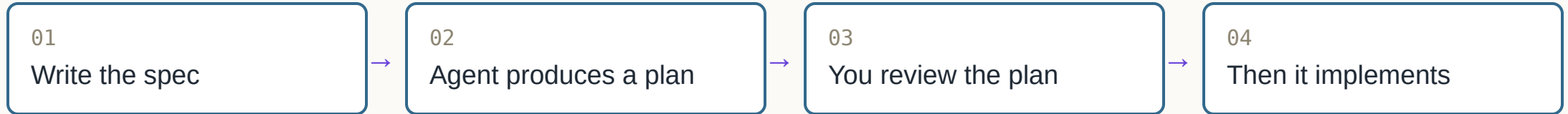
- ▶ Immediately — it does not work
- ▶ Day one is fine; the pain starts when you must **change** it
- ▶ Only if you picked the wrong tool
- ▶ It does not, if the tests pass

Day one is euphoric;

- ▶ Which is exactly the problem: the **feedback arrives long after the decision**
- ▶ You cannot feel comprehension debt accruing. You can only feel it arriving

Within a **quarter**, teams report a fifth to a third of sprint capacity going on bugs traceable to AI-generated code.

The counter-trend: spec-driven development;



- ▶ The artefact you review becomes the **intent**, not the implementation
- ▶ One page of spec is easier to judge than 800 lines of generated code

The two shapes;

Vibe coding: intent -> code -> hope

Spec-driven: intent -> spec -> plan -> tasks -> code -> check against spec

- ▶ Skip the plan review and spec-driven becomes vibe coding **with paperwork**

The honest objection;

"I'd rather review code than all these markdown files."

- ▶ Ceremony that **feels** like rigour is not rigour
- ▶ A spec nobody reads is worse than no spec: it manufactures confidence

Spec-driven has a real cost. If the task does not justify it, the cost is all you get.

Predict: a weekend prototype you intend to delete;

Which mode?

- ▶ Spec-driven — always the responsible choice
- ▶ Vibe coding — you do not know what you want yet
- ▶ Spec-driven, but a short spec
- ▶ Neither; write it by hand

Vibe coding – and this is not the lazy answer;

- ▶ You cannot specify what you have not yet understood
- ▶ Sometimes building the thing **is** the requirements gathering

The failure is not vibe coding. The failure is vibe coding something you then **keep**.

Choosing, deliberately;

	Vibe coding	Spec-driven
Stakes	Prototype, demo, throwaway	Production, shared, long-lived
Requirements	Discovering them	Known well enough to write
Being wrong costs	Delete and retry	Somebody maintains it for years
You review	The running app	The spec, then the tests

Predict: a large project, built over ten weeks;

- ▶ Vibe coding — speed matters, deadlines are real
- ▶ Spec-driven throughout — the stakes are high
- ▶ Vibe the exploration, specify what you keep
- ▶ It does not matter as long as it works

Both – and switching on purpose;

- ▶ Explore by building. Then **write the page** before you commit to it
- ▶ Mode is chosen **per task**, not per project

If you have to defend it in person, anything you cannot explain is something you should have specified.

Try the table on tasks it never listed;

Call a mode for each:

- ▶ A script to rename two hundred files, once
- ▶ A chart for tomorrow's meeting
- ▶ The page that stores other people's addresses
- ▶ The function that works out what a customer is charged
- ▶ A prototype to find out whether an idea is even possible
- ▶ Now the rename script again — but it runs **every month, on a shared drive**

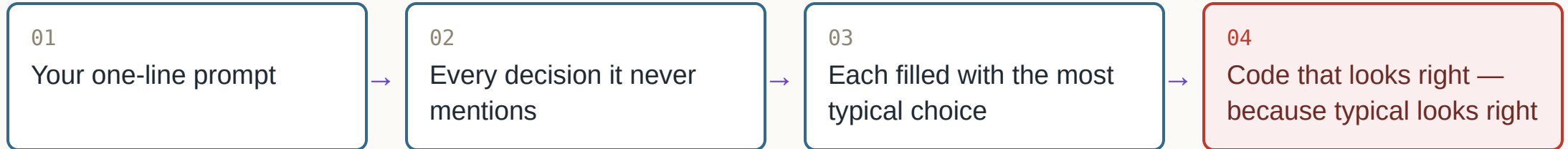
The table is a set of questions, not a list of task types. Change the answers and the mode moves.

// break

Ten minutes;

Part 2: what a spec actually changes, why patching is not planning, and how to measure what you did not read.

Every gap gets filled;



- ▶ The model produces the most plausible code **given what is in front of it**
- ▶ Every requirement you leave out is not left open. It is answered — by the training data, not by you

Underspecified is not unspecified. The decisions still get made — silently, and not by you.

Predict: a one-page spec instead of a one-line prompt;

What changes about the code?

- ▶ The model tries harder, so the code is better
- ▶ Nothing — the model mostly ignores long instructions
- ▶ The decisions you wrote down become yours; the rest are still the model's
- ▶ The style changes; the decisions do not

The model did not get smarter. The gaps got smaller;

Spec — does, does not do, the data's shape, acceptance criteria

decided by you

Plan — the model's reading of your spec, in prose

reviewable

Code — the plausible continuation of both

check against the spec

- ▶ A spec buys three things: decisions **you** made, something the output can be **wrong against**, and a check that is not "does it run"
- ▶ The **does not do** section bounds the work — leave it out and the typical continuation includes features nobody asked for

Try it now: count the gaps;

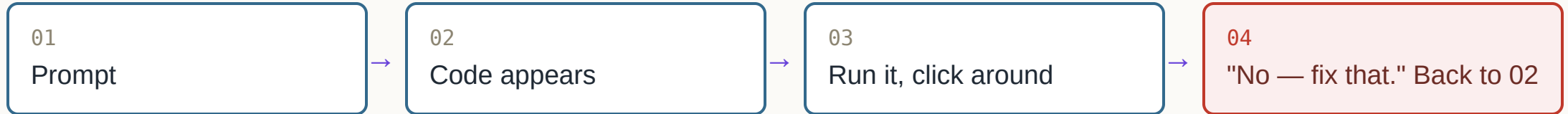
Seven minutes, your own laptop, whichever assistant you have.

PROMPT

I want to add a due date to each task in a to-do app that stores its tasks in the browser. Before writing any code, list every decision you would have to make that I have not specified. Number them. Do not write code.

- ▶ **Notice the length of the list.** Every entry would have been decided silently by "just add due dates"
- ▶ Pick the three you would have wanted to make yourself. Written as one line each, those are the start of a spec
- ▶ If it wrote code anyway, notice that too

Prompt, run, repeat;



- ▶ Every correction is a **patch on the previous guess** — the structure is whatever attempt one chose
- ▶ You correct **behaviour**, after the code exists — and only the behaviour you happened to click
- ▶ The conversation fills with wrong turns, and each new patch is generated on top of all of them

Predict: five rounds of "no, fix that" later;

What has happened to the structure of the code?

- ▶ It converged — each round improved the whole
- ▶ It is still whatever attempt one chose, with five patches on top
- ▶ The model refactored it as it went
- ▶ There is no structure to speak of

Plan first – what actually differs;

	Prompt, run, repeat	Plan, then implement
You correct	Behaviour, after the code exists	The model's reading of your intent , before any code
A correction costs	A patch on a patch	A sentence
What carries forward	Every wrong turn	The plan you approved
The structure is chosen by	Attempt one, by accident	The plan, on purpose
You know it is done when	It seems to work	It meets the criteria you wrote

// the plan is the model's interpretation, made visible while a fix is still cheap

Case 2: one feature, two routes;

The to-do app from case 1 exists and works. Add a **due date** to each task.

Route A — one prompt, then run it and see

vibe

Route B — half a page of spec, a plan, a review, then the code

spec-driven

- ▶ Same feature, same starting app, same assistant — only the route differs
- ▶ Watch the **decisions**, not the code. You listed them ten minutes ago

Route A: one prompt;

PROMPT

Add a due date to each task.

► It works: a fresh task takes a date and shows it. **Day one is euphoric**

Decided somewhere in the code	By
Stored as what — text, a number, a date object?	the model
Optional, or required?	the model
What happens to tasks saved before the feature?	the model — or nobody
Does the list now sort by it?	the model

Route B: half a page first;

Feature: due dates

Does: each task may carry a due date, shown beside the task.

Does not: reminders, notifications, time zones, recurring tasks.

Data: ``due`` is an optional text field on each task, as YYYY-MM-DD.

Tasks saved before this change have no ``due`` and must still load and display unchanged.

Acceptance:

1. A task saved with a due date still shows it after a page refresh.
2. A task saved before this change still appears, with no date.
3. A task saved with no due date can be given one later.

- Every line is a decision the model would otherwise have made. The **does not** line is the one that stops the feature growing

Route B: the plan – and the drift;

PROMPT

Here is spec.md. Give me a plan: which files you will change and what each change does. Do not write code yet.

REPLY

1. Add an optional due field when a task is created or edited.
2. Rewrite the storage format to the new shape; discard any saved entry that does not match it.
3. Show the date beside each task and sort overdue tasks to the top.
4. Check the three acceptance criteria.

- ▶ Step 2 **deletes every existing task** — criterion 2 says they must survive
- ▶ Step 3 adds sorting nobody asked for. Both are one sentence to fix — **here**

What each route produced;

	Route A	Route B
Time	Minutes	Longer — the spec is most of it
Decisions made by you	None	The ones in the spec; the rest still the model's
Tasks saved before the change	Whatever happened	Criterion 2 — checked
What you reviewed	The running app	Spec, then plan, then three criteria
What you can tell a reviewer	"It works"	Why each decision went the way it did

Route B is slower. The question is what the slowness bought — on a to-do app, maybe nothing; on the app that holds other people's data, criterion 2.

How the debt accrues;

The line you read: it saves the list

read

...which calls a helper that turns the list into text — how?

unread

...with a library you did not choose, on defaults you do not know

unread

...into a shape the model picked, which your due-date change must now preserve

unknown

- ▶ Each accepted-unread line is a small loan. They **compound**: the line you read means whatever the lines you did not read make it mean
- ▶ You cannot feel it accruing. You feel it when you must **change** something

Predict: 300 generated lines accepted, 150 read;

How much do you owe?

- ▶ 150 lines
- ▶ Nothing — it works
- ▶ More than 150: the lines you read depend on the lines you did not
- ▶ Whatever the tests do not cover

Measuring it: three instruments;

Instrument	How	What it measures
The explain test	Pick a line. Write what you think it does. Ask. Compare	The gap between assumption and fact
The change estimate	Name a feature. List the files you must understand first	The bill, before you pay it
The review ratio	Of what you accepted, how much did you read?	How fast you are borrowing

Fewer than half of daily users say they always review before committing — direction, not decimal. Whatever your own ratio is, it is the rate at which the debt grows.

Common mistakes: choosing a mode;

- ▶ Vibe coding something you then **keep**
- ▶ Writing a specification for a prototype you will delete on Sunday
- ▶ Producing a spec nobody reads, and calling that rigour
- ▶ Judging a generated app by whether it runs
- ▶ Choosing a mode by habit rather than by task

Common mistakes: running the mode;

- ▶ A spec made of adjectives — "robust", "clean", "user-friendly" — instead of decisions
- ▶ Reading the plan as the model restating your spec, and skimming it
- ▶ Patching a wrong structure for the fifth time because each patch is cheap
- ▶ Counting the lines you read as understanding, when they rest on lines you did not
- ▶ Measuring the debt by whether the demo runs

Summary;

- ▶ Vibe coding is **excellent** where being wrong is cheap and the artefact is disposable
- ▶ Adoption outran governance: **1.7×** the major issues, **~45%** with an OWASP Top-10 issue — direction, not decimals
- ▶ **Comprehension debt** is borrowed understanding. It compounds through lines nobody read; the explain test, the change estimate and the review ratio measure it
- ▶ **Spec-driven** moves review from implementation to intent, at a real cost of its own. A spec makes the **gaps smaller**, not the model smarter: the decisions become yours
- ▶ **Plan first**: a misunderstanding costs a sentence, not a rewrite
- ▶ Choose **per task**: stakes, reversibility, lifespan, and who maintains it

"Forget that the code even exists" is fine — right up to the moment you have to change it.