

```
// making the machine check the machine
```

Automated Checking and Evals;

If you are not reading every line;

...then something else is checking it.

- ▶ What, exactly?
- ▶ And who checks **that**?

The one idea;

Automated checks are what make it **defensible** to ship code you did not read line by line. Without them you did not move up a level — you stopped checking.

- ▶ CI is not administration. It is the other half of the trade

The two hours;

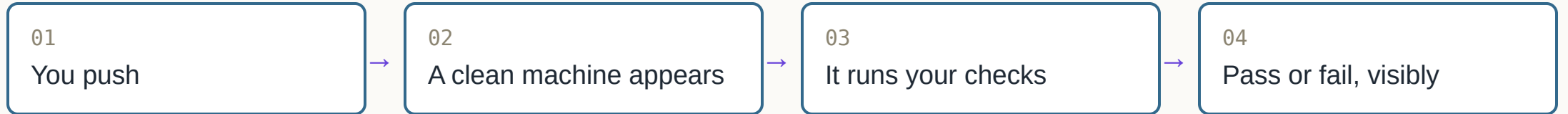
Part 1 — what to check, and in what order

- ▶ A pipeline is a script with a trigger; what goes in it, by value
- ▶ AI inside the pipeline, and the one honesty rule
- ▶ **Evals** — a set and a rate, for things that answer differently every time
- ▶ The assertion ladder, and the judge at the top of it

Part 2 — how it actually works

- ▶ What a run really is, and why first runs fail
- ▶ What each rung of the ladder can and cannot catch
- ▶ A second case: output that feeds other code — then **try it now**

A pipeline is a script with a trigger;



- ▶ The only real difference from running them yourself: **it runs whether or not you remembered**

What goes in, roughly in value order;

Check	Cost to add	Catches
Does it build / import	Minutes	Broken merges
Linting and formatting	Minutes	Style drift, some bugs
Secret scanning	Minutes	The mistake you cannot undo
Dependency scanning	Minutes	Known vulnerabilities
Unit tests	Real effort	Actual regressions

// the top four are nearly free; do them first

Predict: your project has few tests. What is worth adding first?;

- ▶ Nothing — CI is pointless without a test suite
- ▶ A full test suite, before anything else
- ▶ Secret scanning and dependency checks
- ▶ A deployment step

The checks you cannot undo by fixing the code;

- ▶ A failing test → a red build. Push a fix
- ▶ A leaked key → rotate it, and hope. **Pushing a fix does not un-leak it**

Order your checks by **irreversibility**, not by sophistication.

AI inside the pipeline;

- ▶ **Review on pull requests** — asks the questions a linter cannot
- ▶ **Issue triage** — classify, label, route
- ▶ **Documentation drift** — does the README still describe this code?

Ask it what a compiler **cannot** check. Never ask it whether the code compiles — a compiler answers that exactly, and a model guesses.

What you ask it, and what you do not;

VAGUE PROMPT

Review this pull request and tell me whether the code is correct.

- ▶ "Correct" is the compiler's question and the tests' question. A model **guesses** at it, in confident prose

BETTER PROMPT

Here is a diff. List only: a function or variable whose name no longer matches what it does; a docstring or README line this change made false; an edge case the change does not handle. If there are none, reply NONE.

- ▶ A closed list of things no compiler can check, and a fixed word for "nothing" — so an empty review and a broken review can never look the same

A failed call is a failure, not a finding;

- ▶ If the model call errors, the run **failed**. It did not "find nothing"
- ▶ A job that reviewed nothing must go **red**

Otherwise you get the worst outcome available: a green tick over a check that never ran.

The turn: an app with an AI feature;

- ▶ Same input. **Different output.** Every time
- ▶ Every testing instinct you have assumes determinism

```
assert summarise(article) == "Revenue grew 12%."
```

- ▶ This test fails on a perfectly good answer

Evals: measure a rate, not a result;

Stop asking "did this one call work?" Start asking "**across a set of cases, how often is it acceptable — and is that number moving?**"

- ▶ One case tells you almost nothing
- ▶ Twenty cases tell you whether your change helped

The assertion ladder;

Rung	Check	Use when
Exact match	<code>output == expected</code>	Classification, extraction
Contains / regex	Key fact present	The answer must mention something
Structural	Valid JSON, required fields	The output feeds other code
Property	Length, no leaked prompt, cites a source	Always worth adding
LLM-as-judge	Another model scores it	Nothing above can express it

► Use the **strongest rung the task allows**, not the fanciest

Predict: how do you test a summariser?;

Your feature summarises articles. Which do you build first?

- ▶ LLM-as-judge — summaries are subjective
- ▶ Exact match against a reference summary
- ▶ Property checks: mentions the key figure, under 50 words, non-empty
- ▶ You cannot test it; ship and see

Properties first, judge last;

- ▶ Property checks are **cheap, deterministic and repeatable**
- ▶ They catch the failures that actually happen: empty output, prompt echo, runaway length, missing the key fact
- ▶ Keep the judge for the residue that no property can express

LLM-as-judge, honestly;

- ▶ Give a model the output and a rubric; ask for a score and a reason
- ▶ It works. It is also **biased in known ways**:

Bias	What happens
Self-preference	Rates output from its own family higher
Position	In an A/B comparison, order changes the verdict
Verbosity	Longer answers score better than they deserve

- ▶ Always ask for the **reason**, not just the score — an unjustifiable score is usually visibly unjustifiable

Predict: you tweak the prompt and your three test cases improve;

Is the feature better?

- ▶ Yes — the evidence is right there
- ▶ Unknown, until you measure the whole set
- ▶ Only if the model version stayed the same
- ▶ Yes, if the three cases were representative

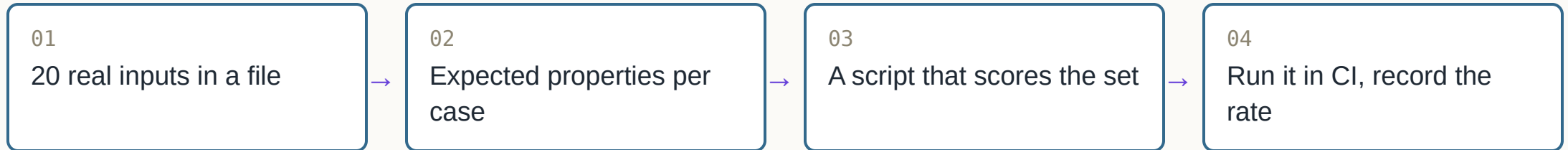
Unknown – and this is the whole point;

- ▶ You improved the cases you were looking at
- ▶ You have no idea what happened to the ones you were not

Without a measured set, prompt tuning is **folklore**. With one, it is engineering.

- ▶ Run the set on every change. Track the number over time

What this looks like for a project;



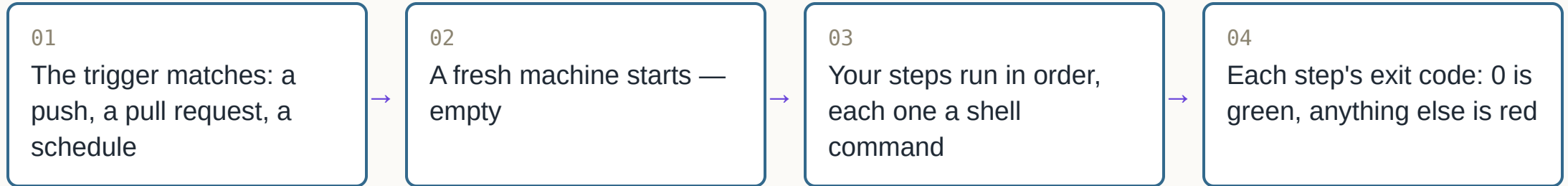
- ▶ Twenty cases in a JSON file is a **real** eval suite
- ▶ Do not build a framework. Build the file

```
// part 1 named the things. part 2 opens them up
```

Ten minutes;

Part 2 answers two questions: what actually happens when a run starts, and how you score an answer that is never the same twice.

What actually happens when you push;



- ▶ The machine is **not your laptop**. No packages, no files, no keys — not even your code until a step fetches it
- ▶ A step that exits non-zero fails the job, and the steps after it do not run
- ▶ That emptiness is the point: the run can only pass on what is actually **committed**

The smallest real workflow;

```
name: ci
on: [push]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7          # 01: fetch the code
      - uses: actions/setup-python@v7
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt # 02: it has nothing
      - run: python -m pytest              # 03: the exit code decides
```

// it starts in the repository root, not in your folder

Why first runs fail;

The log says	What actually happened
No module named ...	Fresh machine. Installed on your laptop is not installed here
No such file or directory	No checkout step — or the run started in the repository root, not your folder
The model call fails: no key	Nothing from your <code>.env</code> is there. A secret reaches the run only if the workflow hands it in
Red on the first push, green on the second	You fixed the workflow , not the code. That is what the first run is for
Green, but it ran nothing	The step exited 0 anyway. Next slide

Predict: this step is green. What did it prove?;

```
- name: AI review  
  run: python review.py || echo "review step finished"
```

- ▶ The review ran and found nothing
- ▶ The review ran; we do not know what it found
- ▶ Nothing — it is green whether the review ran, failed, or was never installed
- ▶ That `review.py` exists

Green means "every step exited 0". Nothing more;

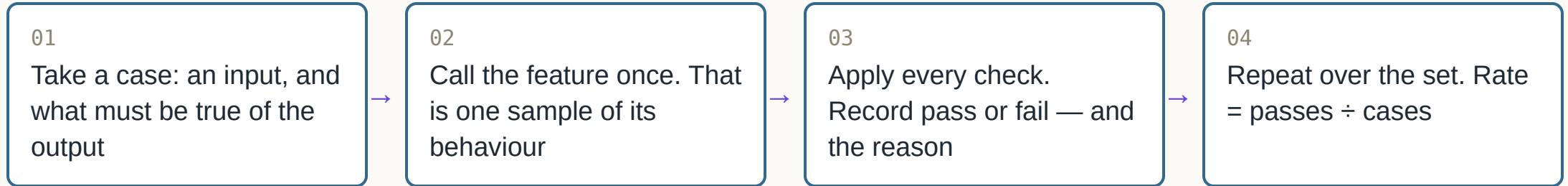
- ▶ `||` runs its right-hand side only when the left fails — and `echo` always succeeds. The same swallow in Python:

```
try:
    findings = review(diff)
except Exception:
    findings = "no findings"    # the error just became a verdict
print(findings)               # ...and the step exits 0
```

- ▶ A job **skipped** by a condition is not red either. Skipped and passed look the same from a distance

Make it go **red on purpose** once. A pipeline you have never seen fail is one you have no evidence works.

How an eval run scores one answer;



- ▶ One output is a **sample**, not the answer. The same case can pass now and fail next run — that variability is what you are measuring
- ▶ Twenty cases: one case is **five points**. A small move on one run is not yet a regression; the same case failing every run is
- ▶ The reason string is what you read when the number moves

What each rung can and cannot catch;

Rung	Catches	Blind to
Exact match	Any deviation at all	Nothing — but rejects every good paraphrase too
Contains / regex	The key fact missing	The key fact present with its meaning reversed
Structural	Output that will not parse, missing fields	Wrong values in the right shape
Property	Empty, too long, echoed prompt	Whether any of it is true
Model-as-judge	Meaning, faithfulness, tone	Its own bias — its verdict is a sample too

// below the judge, every rung is exact about shape and silent about meaning

Predict: which check catches this?;

The article says revenue **rose** 12%. The feature returns:

REPLY

Revenue fell 12% this quarter, driven by renewals.

Your checks: non-empty · under 50 words · mentions "12%" · does not echo the prompt.

- ▶ The length check
- ▶ The "mentions 12%" check
- ▶ None of them — all four pass
- ▶ Exact match is the only fix

Presence is not meaning;

- ▶ Every rung below the judge checks **shape**: is the token there, does it parse, how long is it
- ▶ A reversed claim has perfect shape. This is the residue the judge exists for
- ▶ Two honest options: a **targeted property** if the domain allows one — a direction word that contradicts the source — or a **judge** given the article and one question: *is every claim in the summary supported by it?*

Give the judge the **source**. Shown only the summary, it cannot check faithfulness — it can only rate how the sentence reads, and a wrong summary reads perfectly well.

Try it now: three runs, one figure;

Type this at whatever assistant you have. Then open a **new conversation** and type it again. Three runs in total.

PROMPT

Summarise this in one sentence: "The library moved study-room bookings online in March. Bookings rose 40% in the first month, mostly for evening slots, and no-shows fell by a third once reminders were added."

- ▶ **Notice:** the three sentences differ. A test of `==` would have failed a correct answer on at least one run
- ▶ Write down **three properties** all three share that a wrong summary would fail. Did any run drop a figure — or change one?

A second case: output that feeds other code;

A support inbox. The feature reads a customer message and returns fields for the ticketing system:

```
{"product": "kettle", "issue": "stopped heating", "priority": "high"}
```

- ▶ Different in kind from the summariser: some fields have **one right answer**
- ▶ `priority` is a label from a fixed list → **exact match**. `product` must come from the message → a **property**. The whole thing must parse → **structural**
- ▶ Strongest rung the task allows — and this task allows higher rungs than a summary ever could

One case in the file;

```
{
  "id": "ticket-07",
  "input": "My kettle stopped heating after two days. I need a replacement before Friday.",
  "expect": {
    "keys": ["product", "issue", "priority"],
    "priority": "high",
    "product_in_input": true
  }
}
```

- ▶ Each case says what must be **true** of the output, not what the output must **be**
- ▶ Twenty of these is an afternoon. It is also a real eval suite

Scoring it: three rungs in a dozen lines;

```
def score(case, output):
    try:
        data = json.loads(output)                # structural
    except json.JSONDecodeError:
        return False, "not JSON"
    for key in case["expect"]["keys"]:            # structural
        if key not in data:
            return False, f"missing field: {key}"
    if data["priority"] != case["expect"]["priority"]: # exact
        return False, f"priority was {data['priority']} "
    if data["product"].lower() not in case["input"].lower(): # property
        return False, "product not in the message"
    return True, ""
```

- Checks run in **ladder order**, so the reason names the earliest thing that went wrong. Every line is deterministic: the only thing that varies is the feature

What the run found;

```
Running 20 cases
```

```
ticket-01 PASS
```

```
ticket-02 FAIL not JSON: begins "Here is the JSON you asked for:"
```

```
ticket-03 PASS
```

```
ticket-07 FAIL priority was medium
```

```
...
```

```
Pass rate: 16/20 (80%)
```

- ▶ The commonest failure was not a wrong answer. It was a right answer wrapped in **prose** — and the parser downstream would have crashed on it
- ▶ Two reasons, two different problems. The strings tell them apart without reading twenty outputs

The fix is a prompt change – so prove it;

VAGUE PROMPT

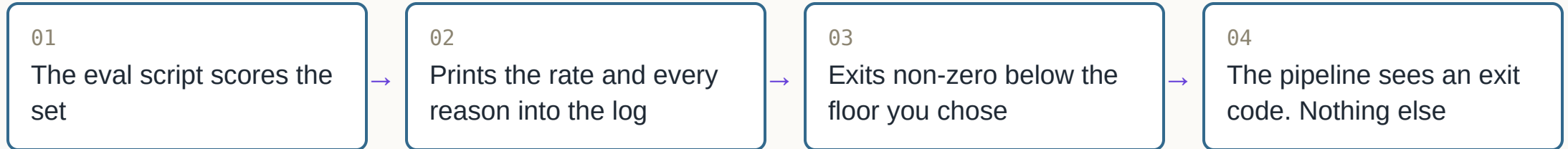
Extract the product, the issue and the priority from this message.

BETTER PROMPT

Extract the product, the issue and the priority from this message. Reply with one JSON object and nothing else: no sentence before it, no code fence around it. priority must be one of: low, medium, high.

- ▶ Then re-run **the whole set**, not ticket-02. Did a case that passed before start failing?
- ▶ A stricter format instruction can cost you a field the model used to include. Only the set shows that

The two halves meet at the exit code;



- ▶ The pipeline does not know what an eval is. It knows that one step returned 1
- ▶ So "what does red mean?" is **your** decision, in the script: below a floor, or below the last run
- ▶ And the rule from part 1 still applies: if the feature could not be called at all, exit non-zero. **Zero passes of zero cases is not 100%**

Predict: 20 of 20, ten changes in a row. Good news?;

Your eval set has passed every case on each of the last ten prompt changes.

- ▶ Yes — the feature is solid
- ▶ Yes, as long as the cases were realistic
- ▶ No — the set has stopped measuring anything new
- ▶ You cannot know without a judge

A set everything passes has stopped telling you anything new;

- ▶ Ten changes, ten perfect scores: you cannot tell the change that helped from the one that did nothing
- ▶ A rate is information only if it can **move**. When it stops moving, keep the passing cases as the regression guard and add the case that fails today
- ▶ The best new cases are real failures: the reversed figure, the JSON wrapped in prose, the run that dropped the number

The goal is not 100%. The goal is a number that goes **up when you improve things and down when you break them**.

Common mistakes;

- ▶ Treating CI as administration rather than as the other half of the trade
- ▶ Trusting green without ever having watched the pipeline go **red**
- ▶ Assuming the runner has what your laptop has: packages, files, keys, your folder
- ▶ Letting a failed model call be reported as "no findings" — in a `try` or in a `||`
- ▶ Testing a non-deterministic feature with `assertEqual`
- ▶ Reaching for a judge when a property would be exact — or checking a fact is *present* and believing you checked it is *true*
- ▶ Tuning a prompt against the two examples you looked at; keeping a set at 100% instead of adding the case that fails

Summary;

- ▶ Automated checks are what make **not reading every line** defensible
- ▶ Order checks by **irreversibility** — secret scanning before test suites
- ▶ A run is a **fresh machine** doing only the steps you listed; green means every step exited 0, nothing more
- ▶ Ask AI what a compiler **cannot** check; a failed call is a failure, not a finding
- ▶ Non-deterministic features need a **set** and a **rate**; climb the ladder, **properties first, judge last** — shape below, meaning only at the top
- ▶ Measure before and after, or "better" is a feeling — and a rate that cannot move is not measuring

Twenty cases in a file, scored on every change, red when the number drops. That is the difference between engineering and folklore.