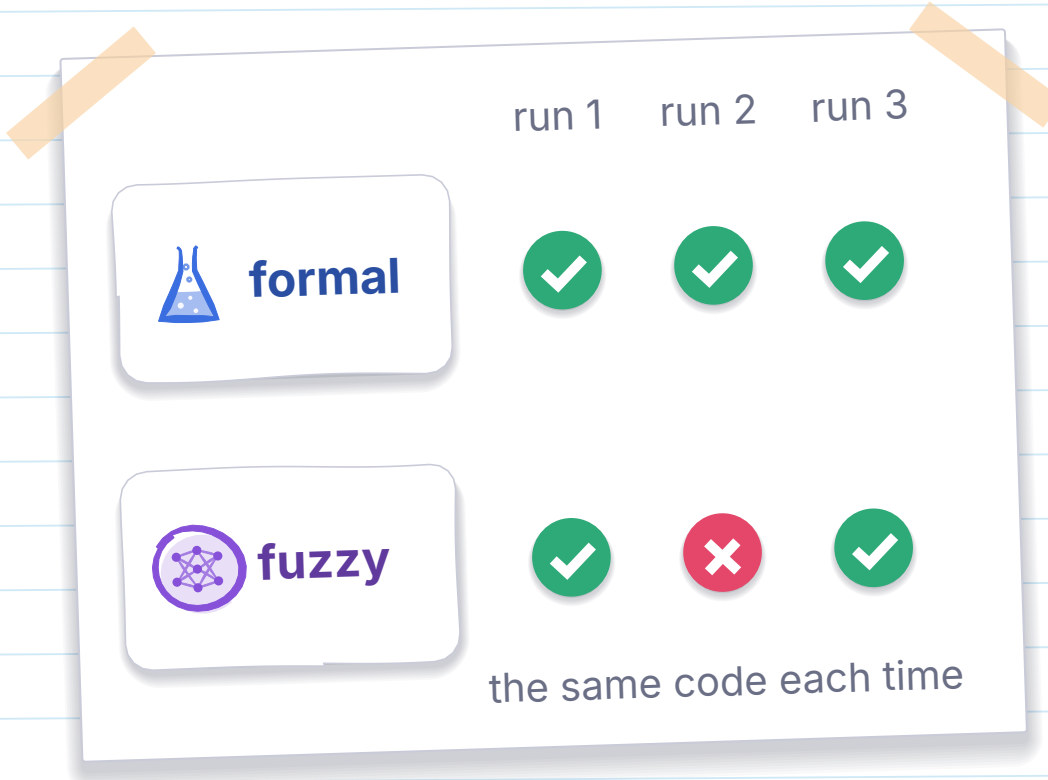
Prof. Aviv Zohar

Semester A, 2026–7

Formal tools and fuzzy tools

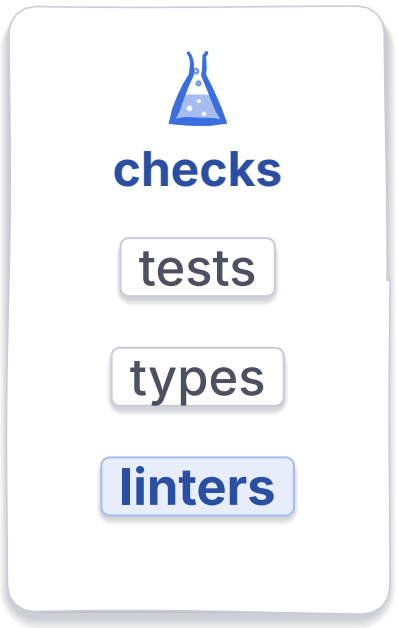
The two kinds of tool that check your code



Checks built for people fit agents too



make mistakes
forget earlier changes
can't hold a whole
codebase in mind



todo/storage.py

```
1 import os
2 import json
3
4 def save(items):
```

never used
legal, but suspicious

A linter reports it
without running the code.

Checks built for people fit agents too

people

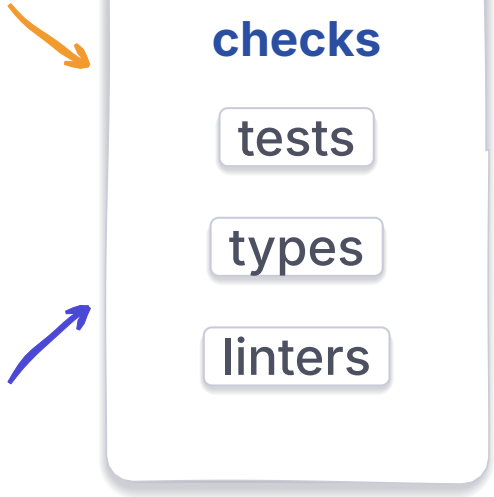


make mistakes
forget earlier changes
can't hold a whole
codebase in mind

agents



make mistakes
lose track of
earlier changes



todo/storage.py

```
1 import os
2 import json
3
4 def save(items):
```

Checks built for people fit agents too

people

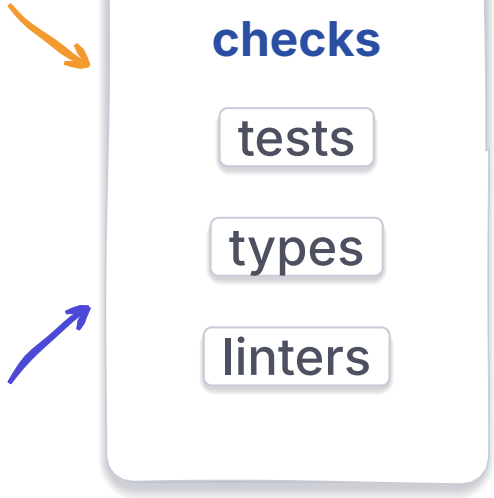


make mistakes
forget earlier changes
can't hold a whole codebase in mind

agents



make mistakes
lose track of earlier changes

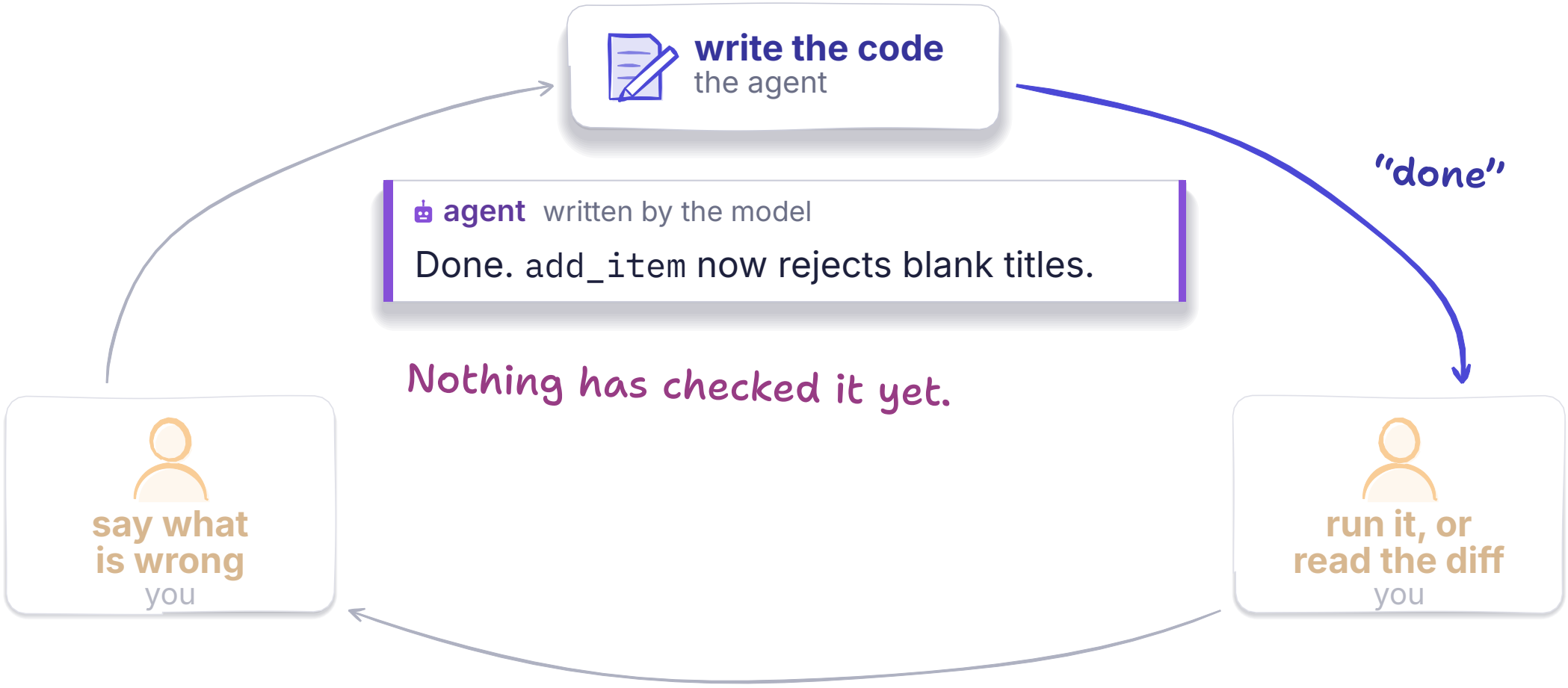


todo/storage.py

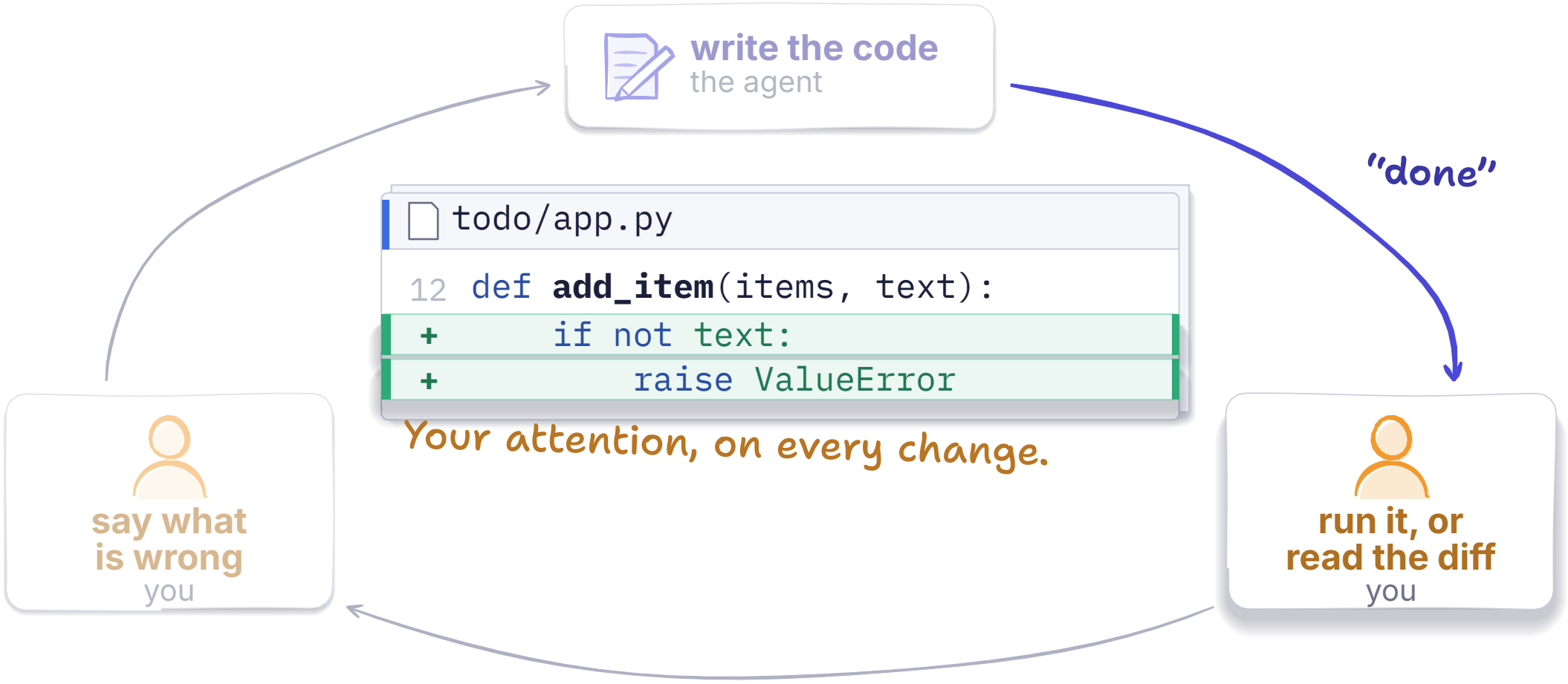
```
1 import os
2 import json
3
4 def save(items):
```

Both make mistakes that a check can find.

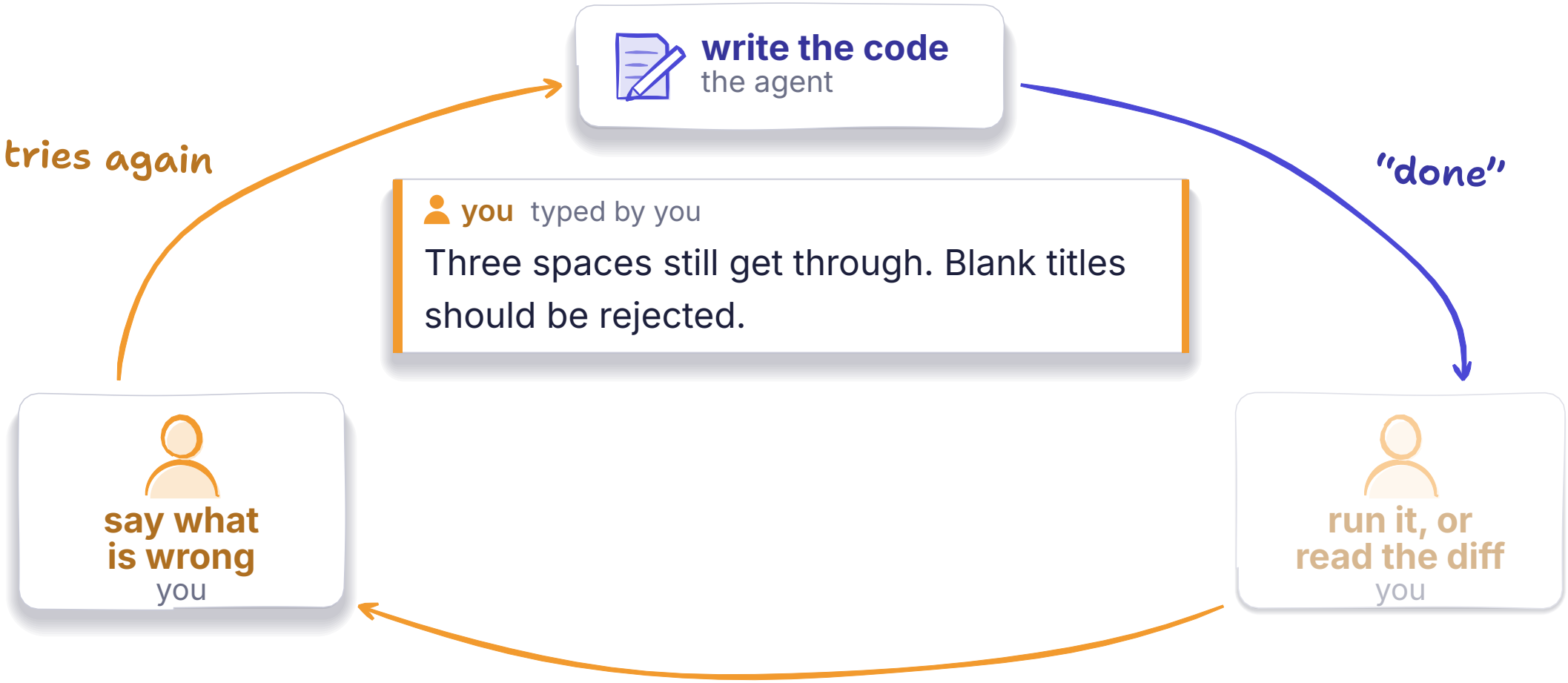
Without a check, every turn waits on you



Without a check, every turn waits on you



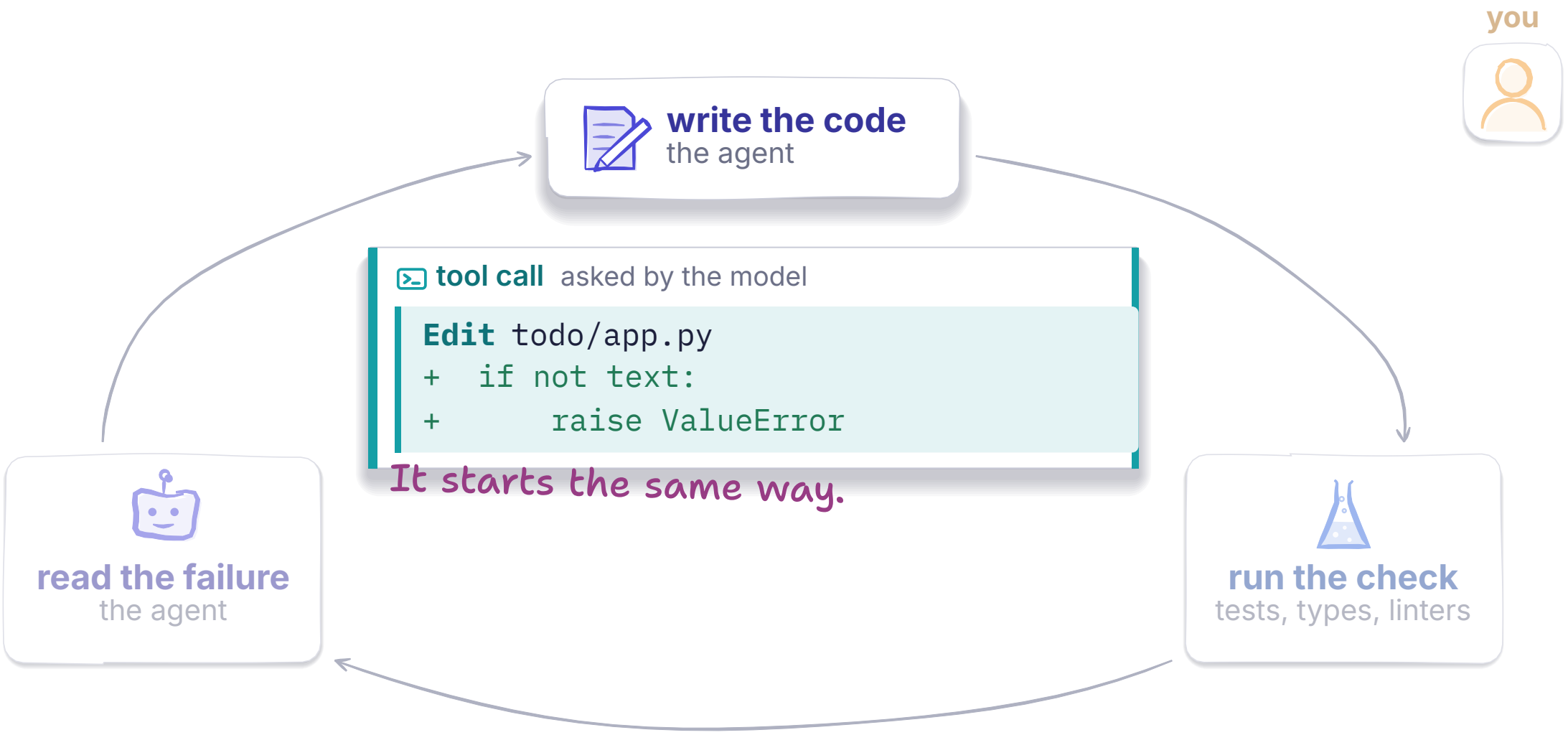
Without a check, every turn waits on you



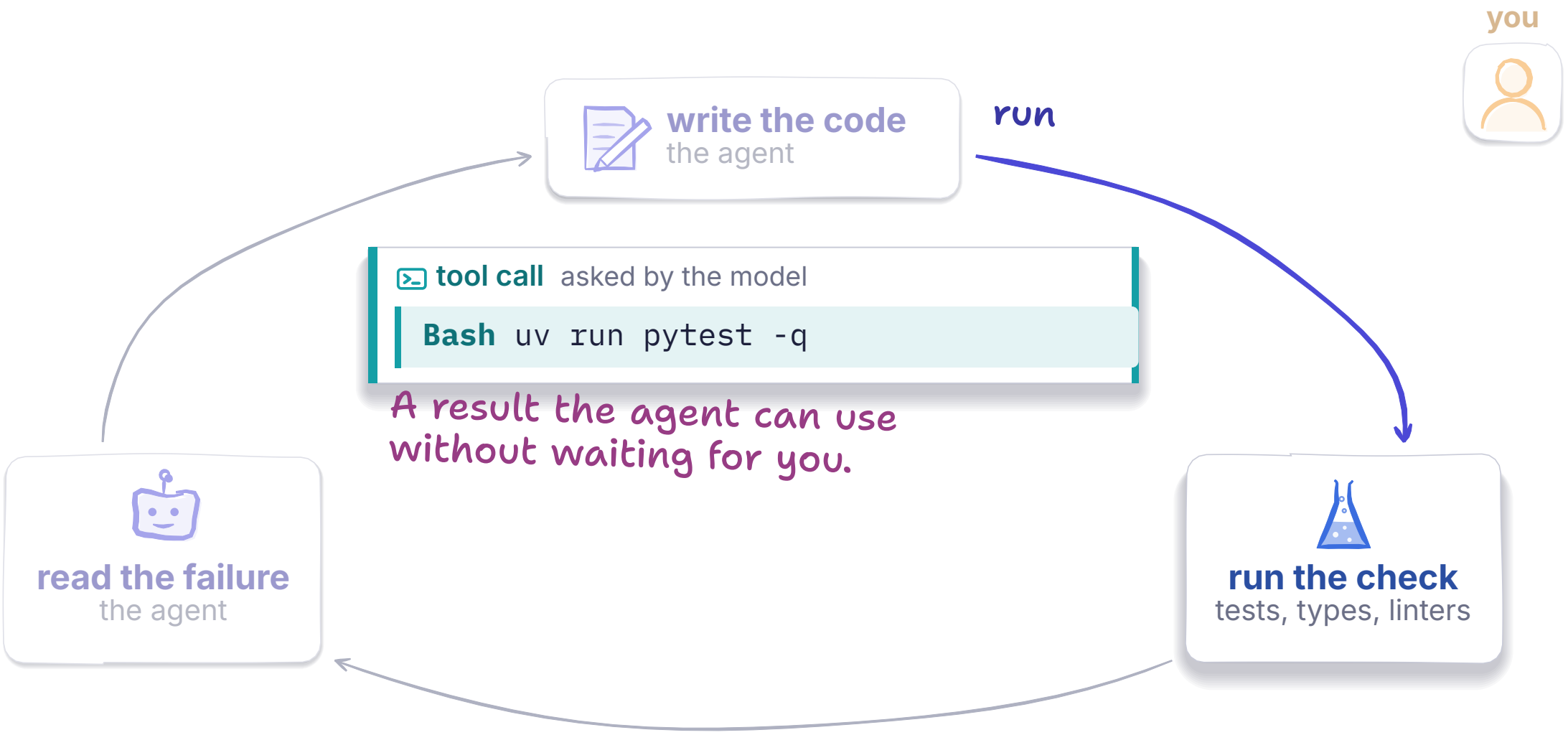
Without a check, every turn waits on you



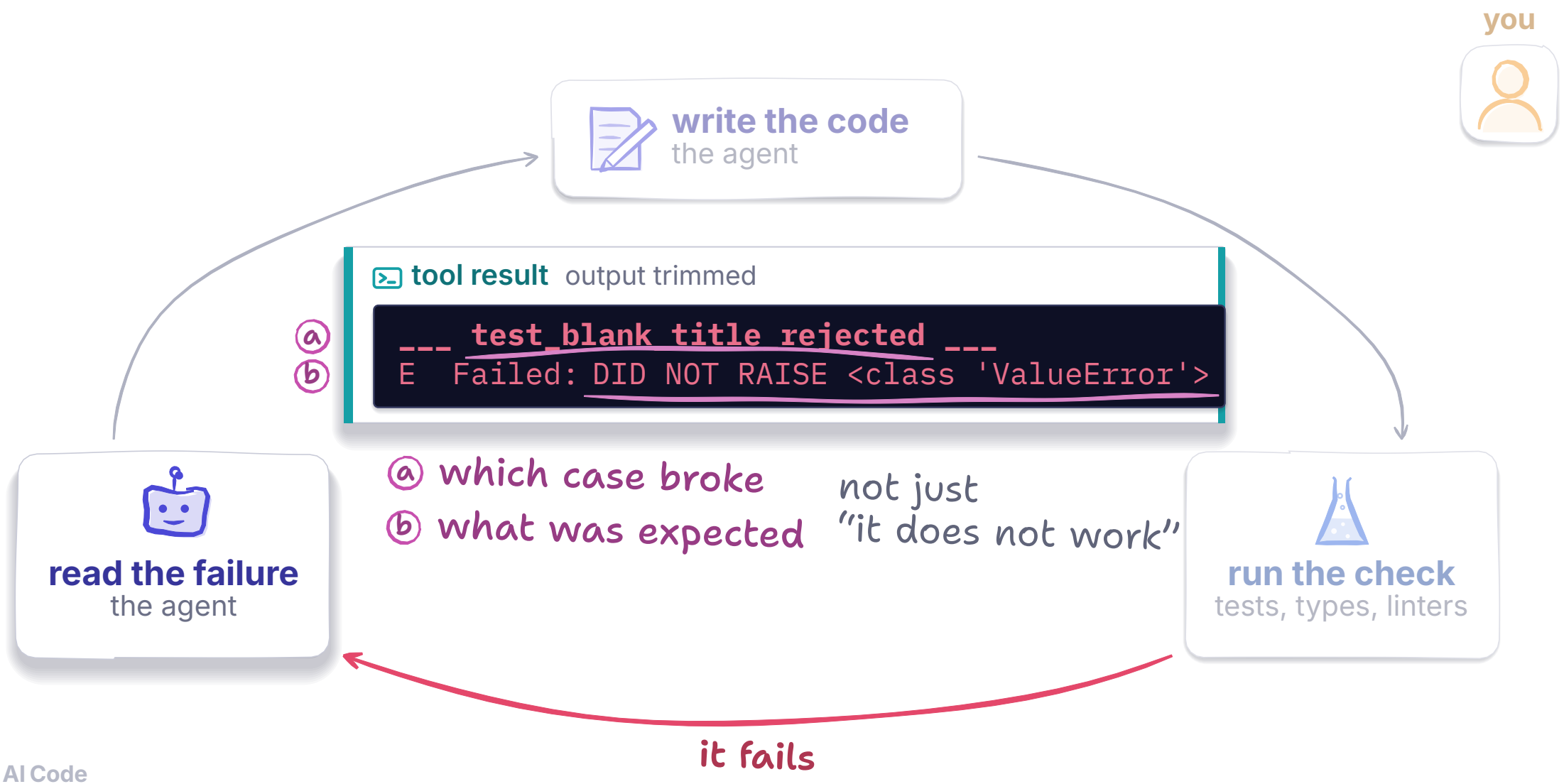
With a check, the agent can go round on its own



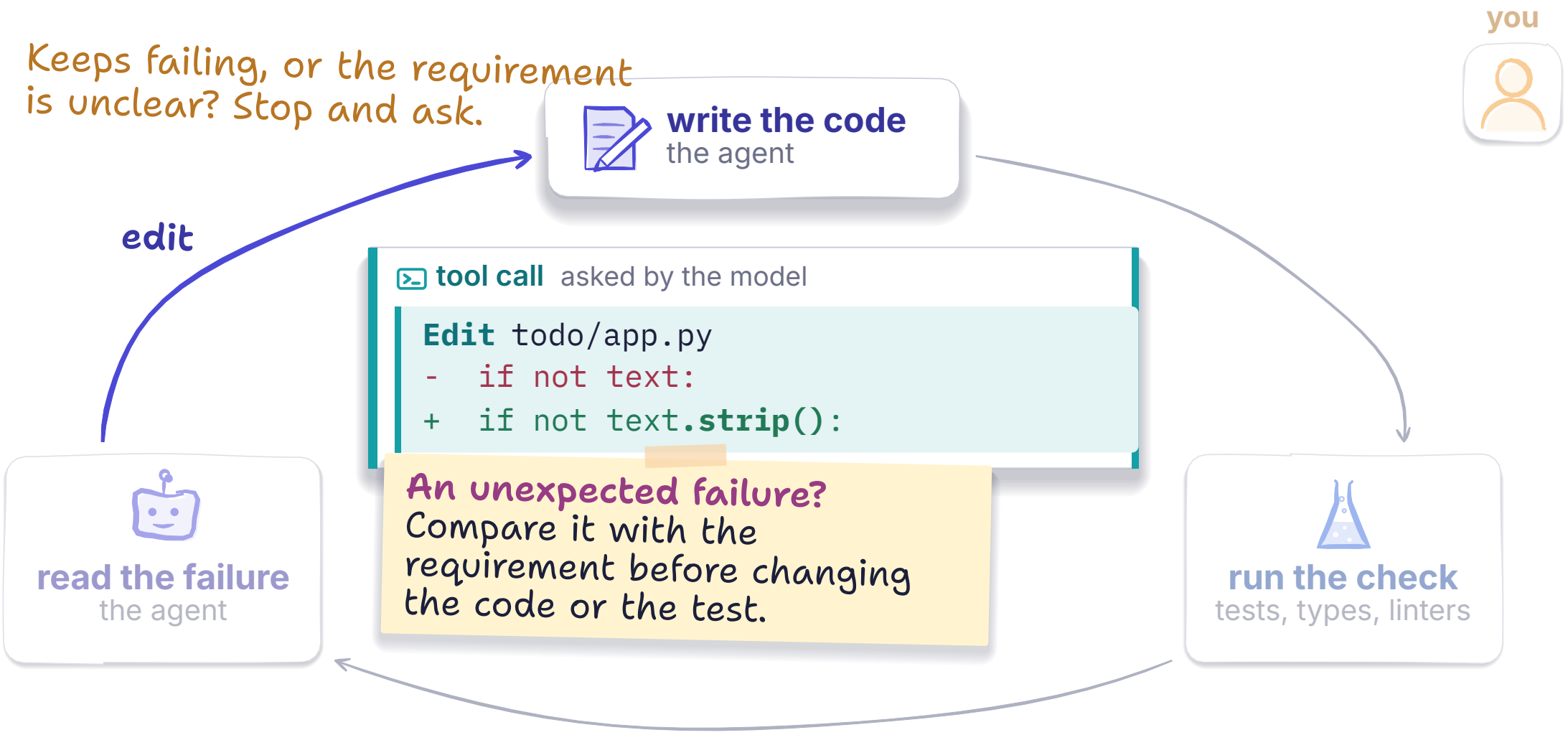
With a check, the agent can go round on its own



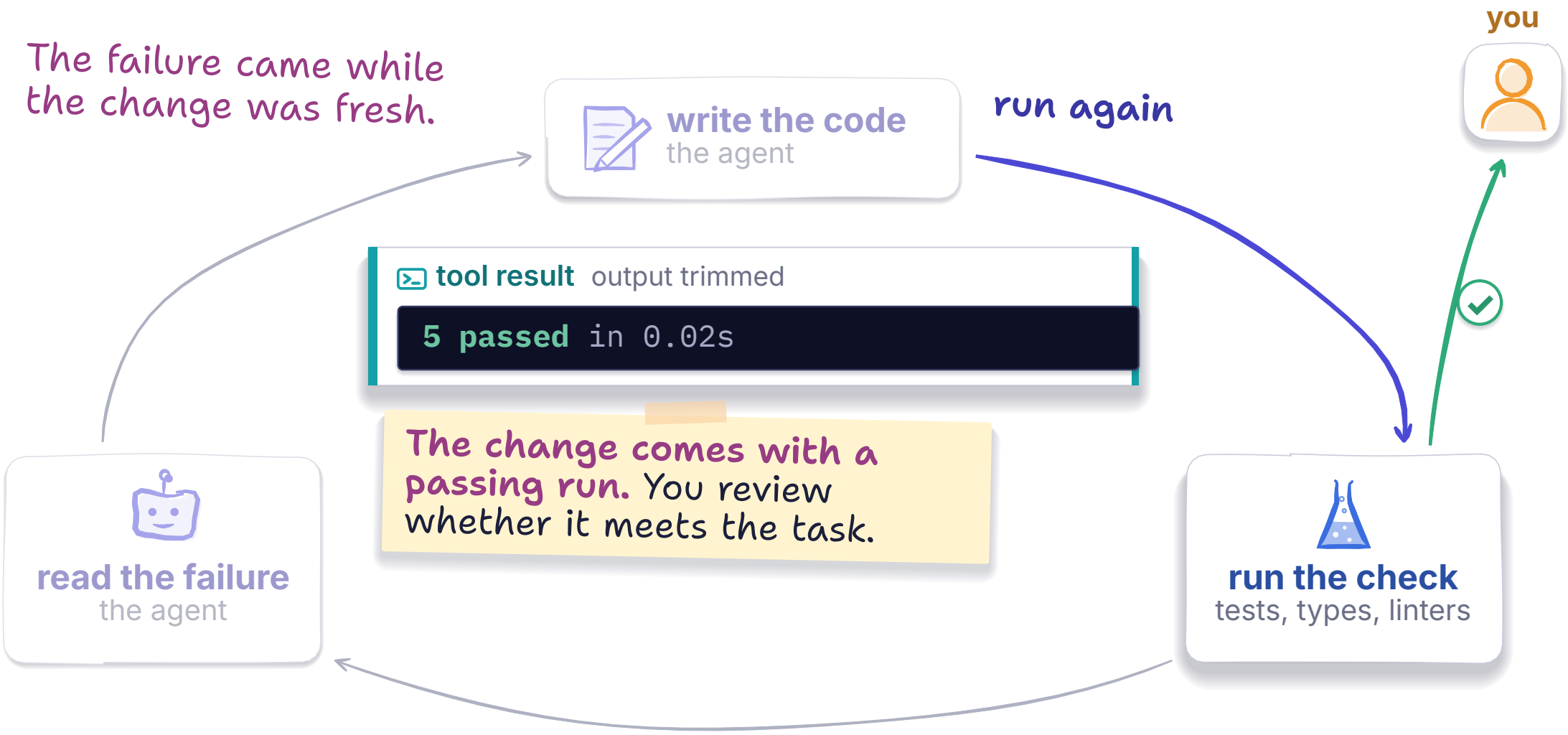
With a check, the agent can go round on its own



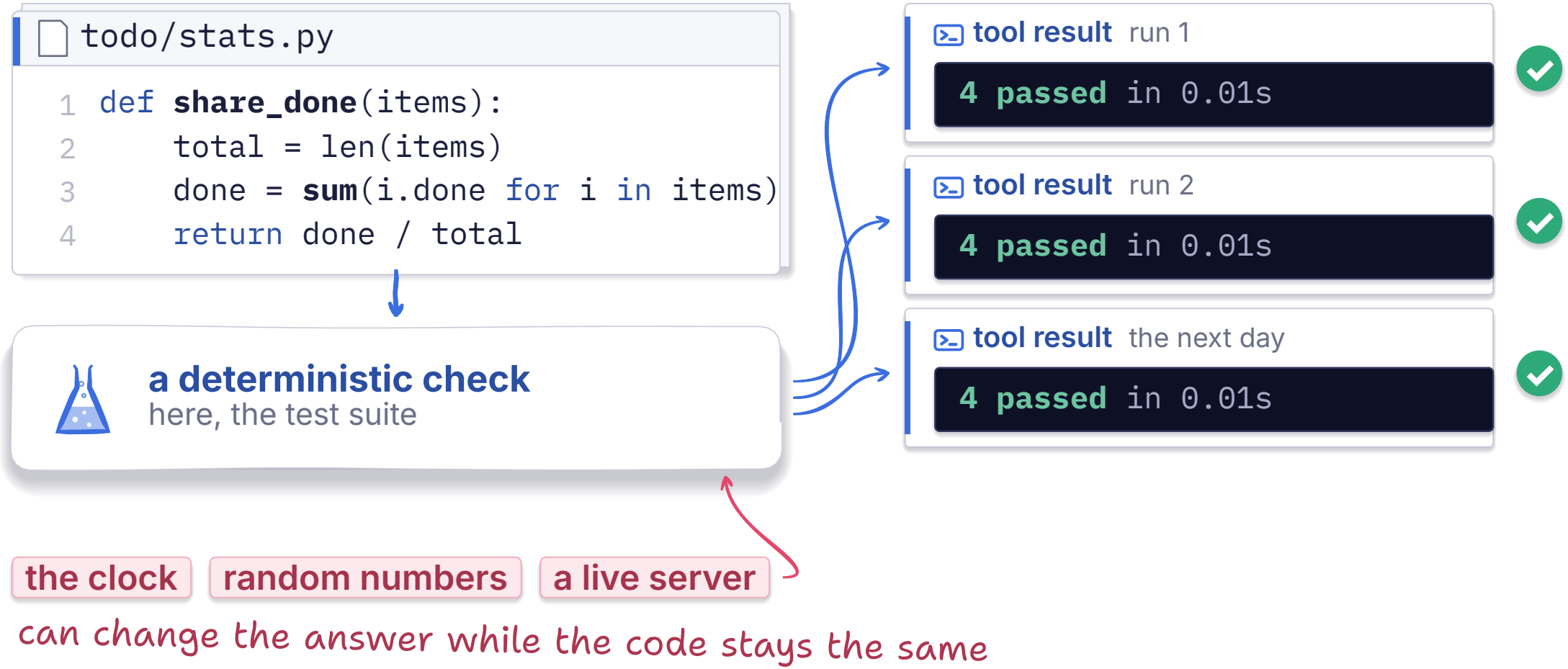
With a check, the agent can go round on its own



With a check, the agent can go round on its own



A formal tool gives the same answer every time



A formal tool gives the same answer every time

todo/stats.py

```
1 def share_done(items):
2     total = len(items)
3     done = sum(i.done for i in items)
4     return done / total
```

a deterministic check
here, the test suite

tests | type checkers | linters | compilers

formal tools

tool result run 1

```
4 passed in 0.01s
```

tool result run 2

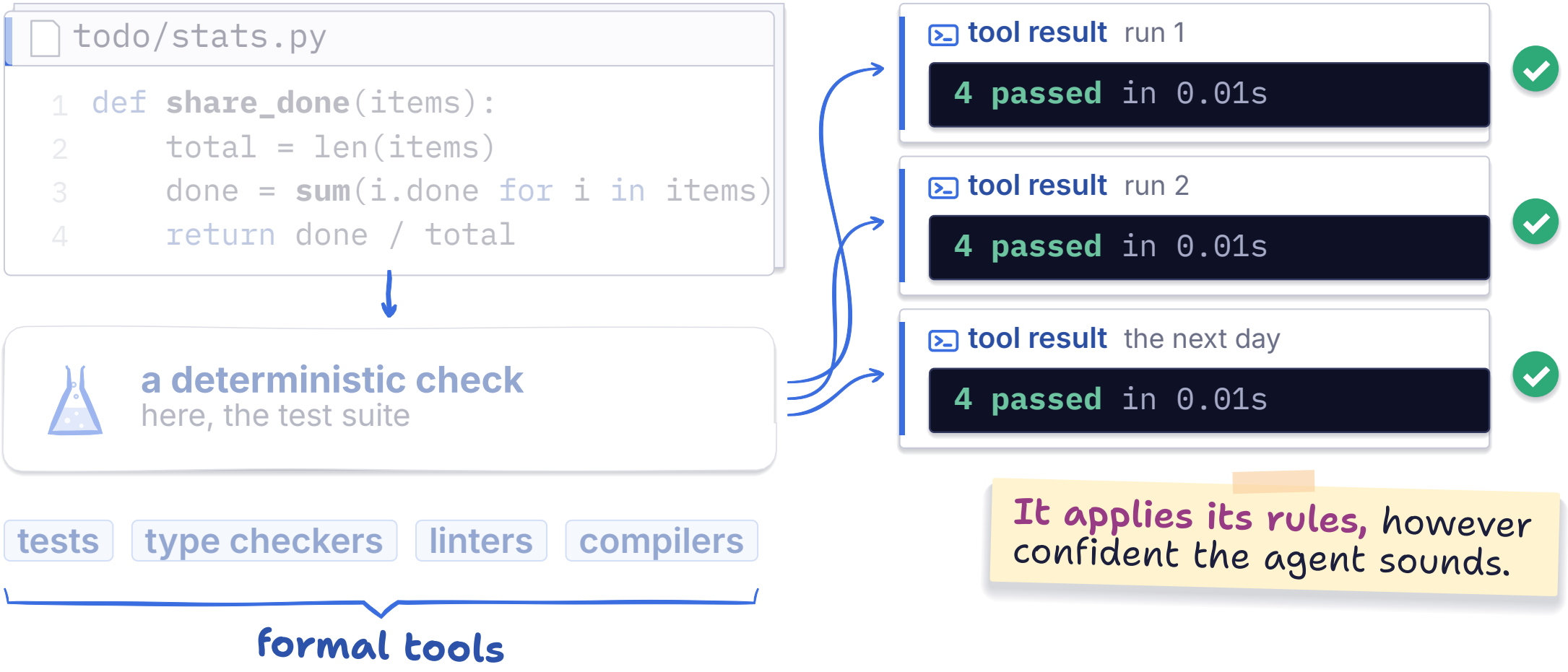
```
4 passed in 0.01s
```

tool result the next day

```
4 passed in 0.01s
```

Narrow: only what it was given.
Exact: each condition passes or fails.

A formal tool gives the same answer every time



The agent as a reviewer is broad, but it varies

todo/stats.py

```
1 def share_done(items):
2     total = len(items)
3     done = sum(i.done for i in items)
4     return done / total
```



the agent, as reviewer
asked: “Is this correct?”

Broad: it can answer almost any question about the code.

run 1

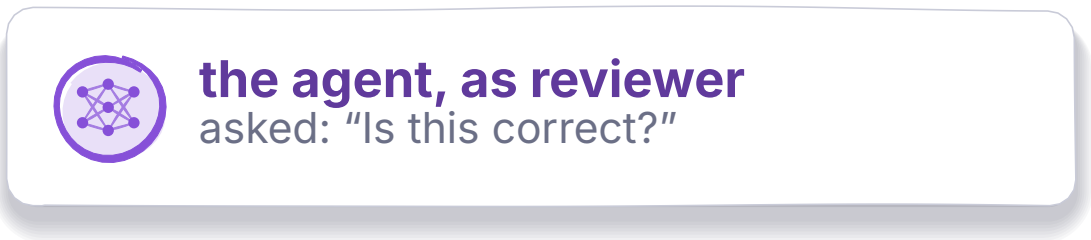
run 2

run 3

The agent as a reviewer is broad, but it varies

todo/stats.py

```
1 def share_done(items):
2     total = len(items)
3     done = sum(i.done for i in items)
4     return done / total
```



The code has not changed at all.

agent run 1
Correct and complete.

agent run 2
The empty list is not handled.

agent run 3
Correct. Maybe rename `total`.

The agent as a reviewer is broad, but it varies

todo/stats.py

```
1 def share_done(items):
2     total = len(items)
3     done = sum(i.done for i in items)
4     return done / total
```

the agent, as reviewer
asked: “Is this correct?”

todo/stats.py read
todo/storage.py skipped, though it matters

agent run 1
Correct and complete. ✓

agent run 2
The empty list is not handled. ✗

agent run 3
Correct. Maybe rename `total`. ✓

agent no tool call ran
I checked the empty list. ?

The agent as a reviewer is broad, but it varies

todo/stats.py

```
1 def share_done(items):
2     total = len(items)
3     done = sum(i.done for i in items)
4     return done / total
```



the agent, as reviewer
asked: “Is this correct?”

agent run 1
Correct and complete.

agent run 2
The empty list is not handled.

agent run 3
Correct. Maybe rename `total`.

The verdict changes; the code does not. A useful reviewer, not a stable check.

The two kinds of tool answer different questions

conditions encoded as checks

expected behaviour

compatible types

project rules

broader review questions

Are the names clear?

Does the structure make sense?

Is an important case missing?

The two kinds of tool answer different questions



the formal tool: exactly these

conditions encoded as checks

- expected behaviour
- compatible types
- project rules

broader review questions

- Are the names clear?
- Does the structure make sense?
- Is an important case missing?

A passing suite says nothing about a confusing design.

The two kinds of tool answer different questions

the agent reaches across both



conditions encoded as checks



expected behaviour

compatible types

project rules

```
share_done([]) == 0
```

broader review questions

Are the names clear?

Does the structure make sense?

Is an important case missing?

It can be wrong, even here.

You decide what it should give:
now it is a test.

The two kinds of tool answer different questions

first: run the deterministic checks

1

conditions encoded as checks

expected behaviour

compatible types

project rules

a local test run: **seconds** of computer time

then: ask the agent

2

broader review questions

Are the names clear?

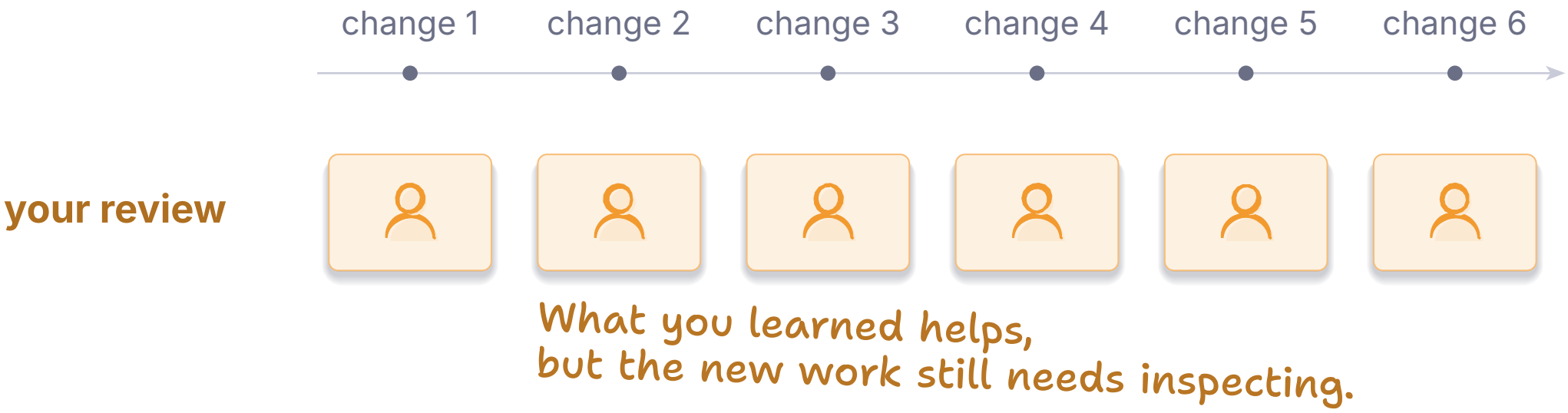
Does the structure make sense?

Is an important case missing?

a model review: paid in **generated tokens**

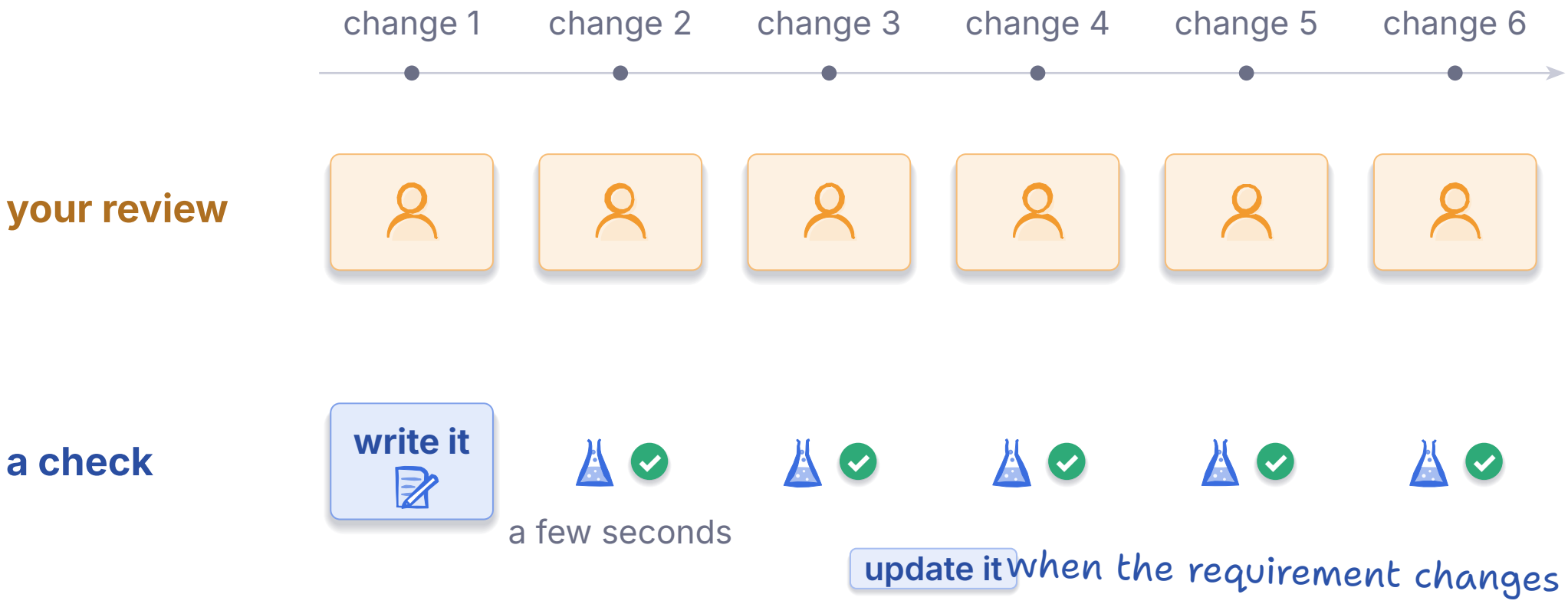
Can you state the condition exactly? **Check it with a deterministic tool.**

A check you write once runs after every change

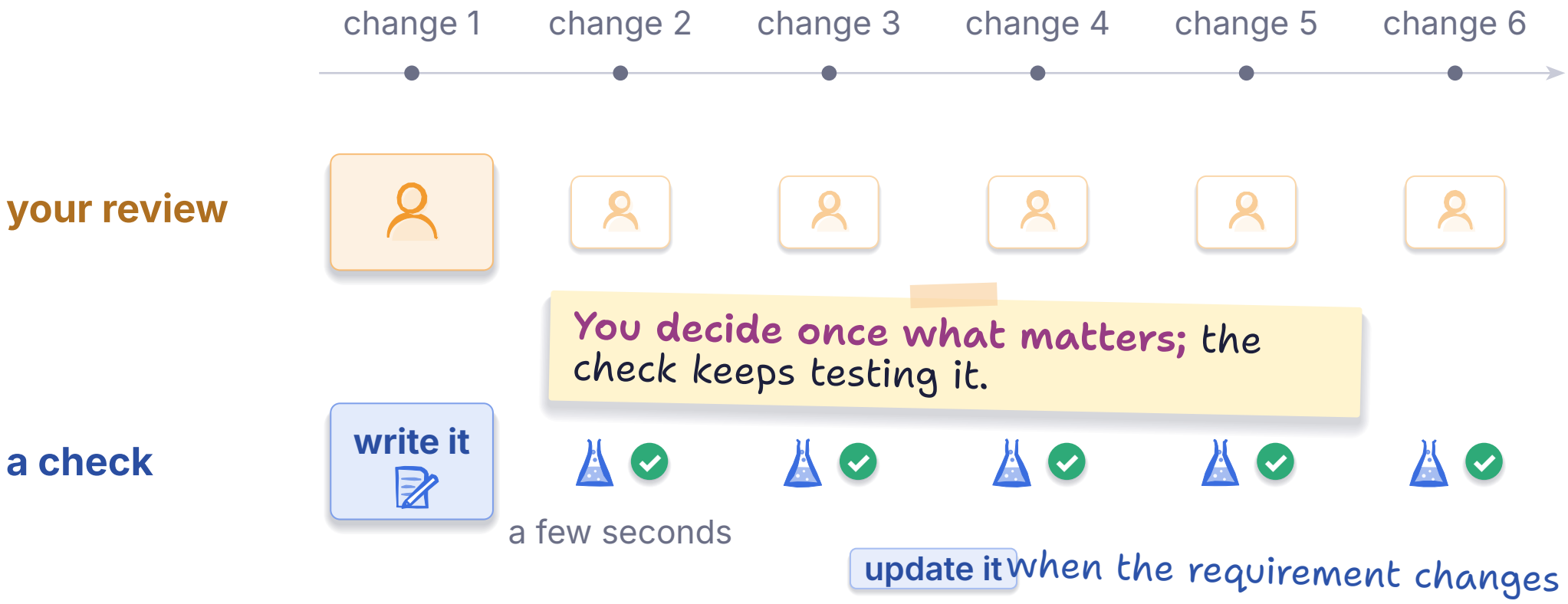


What you learned helps,
but the new work still needs inspecting.

A check you write once runs after every change

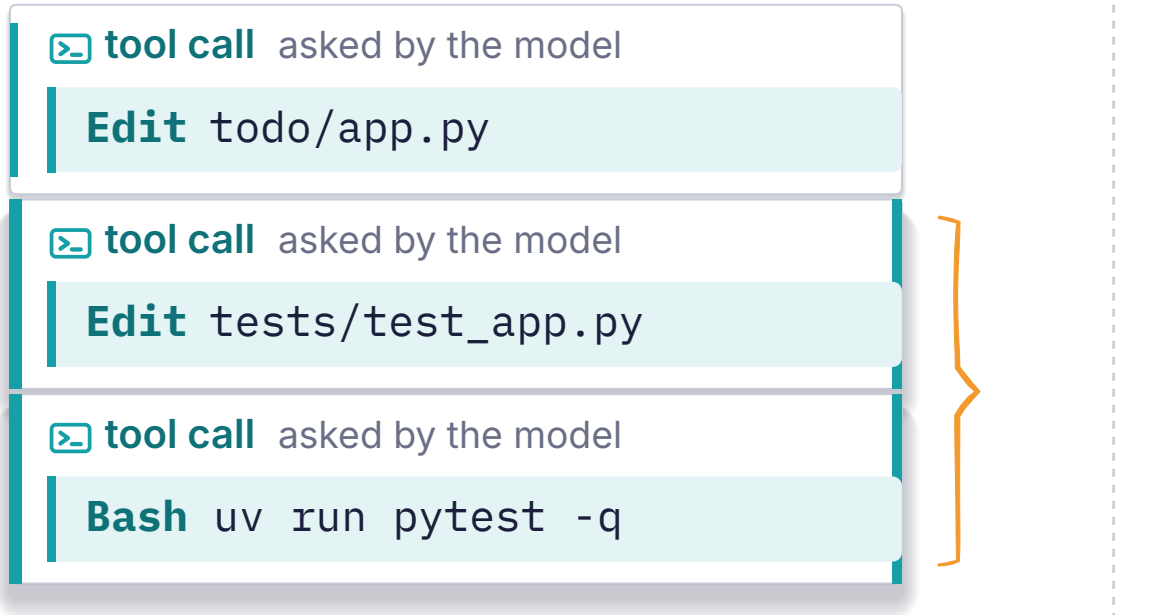


A check you write once runs after every change



Tests cost something, and pay back twice

the costs



more tokens and more time, on every task

the return

Tests cost something, and pay back twice

the costs

tool call asked by the model

```
Edit todo/app.py
```

tool call asked by the model

```
Edit tests/test_app.py
```

tool call asked by the model

```
Bash uv run pytest -q
```

todo/app.py

tests/test_app.py

a requirement changed, so its tests change too

the return

Tests cost something, and pay back twice

the costs

tool call asked by the model

```
Edit todo/app.py
```

tool call asked by the model

```
Edit tests/test_app.py
```

tool call asked by the model

```
Bash uv run pytest -q
```

todo/app.py
tests/test_app.py

a requirement changed, so its tests change too

the return

1. finds a defect in **this** change

`1 failed, 4 passed` caught now

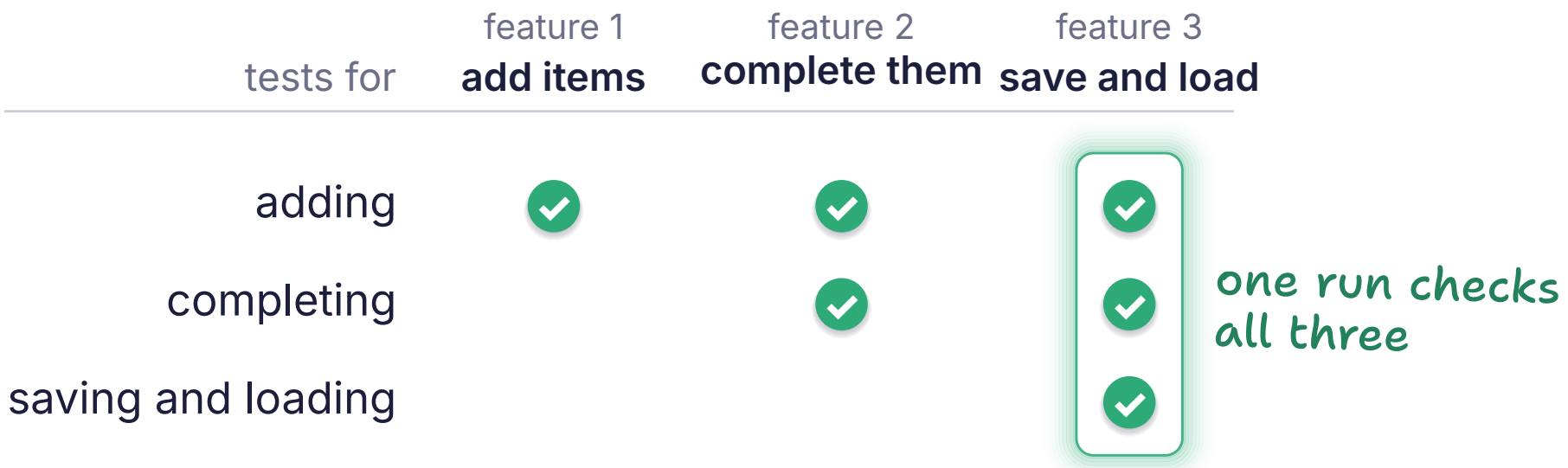
2. keeps checking it on **every later** change



change 2, 3, 4, ... : the tests stay in the repository

The costs are **easy to see**. The return comes in two parts.

Kept tests catch regressions, not new combinations



Kept tests catch regressions, not new combinations

tests for	feature 1 **add items**	feature 2 **complete them**	feature 3 **save and load**
adding	✓	✓	✗
completing		✓	✓
saving and loading			✓

Adding save and load broke adding items.

A regression: a change breaks a feature that used to work.

Fixed a bug? Keep the test that reproduced it. If the bug comes back, it fails again.

Kept tests catch regressions, not new combinations

tests for	feature 1 **add items**	feature 2 **complete them**	feature 3 **save and load**
adding	✓	✓	✓
completing		✓	✓
saving and loading			✓
the combination			*no test*

todo.json

```
[{"title": "Buy milk"}]
```

no done field saved

1. complete an item
2. save the list
3. load into a fresh app
4. select *Active*:
   the done item is listed

Kept tests catch regressions, not new combinations

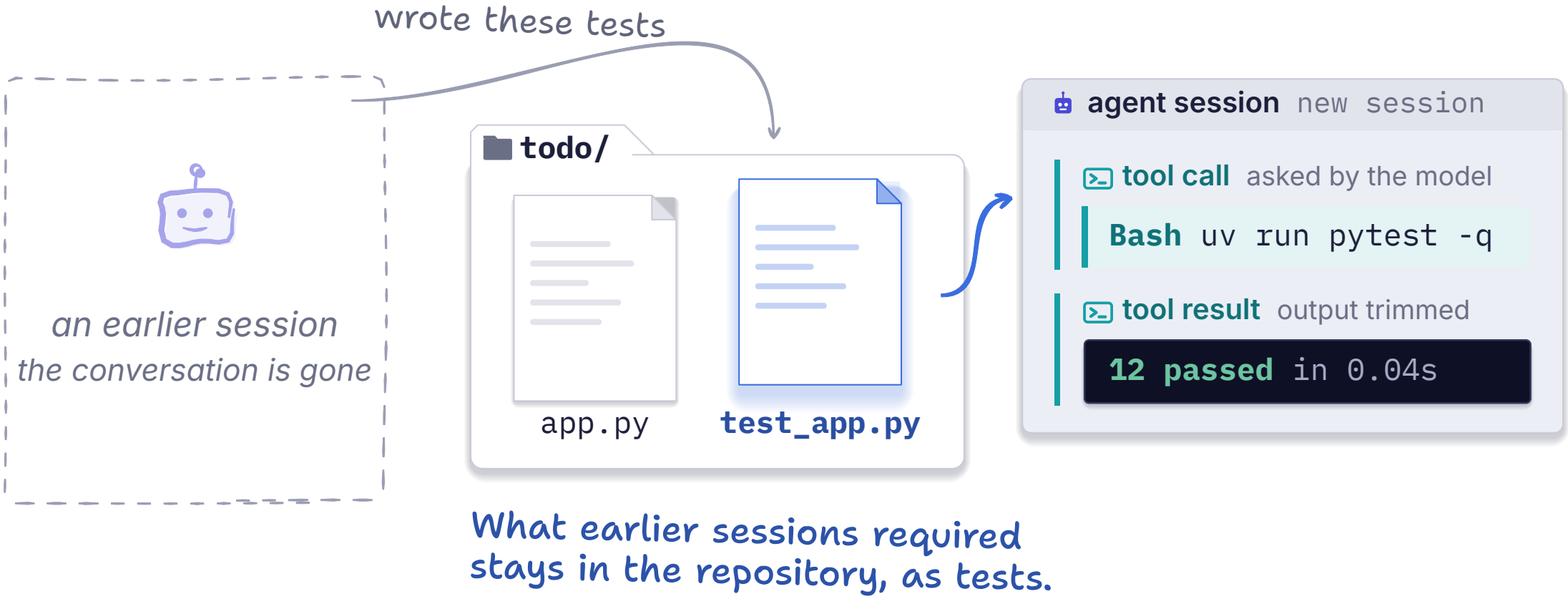
tests for	feature 1 **add items**	feature 2 **complete them**	feature 3 **save and load**
adding	✓	✓	✓
completing		✓	✓
saving and loading			✓
the combination			✓ new, and kept

todo.json

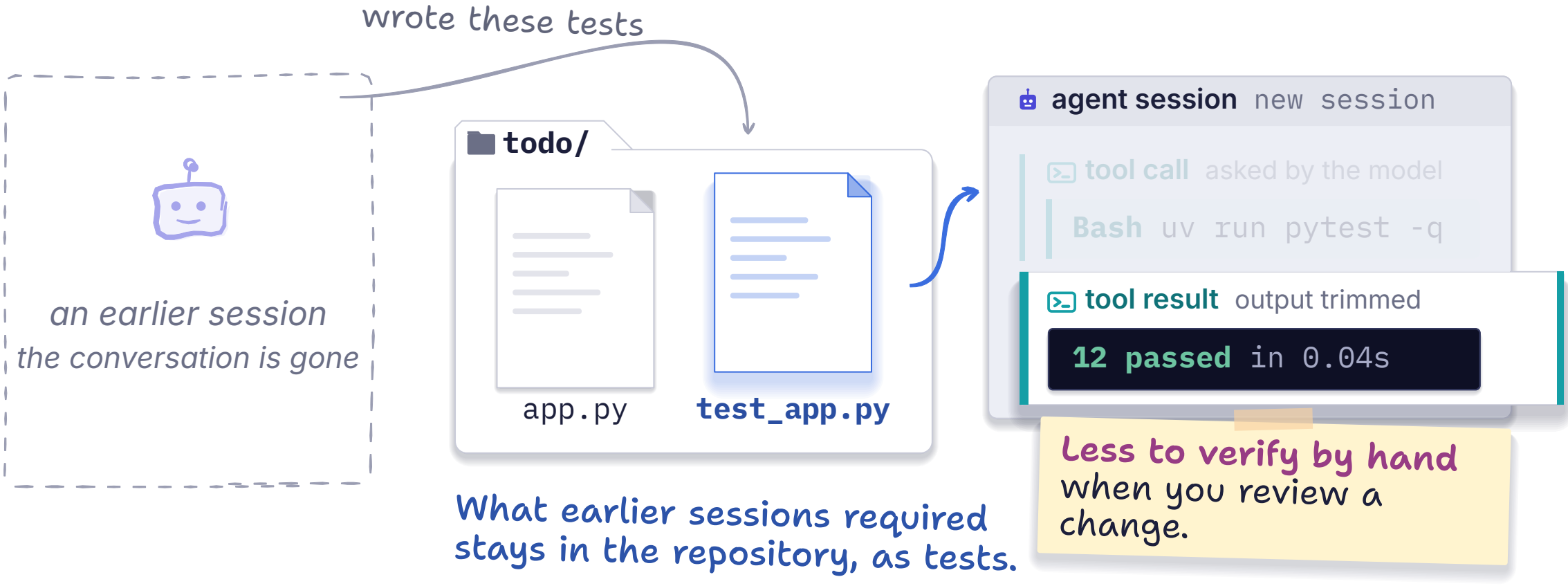
```
[{"title": "Buy milk"}]
```

Features share state?
Test them together,
and keep that test.

Kept checks outlast the conversation



Kept checks outlast the conversation



In a language you cannot read, tests still tell you something

src/lib.rs

```rust
fn share_done(items: &[Item]) -> f64 {
    if items.is_empty() {
        return 0.0;
    }
    let done = items.iter()
        .filter(|i| i.done).count();
    done as f64 / items.len() as f64
}
```

Written by the agent, in Rust.
Can you review it fluently?

what you can still state

an empty list → `0`

2 of 4 items done → `0.5`

shell

```
~/todo $ cargo test
test result: ok. 2 passed; …
```

Run the tests while you learn enough to read the code, or get help reviewing it.

How much checking? Ask whether the code has a future

a prototype you
delete on Thursday



a repository three
people push to every day



repeated use

how often the checks will run

How much checking? Ask whether the code has a future

checks cost

setup · tokens · running time · maintenance

they buy

dependability · unattended work

a prototype you delete on Thursday



a repository three people push to every day



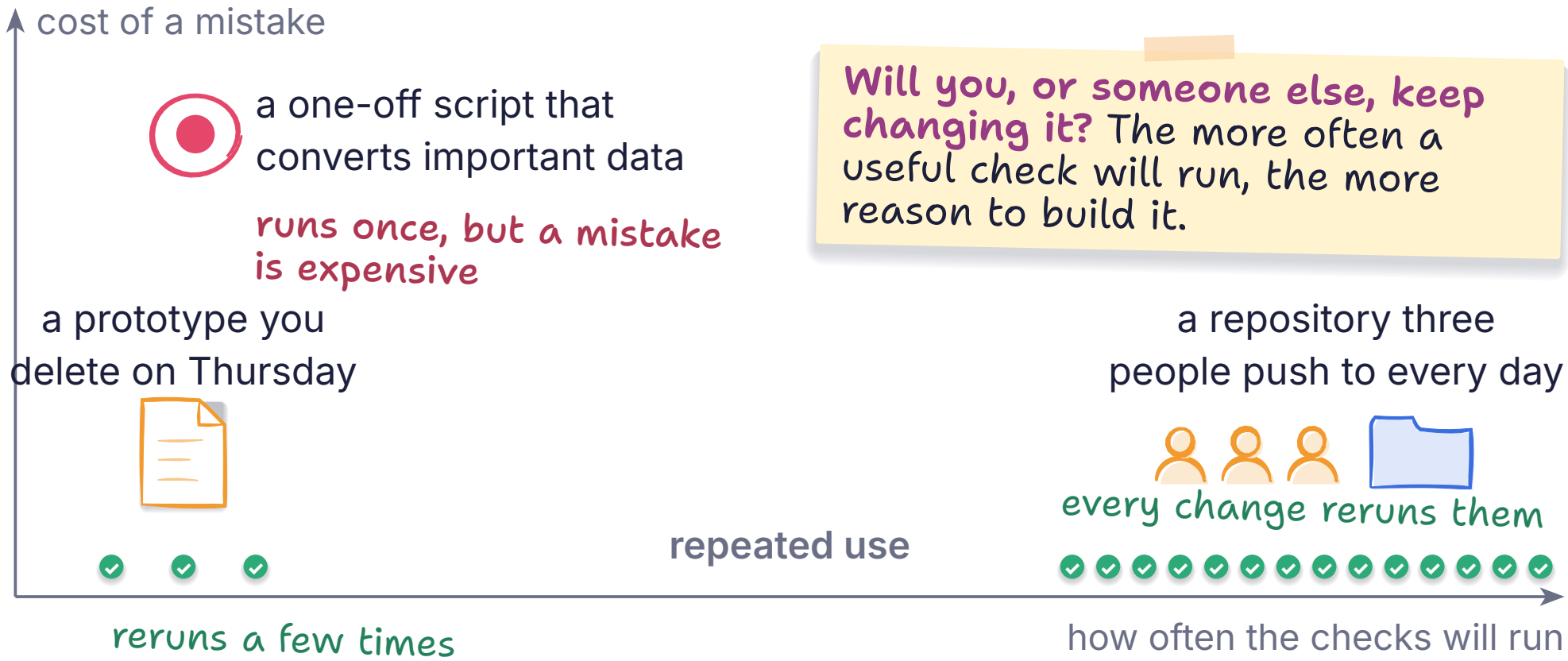
every change reruns them

repeated use

reruns a few times

how often the checks will run

How much checking? Ask whether the code has a future



Will you, or someone else, keep changing it? The more often a useful check will run, the more reason to build it.

Recap: formal tools, fuzzy tools, and kept checks



Formal and fuzzy

Formal tools are narrow and exact about specified conditions. Fuzzy tools are broad, but their judgments vary. Together they cover both.

Recap: formal tools, fuzzy tools, and kept checks



Formal and fuzzy

Formal tools are narrow and exact about specified conditions. Fuzzy tools are broad, but their judgments vary. Together they cover both.



Checks you keep

They retest earlier requirements after every change. They miss what they never exercise: when features share state, test them together.

Recap: formal tools, fuzzy tools, and kept checks



Formal and fuzzy

Formal tools are narrow and exact about specified conditions. Fuzzy tools are broad, but their judgments vary. Together they cover both.



Checks you keep

They retest earlier requirements after every change. They miss what they never exercise: when features share state, test them together.



Three formal checks

Tests compare behaviour with expectations. Types check that values fit where they are used. Linters check patterns and project rules.

Recap: formal tools, fuzzy tools, and kept checks



Formal and fuzzy

Formal tools are narrow and exact about specified conditions. Fuzzy tools are broad, but their judgments vary. Together they cover both.



Checks you keep

They retest earlier requirements after every change. They miss what they never exercise: when features share state, test them together.



Three formal checks

Tests compare behaviour with expectations. Types check that values fit where they are used. Linters check patterns and project rules.

The agent runs the checks, fixes what fails, and brings you a passing change.

Your review can focus on what the checks do not cover.

Happy testing.

See you in the next video.