Exercise 5 - Break a Test on Purpose, Then Diagnose It

Exercises · https://docs.functionize.com/hc/en-us/articles/43767017574679-Exercise-5-Break-a-Test-on-Purpose-Then-Diagnose-It

Lab handout — Lab 5
A printable worksheet for this exercise: the tasks, the exact prompts, a checkpoint and stretch challenges. Hand it round when you are onboarding a team.
Download the PDF (74 KB)

Everyone can read a green test. The skill worth having is reading a red one. In this exercise you will break a test deliberately, for a realistic reason, and work out why — using exactly the screens you would use on a real failure. About fifteen minutes.

This is the most useful exercise in the series. Do not skip it.

Step 1 — Build a test that passes

Create a guest basket test:

```
Go to https://ecom.functionizeapp.com/testsites/anaqa/index.php?route=common/home and open the Electronics > Air Conditioner category. Open the product Boltas 2 Ton 3 Star Split AC, verify its price is $966.00, then click Add to Cart and verify the cart shows 1 item totalling $966.00. Do not sign in - use the guest cart.
```

Approve it, let it build, and run it. It should pass.

Step 2 — Break it, realistically

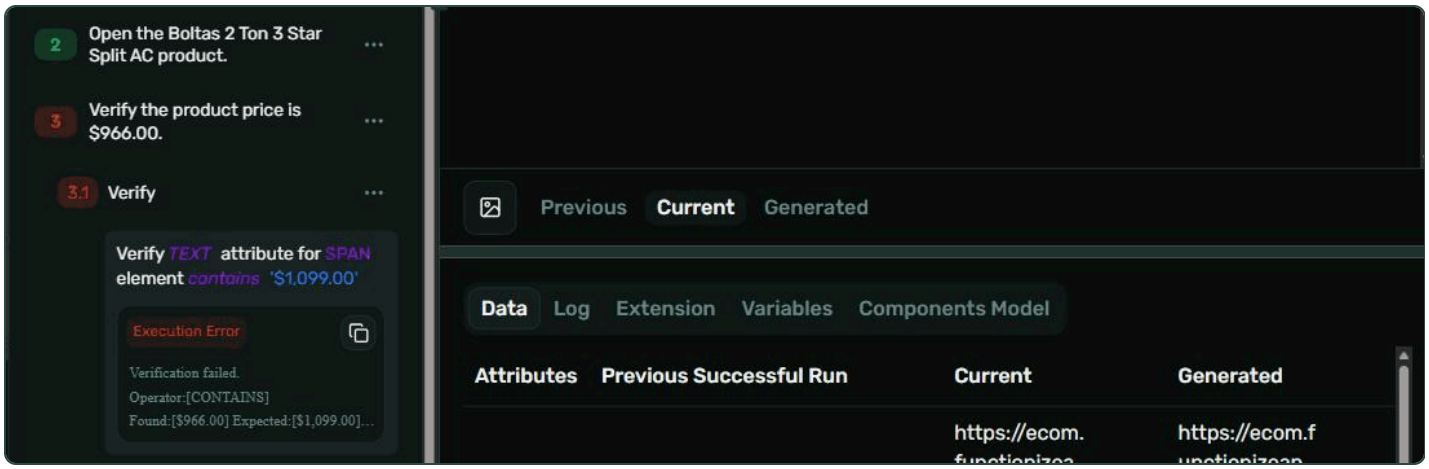
Now simulate something that happens constantly: a price change that the test was updated for before the store was. Ask for it:

```
The supplier has raised the price of this product. Update the price verification so it expects $1,099.00 instead of $966.00. Change only that expected value; do not re-run the test.
```

The agent makes the targeted edit and tells you it has not run anything. Note that it changed only the product-price check and left the separate cart-total check alone — that precision matters in a moment.

Step 3 — Run it and find the failure

Click **Run Test**. It fails:

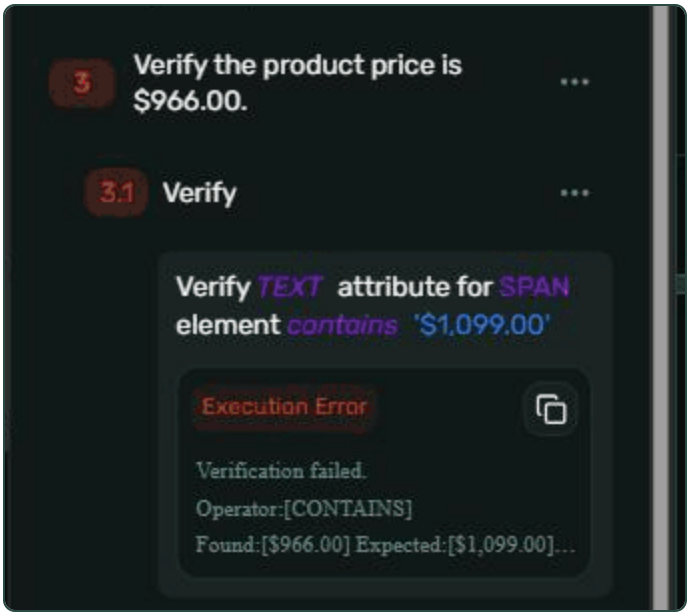


Before reading on, practice the first question: *which step is red, and did the ones before it pass?*

Everything that navigates to the product passed. Only the price check is red. That single observation eliminates a whole category of guesses — not a broken URL, not a navigation problem, not a timing issue getting to the page.

Step 4 — Read the error

Expand the failing step. Underneath the action is an **Execution Error**:



Read it in three parts:

- **The action** — what the step was trying to prove.
- **The operator** — `Operator:[CONTAINS]`, how it compared. A `CONTAINS` check failing is a strong signal: the expected text was not present anywhere in that element.
- **What it found** — `Found:[$966.00] Expected:[$1,099.00]`.

That last line is the answer. The store charges $966.00; the test expected $1,099.00.

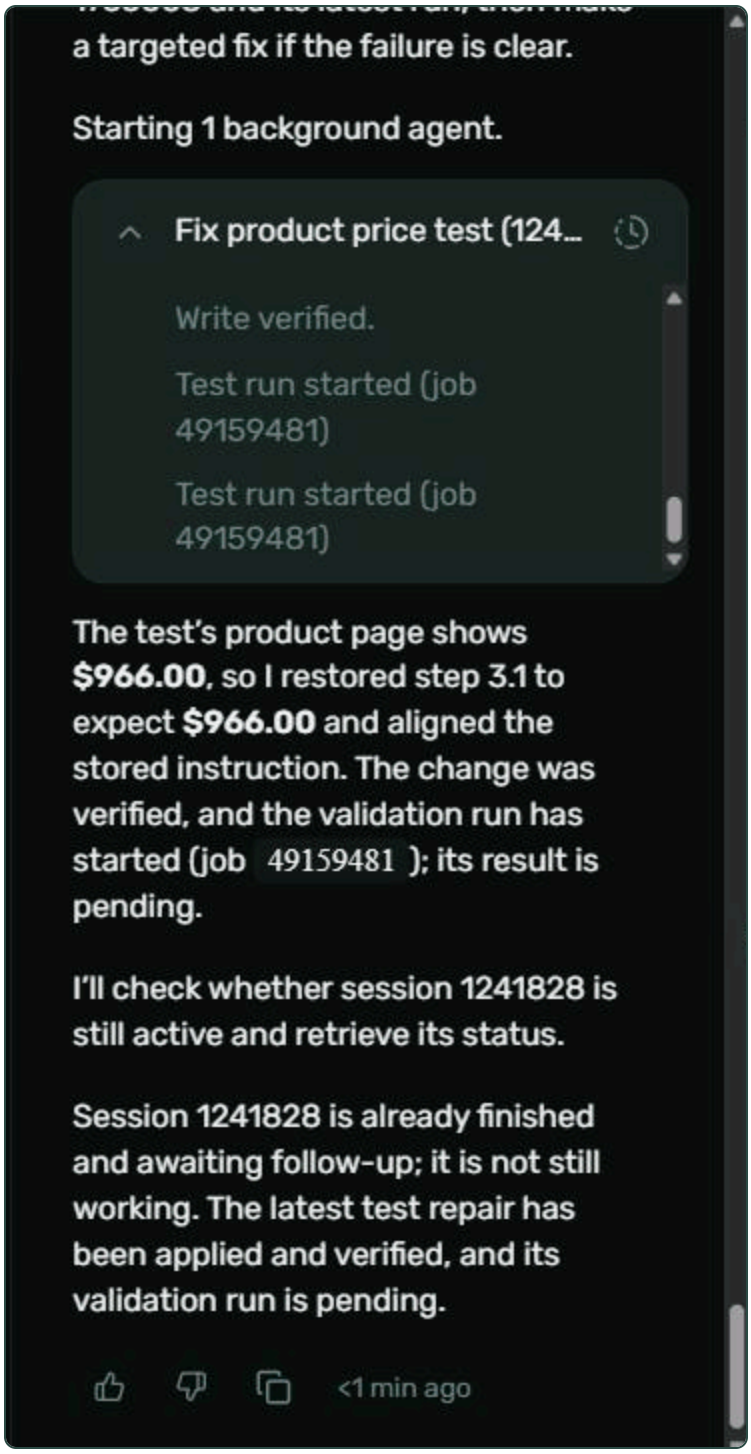
Now check the **Data** tab. Current matches Previous Successful Run — the store did not change. The expectation moved, which is exactly what you did in Step 2.

Step 5 — Ask the agent

Click **Fix Test** in the header. It is a shortcut that sends "*Fix this test for me*" to the agent — you could type the question yourself.

Step 6 — Read what it decides, and why

Here is the part that makes this exercise worth doing. The agent **refuses to fix it**:



Its reasoning: step 3.1 expected $1,099.00 but the product page still shows $966.00, and the separate cart-total check for $966.00 passed — so the test is not broken, the expectation is. *No changes were made and no test run was triggered.* It then offers the two real choices: point the check at a different element if the new price lives somewhere else, or change the environment's product data to match.

This is the distinction to take from the whole series:

- **Drift** — the test is trying to prove the right thing but looking in the wrong place, or too early. A moved button, a renamed class, a slow panel. Studio repairs this, often without being asked.
- **Disagreement** — the test asserts something the application does not do. No selector, comparison or timeout can fix that. There is nothing wrong with how the test looks; there is a difference of opinion about what is correct.

A tool that silently "fixed" the second case by rewriting your assertion to match whatever the page happened to say would be worse than useless — it would quietly delete the failure you wanted to catch. So it stops and asks instead.

A declined fix is not the product giving up. It is telling you the failure is a decision, not a defect in the test.

Step 7 — Resolve it

```
 It should be the main product price, and the store still sells it at $966.00 - my instruction was
wrong. Put the check back to expecting $966.00, align the stored instruction, and re-run the test to
confirm it passes.
```

It applies the change, offers to align the underlying instruction so a future regeneration keeps the right value, and runs the test to prove the repair. Green again.

The checklist, for real failures

1. Which step is red — and did the ones before it pass?
2. What does the Execution Error say it *found*?
3. Does **Current** differ from **Previous Successful Run**? If so, your application changed.
4. Does the step's screenshot show the page you expected?
5. Ask the agent. It has the run and it is faster than you are.
6. If it declines to fix, decide which is wrong — the test or the application. That one is yours.

Checkpoint

You are done when you have made a test fail, explained the failure from the Execution Error alone, and put it back to green.

What you learned

A failure is evidence, and the evidence is already collected. Most of diagnosis is reading it in the right order.

Next: Exercise 6 — Group tests into an orchestration.

Where to go next

- Exercise 6 - Group Tests Into an Orchestration — turn tests into a suite
- Diagnosing a Failed Test in Functionize Studio — the full diagnosis loop
- Keeping a Suite Green: Flake, Tiers and Who Owns Red — flake, tiers, and who owns a red result