Diagnosing a Failed Test in Functionize Studio

Studio Basics · https://docs.functionize.com/hc/en-us/articles/43766865298583-Diagnosing-a-Failed-Test-in-Functionize-Studio

When a test goes red in Functionize Studio, the run has already collected everything you need to explain it: the page as it was, the value the step expected, the value it actually found, and an agent that can be asked directly. This article works through a real failure from red step to resolution.

Watch: Functionize Studio 101: Diagnose and Fix a Failing Test

A failure taken apart on screen. Watch this once and the evidence tabs below stop being a wall of panels and start being a sequence you can follow.



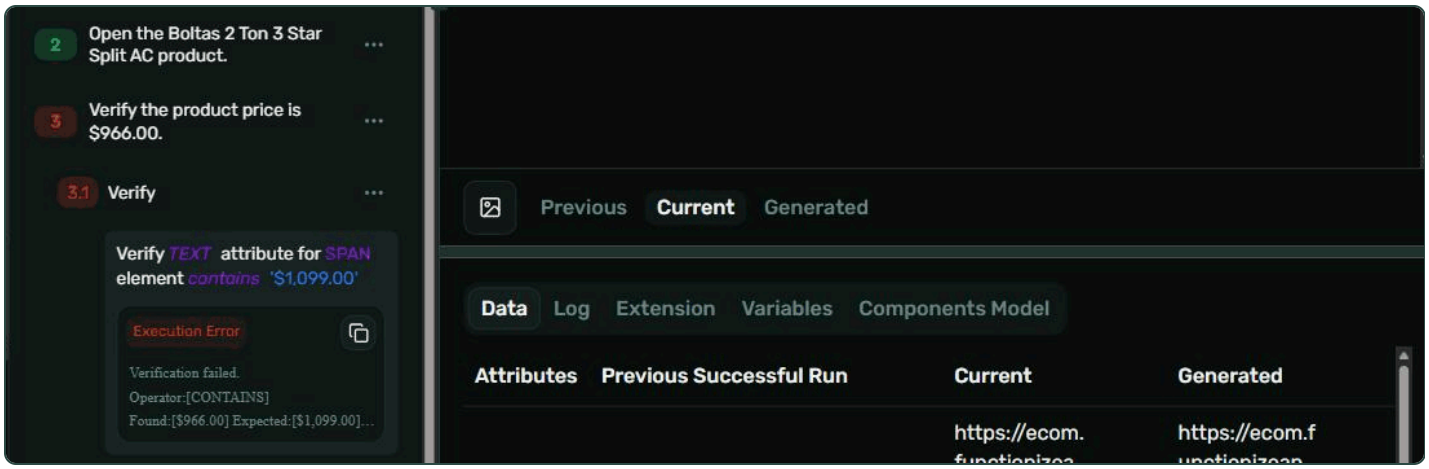
Watch: youtube.com/watch?v=x9hIrEHfMa0

The three places that explain a failure

- **The failing step** — which action broke, and the error it recorded.
- **The Data tab** — what the value was when the test last passed, against what it is now.
- **The agent** — which has the whole run in front of it.

Step 1 — Find the failing step

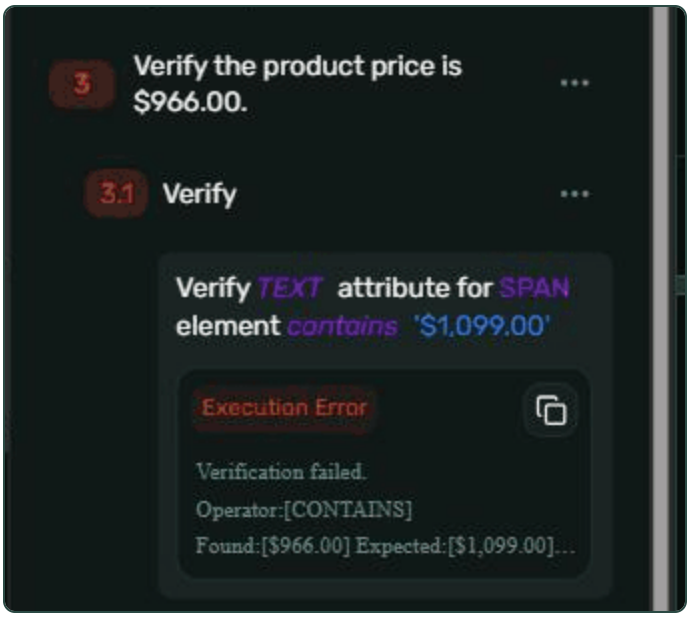
A failed run marks the failing instruction and the action underneath it in red, and the header gains a **Fix Test** button:



The red is deliberately narrow. The steps that navigate to the product all passed — so the failure is genuinely in the price check, not in getting there. That alone rules out a whole category of guesses.

Step 2 — Read the error

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



Read it in three parts:

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

That last line is usually the whole answer. The store charges $966.00; the test expected $1,099.00. The element was found, the step ran, the value simply does not match.

Check the Data tab before concluding

Compare **Current** against **Previous Successful Run**. If they differ, your application changed and the difference is the change. If they match and the step still failed, the expectation moved — which is the case here.

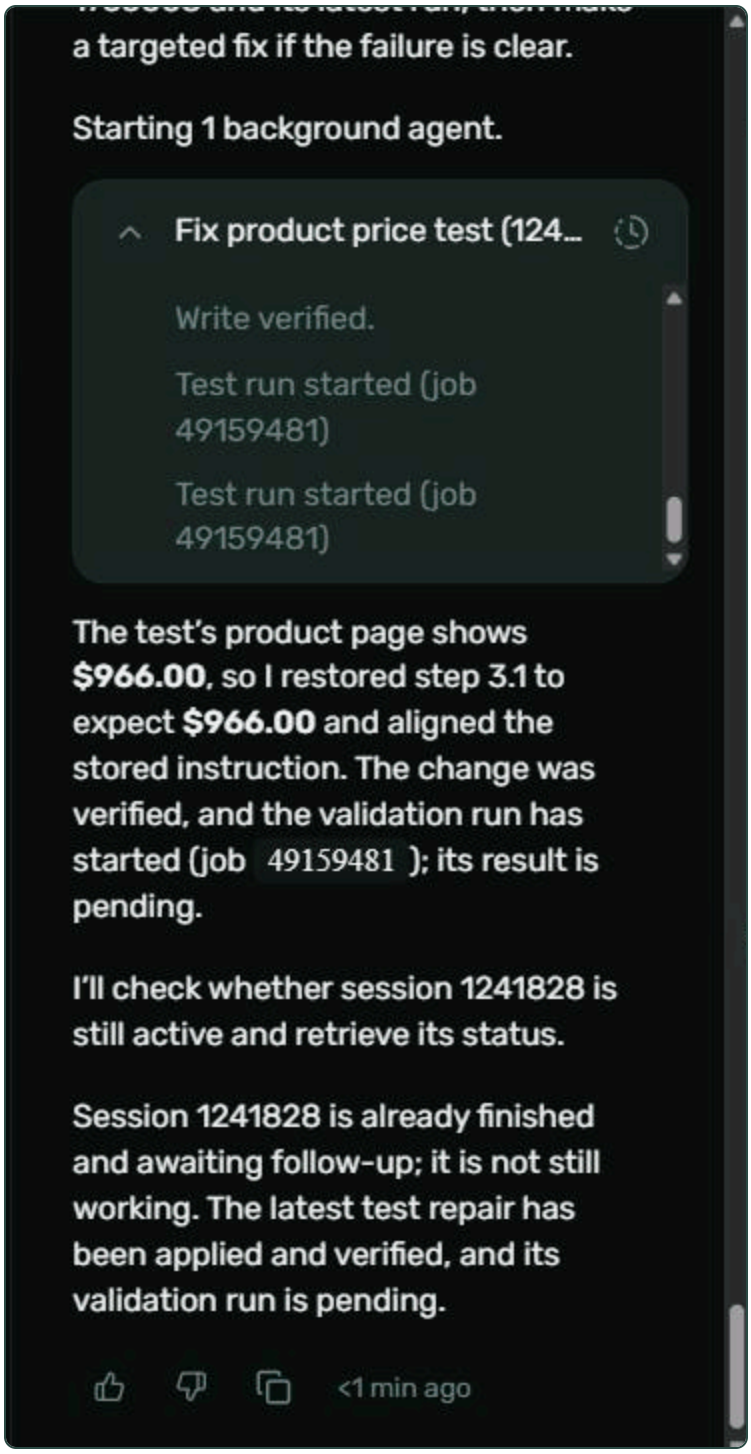
Step 3 — Ask the agent

Fix Test in the header is a shortcut: it sends "*Fix this test for me*" to the agent on the left. You can also just type the question — "why did step 3 fail?" works.

The agent starts a maintenance agent, which diagnoses the run and repairs the test where it can.

Step 4 — Read what it decided, and why

Here is where this example gets useful, because the agent **declined to fix it**:



In its own words:

> I couldn't fix the test without changing the price expectation you explicitly set. In the latest run, step 3.1 expected $1,099.00, but the product page still showed $966.00; the separate cart-total check for $966.00 passed. No changes were made and no test run was triggered.

It then offered the two real choices: point the check at a different element if the new price lives somewhere else, or change the test environment's product data to match.

This is the distinction worth taking from the whole article:

- **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-loading 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, because 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 the 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 the agent 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.

It does not always decline — and that is the point

Running this same scenario more than once, we saw Fix Test do two different things. Once it stopped and asked, as above. Another time it repaired the test itself, explaining that the product page showed $966.00 so it had restored the step to expect $966.00 and aligned the stored instruction.

Both are correct, and the difference is what evidence was available. A test carries two things: the step's expected value, and the underlying instruction it was generated from. When those two still agree with each other, the agent has a trustworthy source of truth and can repair a drifted step confidently. When the edit has made them disagree, there is nothing to repair toward without guessing — so it stops and asks you.

What it will not do in either case is silently invent a new expectation from whatever the page happens to show. Whichever path it takes, it tells you what it did and why, in the conversation attached to the test.

Step 5 — Resolve it

Answer in plain language. "The store still sells it at $966.00 — my instruction was wrong, put the check back and re-run to confirm" is enough. The agent applies the change, offers to align the stored instruction so a future regeneration keeps it, and runs the test to prove the repair.

Ask it to re-run. A repair you have not seen pass is a hypothesis.

A short checklist

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**?
4. Does the step's screenshot show the page you expected? A navigation that quietly went elsewhere is a common cause of a baffling assertion failure.
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.

When to raise a ticket

If the failure is not explained by the above — the runtime errored, the page never loaded, or the result contradicts what you can see in a browser yourself — raise a ticket with the test ID, the browser and version from the run header, and the time of the run. Those three let us find the exact execution rather than asking you to reproduce it.

See also Understanding a Test Run and Its Results, and the **Exercises** section, where Exercise 5 walks you through creating and diagnosing this exact failure yourself.