Proving the Math, Not Trusting It

How I made an AI's financial analysis trustworthy enough to act on

ERIC COHEN · CASE STUDY · STATUS: SHIPPED

I built a system that turns raw financial data into the kind of stock analysis an investor acts on with real money. Generating the numbers was never the hard part. **Trusting them was.**

So I built it to prove its own work. Every figure the system produces is verified by independent code before the file is saved, and I confirmed that check can't be fooled by trying to break it myself. **Across 23 runs, not one wrong number has reached a user.** Getting there meant finding a flaw that had been passing wrong numbers as proven, and rebuilding the check so it couldn't.

> *I don't trust AI output, only proof it can't fake.*

THE STAKES

An automatic check earns trust by default, so a hollow one is more dangerous than no check at all.

Trust is the linchpin: the analysis can be deep and well-built, but if you can't trust the numbers, you can't act on any of it. That trust has to hold automatically. A person could re-verify every figure by hand, but that defeats the purpose, since the value of the system is producing these analyses quickly. If someone has to recompute every cell to believe the result, the manual work just returns. So the automatic check is what **stands between a wrong number and a real-money decision.** When it passes a wrong number, that number reaches the investor looking proven, and the investor buys or sells on it.

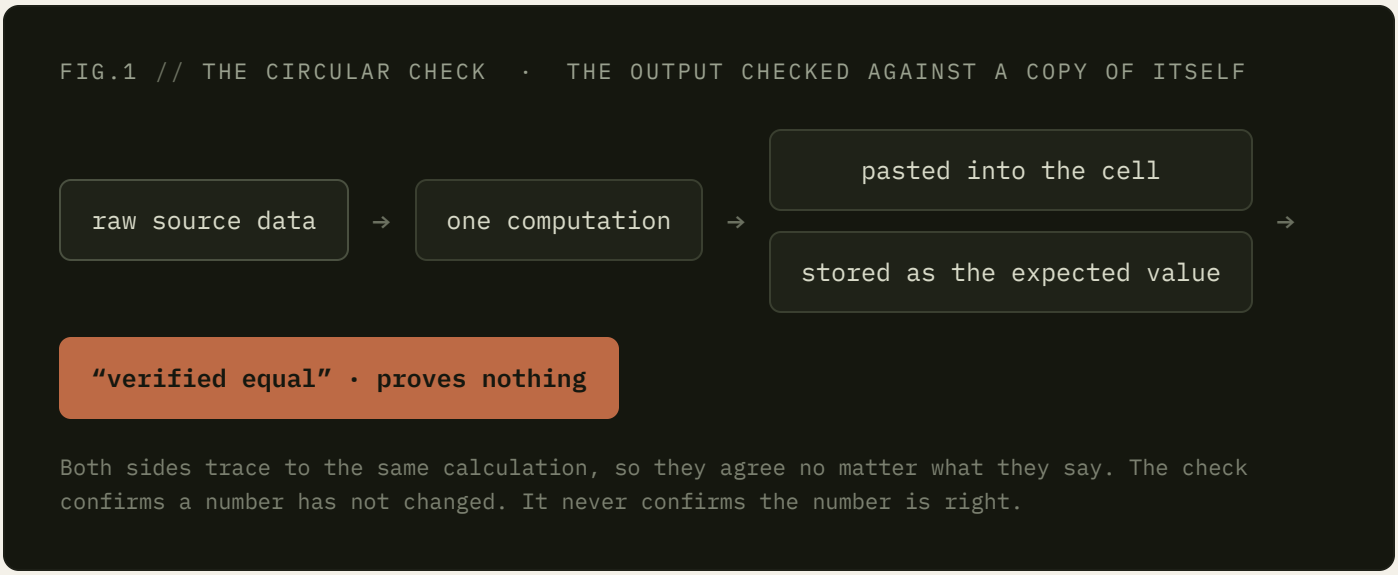
THE FLAW

Passing every test is not the same as being correct, and I caught the system's check grading its own work.

I designed this system, and I used AI to generate the analysis and the code behind it, including the code that checks its own work. The check it had written looked like the right design. It stored a known-good result for each figure, re-ran on every build, and flagged anything that drifted from it. Lock in a trusted answer, get warned the moment something changes it. Dozens of figures, dozens of assertions, green every time.

The flaw was in where that result came from. The AI had generated it from the same calculation that produced the live spreadsheet, so the check was never comparing the output to an independent answer. It was comparing the output to another run of the same code. When both sides come from one source, they agree every time, whatever they say. The check could tell me a number had not changed. It could not tell me the number was right. **Consistency, not correctness.**

I did not find this because the check told me. It never failed, so it never pointed at itself. I found it because a number looked wrong to me on a run where every test was green, and those two facts could not both be true. AI-generated code is convincing, so I treat a clean pass as a claim to test, not a result to trust. I followed the suspect numbers back through the check until I reached the calculation it was meant to be independent of.



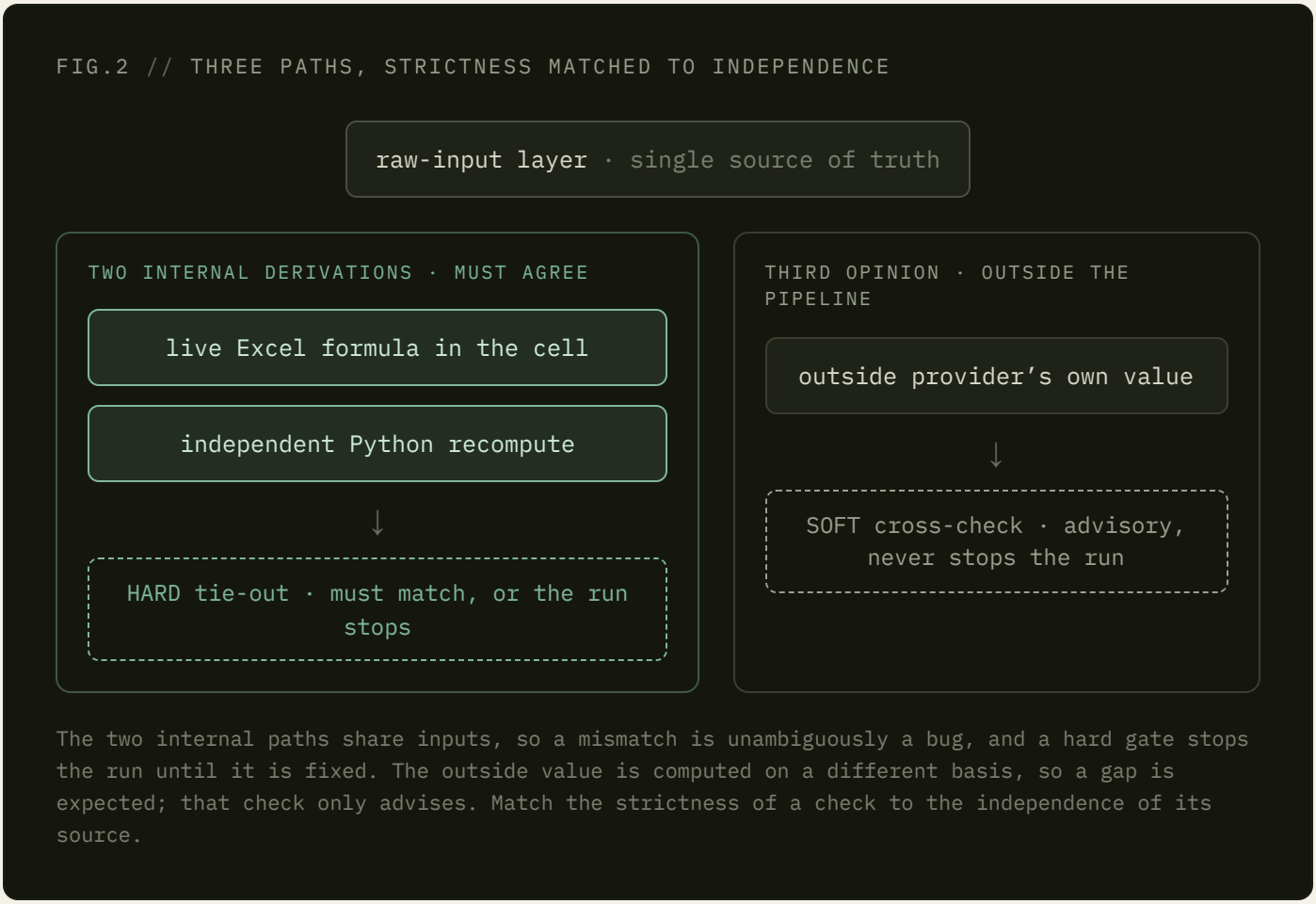
THE FIX

The fix was more than a quality patch: one design choice made the spreadsheet a better tool for the investor and let me prove every number in it.

This built on a decision I had made early. I had designed the spreadsheet as a live model: every figure is a live Excel formula, each one traced back to a single table of raw inputs that holds the source data and nothing else. For the investor, that is the difference between a report and a model. They can click any number to see how it was built, follow it to the raw data, or change an input and watch the whole sheet recalculate. While fixing the check, I saw that the same structure could solve the harder problem. A model built from one set of raw inputs is something a separate program can rebuild from scratch.

That rebuild is the fix. A separate Python script reads the same raw inputs and recomputes every figure in its own code, never reusing the spreadsheet's formulas. Each number now comes from two places that share nothing but the source data: Excel's formula engine inside the spreadsheet, and the Python recompute outside it. **They share no code, so they cannot share a mistake.** The check runs on every build and covers every number, not a spot-check. When the two agree, the agreement is real; when they disagree, the run fails before the output reaches anyone. Two independent paths to one number is proof. The old check only ever had one.

A recompute works here because the question has one correct answer. Arithmetic on fixed inputs gives the same result every time, so a deterministic program is the right instrument: it reproduces the number or it does not.



BLOCK OR WARN

Not every disagreement means a number is wrong, so I had to decide which ones should stop the run until they're fixed and which ones should only warn.

The two internal numbers come from the same raw data and the same definition, so they should produce exactly the same result. If the Excel formula and the Python recompute disagree, something is broken: a value got pasted over a formula, an input was misread, a calculation drifted. There is no innocent explanation, so **a mismatch here stops the run.** The number does not ship; the error gets fixed and the run repeats until the two paths agree.

The third number is different in kind. It comes from an outside data provider, computed on its own basis: a different definition of earnings, a different moment in time, a different way of combining the parts. A gap between my number and theirs is usually not an error, just two reasonable methods landing in different places. If I treated that gap like an internal mismatch, the system would block honest runs constantly and train me to ignore it. So the outside comparison only raises a flag for me to weigh. **It never stops the run.**

This is the same judgment in another form: **fit the check to the question.** Whether a number is correct has one right answer, so a deterministic program is the right tool. Other questions have no single right answer. Whether a set of companies is a sensible peer group, or whether one looks misclassified, is a matter of judgment, and there I let the AI make the call, because judgment is the only instrument that fits. Deterministic code is not better than AI judgment, or worse. Each is right for a different kind of question, and using one where the other belongs is its own mistake.

BREAKING IT ON PURPOSE

A check that passes has proven nothing. I trusted this one only after I tried to break it and it caught me every time.

This whole project started because a check passed while the answer was wrong. So a new check that merely passed would not convince me of anything. The bar was harder: I had to break the output on purpose and watch the check stop it, then feed it correct output and watch it stay quiet. So I went at it one case at a time:

Break the math any way you like: point a formula at the wrong input, change an operator, alter an exponent. However it goes wrong, it computes a number the independent recompute doesn't, and the mismatch fails the run. CAUGHT

Paste a fixed number where a formula belongs. It is rejected even when the number is right, because every computed figure has to be a live formula tied to the raw data, not a value someone typed in. CAUGHT

AND THE OTHER HALF · LEAVING CORRECT NUMBERS ALONE

Run it on **cells that are right as they stand:** a few meant to be plain values rather than formulas, and one figure shown as a label on purpose. NOT FLAGGED

Catching errors is half of it. Leaving correct numbers alone is the other half, and it matters just as much, because a check that flags good cells gets ignored the same as one that misses bad ones. A check I can break on demand, that catches the break and stays quiet on valid output, is one I can stand behind.

WHAT CHANGED

The investor now gets a model they can trust with an investment decision, with every figure proven before it ships.

Green now means proven. Every figure in the model is a live formula, and an independent recompute checks each one before the file is saved. The investor can click any number, trace it to the raw data, change an input, and watch the whole model rebuild. It is a working model they can take apart and trust. Across 23 runs, not one wrong number has reached them.

The approach reaches past finance. Any system that puts AI-generated numbers in front of someone who has to trust them needs the same backbone: **compute the answer a second way, tie the two together, and prove it by trying to break it.** The method fits anything with a number to check. The rule under it is wider: a check is only worth something if it reaches the answer independently of whatever produced it, and only worth trusting once you have tried to fool it. That holds for any AI output a person has to act on.

This case study covers the architecture and the reasoning behind it. The underlying calculations, data sources, and exact check logic are intentionally omitted.

Eric Cohen · cohenemc@gmail.com