I Got the AI Result I Wanted. Then I Ran It Nine More Times

August 30, 2026 · Updated: August 31, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** I had high hopes for frontier models improving my local 27B. One read my production logs and wrote a skill that scored 10.6 points over baseline, exactly the result I was hoping for. It took nine more compilation runs and $4.97 to find out the number was fake, and to see why almost nobody would catch it.

📚 **More on this topic:** Intent Engineering for Local AI Agents · Session-as-RAG · The Hallucination Feedback Loop · Gemma 4 vs Qwen 3.6 on the 3090

I had high hopes for frontier models improving my local LLM. Not fine-tuning it, not replacing it. Just reading its logs and writing down what it keeps getting wrong, so the small model on my own hardware gets better at the job it already does.

If that works it is close to free. A few dollars of API credit buys a text file, the text file goes in the system prompt, and the 27B on my desk classifies better forever. No retraining, no new hardware, and the artifact is readable, so I can argue with it.

So I ran it. I took 24 real traces from mycoSwarm's intent classifier, each one a genuine user message with the correct label and what my local model actually predicted, and I handed them to Claude Opus 5 with a simple brief: write procedural rules that make this 27B more accurate.

It came back with 91 lines of decision procedure. Ordered tests, tie-breakers, a section on which cues not to misread. It had worked out, correctly, that my deployed prompt treats the words "you" and "your" and "we" as retrieval signals when they are just how a person addresses an assistant, so the model fires RAG at anything introspective.

That is a real bug. I have looked at that prompt for months and I had not named it that cleanly.

Then I scored it. Baseline on my 47-item test split was 48.94%. The compiled skill scored **59.57%**.

Up 10.64 points. A frontier model read my own logs, spent 24 cents, and made my local model measurably better at a task it does hundreds of times a day. That is the result I was hoping for. I sat there feeling pretty good about it.

The turn

I only know that number is fake because I paid to do it nine more times.

That was not suspicion. It was in the plan from the start, for a boring reason: the paper that prompted this work reports a single compilation run per configuration, and I wanted to know the run-to-run spread before I trusted any single number. So I ran the identical bundle through the identical model ten times and scored all ten separately.

Skill	Exact	vs baseline
g24_05	42.55%	−6.4 pp
g24_00	46.81%	−2.1 pp
g24_02	48.94%	0.0 pp
g24_06	48.94%	0.0 pp
g24_08	48.94%	0.0 pp
g24_01	53.19%	+4.3 pp
g24_03	53.19%	+4.3 pp
g24_04	53.19%	+4.3 pp
g24_09	53.19%	+4.3 pp
g24_07	**59.57%**	**+10.6 pp**

Ten runs, byte-identical input, and the answers range from six points worse than no skill to ten points better.

The mean is 50.85%, so the honest effect is **+1.91 points**. The standard deviation across those ten, which is the compiler's own noise floor, is **4.65 points, with a 95% confidence interval of 3.21 to 8.50** — ten samples does not pin a standard deviation tightly, and quoting it as one clean number would be the same mistake I am writing this article about.

My +10.64 result sits **1.88 standard deviations above the compiler mean**. It is not a better skill. It is the top of ten draws.

Twenty compiled skills scored against the baseline, nearly all inside the compiler-noise band

I wrote two pass conditions down before any of this ran, which is the only reason I believe the answer. The delta had to clear two standard deviations, and a paired bootstrap interval on the

delta had to exclude zero. It needed to beat 9.29 points and managed 1.91. The bootstrap came back **[−4.26, +8.30]**, straddling zero.

Both fail. There is no effect here.

Why nobody would catch this

Here is the part that has stayed with me.

The obvious defence against a fluke is a validation split: compile a skill, score it on held-out items, keep it only if it improves. That is exactly what the published methods do, and it sounds airtight.

Mine is 15 items. One item moves that score **6.67 points**. The compiler's noise floor is 4.65 points. The gate's smallest possible step is larger than the thing it is supposed to detect, so it cannot tell a genuinely better skill from a lucky draw. It will happily wave through either.

And luck is common here, not rare. Five of my ten runs scored above baseline on test. I never scored them on the validation split, deliberately, so I cannot tell you exactly which would have cleared a gate. What I can tell you is that half my draws landed on the good side of a line, and a gate measuring in 6.67-point steps has no way to know that is what happened.

Run the compiler once and you get one of those ten. Half the time it looks like a win. The gate agrees. You write it up.

This is invisible by construction. It is not a reasoning error in the published work, and it is not something a careful reviewer would catch, because the evidence that it happened does not exist unless somebody pays to generate it. Ten compilation runs where one would do is money spent proving you did not need to spend it.

What I actually ran, and what I did not

Now the caveats, because at this point they matter.

Google Research published WikiSkill on August 27, three days before I ran this. It compiles agent experience into a persistent wiki, proposes skills from that wiki, and keeps the ones that improve a validation score. The results are strong and the paper is careful.

I did not replicate it. I did not run their benchmarks, I did not build the wiki layer, I did not implement gating, and I did not run multiple rounds. Nothing here refutes a single number they published.

What I ran was one task, mine, one-shot, measuring four things their paper leaves open: token cost, run-to-run variance, how much corpus you need, and what happens when the model doing the compiling is not the model doing the work.

The task is intent classification: one user message in, one JSON object out with three enum fields deciding whether to answer directly, search the web, or hit RAG, and where to look. There are 89 hand-corrected gold pairs mined from real chat logs. Two of them now get caught by a deterministic regex before the model ever sees them, and a third turned out to be a byte-identical duplicate of another entry, which I found only because I grepped the assembled prompt rather than trusting my own split code. That leaves a working corpus of 86: 24 for compiling, 15 held for a gate I never used, and 47 for test.

Executor was Qwen3.6-27B at Q4_K_M on an RTX 3090, served through llama.cpp with thinking suppressed, matching my deployed config exactly. Compiler was Claude Opus 5 through the API. Skills went into the system prompt in full with no retrieval, the same way the paper does it and for the same reason: it removes retrieval failures as a confounding variable.

The full run record, all twenty skills verbatim, and the per-call token accounting are in the repo. Every number in this piece comes from that record.

Twenty documents, one strategy

Once you know the scores are a distribution, the other results start explaining themselves.

I expected the ten same-input skills to be near-copies. They are not. Their 5-gram overlap runs from 0.001 to 0.017, which is to say they share essentially no phrasing. One is a short-circuit ladder, STEP 1 through STEP 4 with tie-breakers. Another opens by asking whether the message is dialogue at all, runs seven ordered tests, and closes with a section headed DO NOT MISREAD THESE CUES. They read like they were written by different people.

They also all found the same bug. Every one of them worked out the “you”/“your”/“we” problem.

So does writing it differently make it work differently? Across the 45 pairs, the correlation between textual distance and score distance is **r = +0.118, permutation p = 0.440**. Nothing. Twenty documents that share no sentences and encode the same insight score the same, within noise. The variance in my table is real and it does not live in the text.

Eight traces is worse than nothing

I ran the same compilation at three corpus sizes to see how much log you need before this produces anything useful.

Traces	Skills	Mean exact	Delta
24	10	50.85%	+1.91 pp
16	5	49.79%	+0.85 pp
8	5	**46.38%**	**−2.55 pp**

More traces is better, and eight traces produced skills that scored below no skill at all.

Both halves need saying. The trend is monotone, which is suggestive. The whole range from 46.38 to 50.85 also sits inside one standard deviation of compiler noise, so I cannot tell you the trend is real. What I can tell you is that eight stratified traces, which is the sampling budget the paper uses to feed its wiki maintainer, made things worse on this task rather than better.

The cost metric is counting the wrong thing

The published cost figure counts optimizer API calls, and shows the method needs a constant number of them per iteration regardless of training set size. That is true of the slice it measures. Here is everything I actually spent:

Bucket	Where	Input tokens	Output tokens
Baseline rollout	local	314,960	9,622
Thinking-on probe	local	93,486	64,347
Compilation, 20 calls	**frontier**	**66,495**	**193,248**
Skilled rollout, 20 × 47	local	**1,913,419**	17,975
Gating	—	0	0

Running the skills ate **1.91 million local input tokens, 7.4 times the whole compilation**. Optimizer-call counting sees none of it, because none of it is an optimizer call. That is not a rounding error, it is where the compute went.

Counting calls hides something else. Those ten identical-input calls produced between 9,257 and 14,219 output tokens, a 1.54x swing on the same prompt, and they differ from each other by

more than the entire input cost of all twenty calls put together. One API call is not one unit of anything.

Most of that output I never see. The visible skill is roughly 1,500 tokens against a mean of 9,662, so about 84% of what I paid for was reasoning the model omits from the response by default. Input came to 2.8% of the bill and output 97.2%, total $4.97.

One practical warning. My character-based token estimates ran about 30% low against the real tokeniser: I guessed 2,500 to 2,860 for the largest bundle and it counted 3,693. Structured content packs denser than prose, and a wiki of enum tables and JSON examples is about as structured as text gets.

Then there is the part that never stops costing. Baseline inference takes 651.7 prompt tokens. With a skill appended, 2,035.6. **Every future inference pays 1,383.9 extra tokens, forever**, scaling with the skill (r = 0.922 against character count). Divide the compilation by that tax and you break even at **188 local inferences**.

Do not read that as a dollar figure. Compilation is paid once, in frontier tokens, at frontier prices. The tax is paid forever, in local tokens, at GPU-seconds on a card I already own. Summing them would produce a number that means nothing.

The one thing that worked

All twenty skills fixed the schema.

My baseline emitted one invalid enum across 47 items, `{"mode": "creative"}`, which is not in the vocabulary. Every single skill scored 47 out of 47 valid. Twenty for twenty.

It is worth zero accuracy points. The sanitiser already caught that field and rewrote it to the correct default before anything downstream saw it. Compilation reliably fixed the one defect production had already hidden from me.

I am not going to inflate that into a win, but it is the only consistent effect in the entire run, and it does say something about what these compilers are good at. Give them a spec and they will enforce it. Ask them to out-think a prompt on a genuinely hard judgement call and, here at least, they did not.

What this does not show

The limits cut against my null as hard as they cut against anyone’s claim, and they are worse now that you have a reason to care.

Start with the test split: 47 items. One item moves the score 2.13 points, and the baseline's own bootstrap interval runs [34.0, 63.8], nearly fifteen points either side. That is good enough to catch effects the size the paper reports on its benchmarks, +20 to +40 points. It is nowhere near good enough to rule out a real +3. If skill compilation genuinely buys a few points on this task, my setup cannot see it, and I would not know.

The label space is only half exercised. Real traffic in these logs touches two of the four tool classes and three of the four modes, so nothing here speaks to the ones that never appear.

Then the split itself, which is the weakness I like least. My compile and test items come from the same three chat files, the same user, the same few weeks. There is no distribution shift at all, and that flatters the skill rather than my null result: a compiler writing rules for traffic it has already seen has an easier job than one facing new material. The harder test would use a genuinely held-out source and I did not run one.

And the scope is one task, one model, one shot. No wiki, no gating, no rounds. The paper's central claim is that accumulated knowledge compounds across iterations, and I tested none of that. I would still build the loop if the one-shot had cleared its floor, because the argument that a persistent wiki amplifies a working proposer is plausible. It did not clear the floor, and their own ablation shows the wiki amplifies a proposer that already works rather than creating one from nothing.

Where this leaves me

Nobody runs the compiler ten times. I need to be clear that this is a rational choice and not laziness: when the thing works, nine extra runs buy you nothing but a bill. The only way to discover that your result was a draw from a distribution is to spend money finding out it might not have been, on the assumption that it was.

Which means this failure mode is invisible by construction, and I now assume that most published skill-compilation results are single draws from a spread nobody measured. Not wrong. Unmeasured, which is a different and more annoying thing.

I still think the idea is right. A frontier model reading a small model's logs and writing down what it gets wrong should work, and the skill it wrote me correctly diagnosed a bug I had been staring past for months. The paper may well be right too, on their benchmarks, with their wiki, across their rounds.

What changed is narrower and it is about me. I cannot tell from one run. I got the number I was hoping for, believed it for about an hour, and it was noise. If you have a skill-compilation result you are pleased with and you generated it once, you do not know either.

Everything above is about one substrate: the skill compiled into a prompt, where the cost is 1,383.9 tokens on every call and the artifact is a text file. There is a second substrate that puts the skill in a LoRA adapter instead, where there is no prompt tax at all — and where, as far as I can tell, nobody has run the reproducibility check either.
