NEWS · JULY 14, 2026

A Local AI Wrote a CD Calculator — and Why I Still Had Claude Check the Math

By Dr. Holger Flick



A model small enough to live entirely on my laptop built a working, charted CD calculator from a one-line prompt. It was fast, it looked great, and the arithmetic was correct. It was also, quietly, the wrong tool — and spotting that took a second opinion from the cloud.

Everyone is talking about Qwen's latest release — Qwen3.6, the 27-billion-parameter model from Alibaba's

Qwen team. The story is that a 4-bit quantized build runs on hardware you already own and produces genuinely good code, especially in JavaScript and Python. I wanted to know two things: is that story true, and how far do I trust the output? So I gave it a small, honest task — a certificate-of-deposit (CD) calculator — and then handed the result to Claude in the cloud to referee.

What follows is the whole loop: a fast local model, a plausible-but-wrong first draft, three rounds of iteration, and a cloud model catching the one thing that mattered. The payoff isn't "local bad, cloud good" — it's the opposite. Local AI is now good enough to be a daily tool. You just have to keep verifying the parts that can hurt you.

> **I ran the same test on a second local model**
> After this post, I gave the *identical* prompt to a completely different local model — Ornith 1.0 35B, a mixture-of-experts model — and compared the two head to head. Spoiler: it made the *exact same* domain

> mistake, then repaired itself impressively. The full comparison is here: Two Local Coding Models, One Prompt.

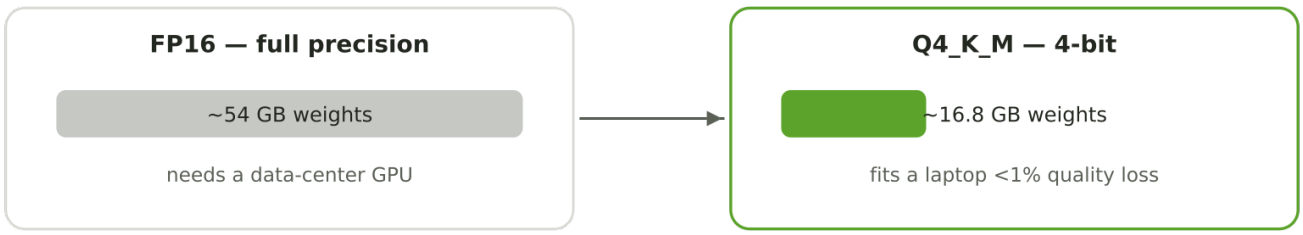
The model: small enough for your desk, good enough to matter

Before the task, a word on what "runs locally" actually means in 2026, because the numbers are the point.

Qwen3.6-27B is a dense 27-billion-parameter model released on 22 April 2026 under the permissive Apache 2.0 license. "Dense" means every parameter is active on every token — as opposed to a mixture-of-experts (MoE) model that only lights up a slice of itself per token. On SWE-bench Verified, a standard agentic-coding benchmark, it scores **77.2%**, up from 75.0% on the previous 27B release — and, notably, it edges out Alibaba's own much larger 397B MoE model on that benchmark (source).

The reason it fits on a desk is **quantization**: storing the model's weights at lower numeric precision. A full-precision (FP16) weight uses 16 bits; a 4-bit quantized weight uses four. The recommended `Q4_K_M` build of Qwen3.6-27B weighs about **16.8 GB** and runs in roughly **18 GB** of RAM or VRAM — with under 1% quality loss versus full precision (source).

Let me show the trade-off visually, because it's the whole reason this is possible.



Quantization trades a sliver of quality for a model that fits in consumer memory

The takeaway from that diagram: the 4-bit build is roughly a third of the memory footprint, and the quality you give up is small enough to be a rounding error for most tasks. That is what turns a research-lab artifact into something you run on a Tuesday.

For context, I ran this on an Apple MacBook Pro with the M5 Max and 128 GB of RAM. Apple Silicon's unified memory architecture — where the CPU and GPU share one pool of high-bandwidth memory instead of copying data across a bus — is a genuine advantage for large models, because the whole model sits in memory the GPU can reach directly. But note the honest scoping: **you do not need my machine.** At ~18 GB, this model runs on a single high-end consumer GPU (an RTX 5090) or a mid-tier Mac (source). My 128 GB is overkill for this particular model; it just means I don't have to think about it.

> **How local stacks up against the cloud, briefly**
> Today's flagship cloud models — Claude Opus 4.8 and Sonnet 5, or OpenAI's GPT-5.6 — run **1-million-token** context windows and are trained at a scale no local model matches. A 27B model on your desk is

> playing a different game: smaller context, less world-knowledge, but *private, free at the margin, and always available*. The rest of this post is about where that difference actually bites.

A fair word on the alternatives

Qwen isn't the only credible local option, and mentioning them costs nothing but earns trust.

If you want to run models like this yourself, the open-source tooling is excellent: Ollama and llama.cpp both load `Q4_K_M` GGUF builds directly, and LM Studio gives you a GUI if you'd rather not touch a terminal. Other strong local families in the same weight class exist too — Meta's Llama, Mistral, Google's Gemma. The point of this post isn't "Qwen wins"; it's that *any* of these has crossed the threshold from toy to tool. Pick by fit: license, language strength, and what your hardware can hold.

The task: one sentence, one working app

I kept the prompt deliberately casual — the way a busy person actually asks — to see how much the model would fill in.

Here is exactly what I typed:

```
implement a simple webpage in html with vanilla javascript and css. no
dependencies. to enter a money value and a cd percentage with how much
interest you earn in each month with a bar chart. i wanna be able to enter
the number of months with the cd percentage that a bank gives me. like you
get a 4.5% CD
```

The model produced a single self-contained page — HTML, CSS, and JavaScript in one file, no build step. It compounded the interest monthly and rendered a bar chart. The chart it hand-rolled overlapped and looked rough, so I added one follow-up:

```
please use chart.js for the chart
```

This is a bigger deal than it sounds. Chart.js is a capable but large library with dozens of chart types and a deep configuration surface; wiring it up correctly by hand takes real time. The model swapped in a properly configured Chart.js bar chart in one turn — exactly the kind of "I know this library so you don't have to" leverage that makes these tools worth having.

The result looked great.



This looks great on first look, but looks can deceive — a clean UI and a Chart.js bar chart showing interest accumulating month over month.

The chart shows interest accumulating over the months at the entered rate. Clean layout, sensible labels, a working live calculation. If I'd stopped here, I'd have shipped it. That's precisely the trap.

The referee: handing it to the cloud

I couldn't easily eyeball whether the *financial model* was right — the arithmetic looked fine, but "looks fine" is exactly the failure mode I was hunting. So I screenshotted the running app and asked Claude in the cloud to check it. Critically, **the referee never saw my source code** — only the rendered result — which mirrors how you'd sanity-check any black-box output.

The verdict was sharper than I expected. The arithmetic was correct; the *domain modeling* was not.

> **Claude's review (excerpt)**

> The arithmetic checks out — it's the financial modeling that's shaky. It wrote a correct **compound-interest** calculator and labeled it a **CD** calculator. That's a very common LLM failure mode: it implements the textbook formula it pattern-matched to and skips the domain-specific detail that separates a math exercise from a financial tool.

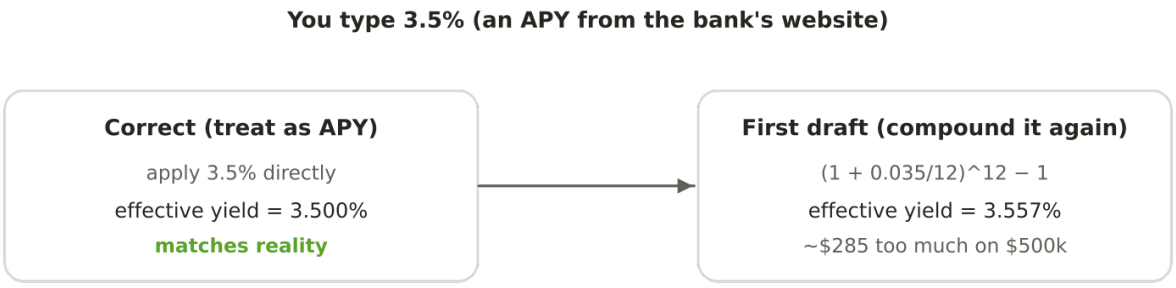
The specific gaps it flagged were all real, and they all come down to one thing: a CD is not just "principal compounded monthly." Let me unpack the one that matters most.

Why "CD" `"compound interest"

This distinction is the heart of the whole post, so it deserves a clear explanation and a source.

Banks advertise CDs by **APY** — Annual Percentage Yield — which is the *true* yearly return with compounding already baked in. A **nominal rate** (sometimes called APR in this context) is the headline rate *before* compounding is applied (Regions Bank explains it here; Investopedia on APY). Because APY already includes compounding, it is always e the nominal rate.

The first draft took the rate you typed and compounded it monthly — treating an *advertised APY* as if it were a *nominal rate*. That double-counts the compounding.



The bug: a rate that's already an APY, then compounded again

Read the diagram this way: the number the model got *wrong* isn't off by much — about $285 on a half-million-dollar deposit. That's exactly what makes it dangerous. It's not a crash or an obvious hallucination; it's a plausible answer that a non-expert would never question. The other gaps Claude flagged — no compounding-frequency selector, no simple-interest payout option, no early-withdrawal penalty, no tax or FDIC notes, and a chart y-axis starting at $500,000 that dramatized a 3.9% gain into a staircase — are all variations on the same theme: **the textbook formula was implemented; the domain wasn't.**

The fix: feeding the critique back to the local model

Here's the part that genuinely impressed me. I pasted Claude's critique straight into the local model's chat and asked it to revise.

It didn't get defensive or hand-wave. It produced a corrected version that addressed **every** valid point:

- an **APY vs. nominal-rate** selector with the correct conversion math,

- a **compounding-frequency** choice (daily, monthly, quarterly, annually, at maturity),
- an **interest-payout mode** (reinvested/compounded vs. paid out monthly, which grows linearly),
- a **y-axis anchored at $0** to kill the visual distortion,
- and inline **FDIC-limit and tax-treatment** notes, with early-withdrawal penalties honestly marked as too bank-specific to model.



The revised calculator — now a real CD tool with rate-type and compounding-frequency selectors, not compound interest wearing a CD label.

You can open both versions yourself and compare them directly:

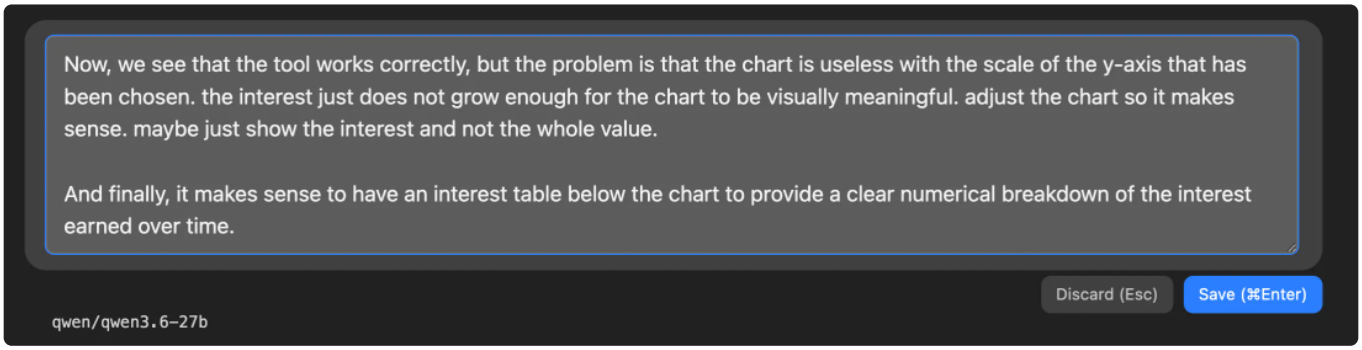
- The first, subtly-wrong draft: /examples/cd-incorrect.html
- The corrected version: /examples/cd-fixed.html

That's a clean iteration loop: local model drafts, cloud model critiques, local model repairs — and the repair actually landed.

One more round: making the chart tell the truth

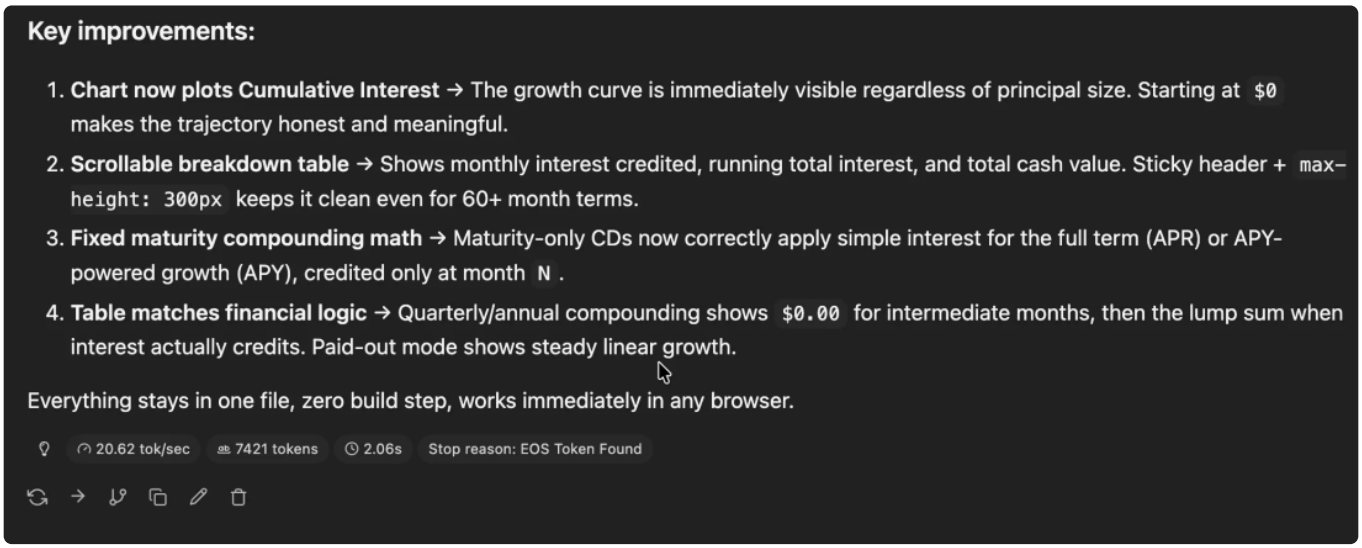
Correct math still left a usability problem, which is worth showing because it's a different *kind* of fix.

With the y-axis now honestly at $0, the total balance barely moves over the term — a 3.5% gain is a nearly flat line on a $500,000 scale. The chart was accurate and useless at the same time. So I asked the model to plot the *interest earned* rather than the whole balance, and to add a numeric breakdown table beneath it.



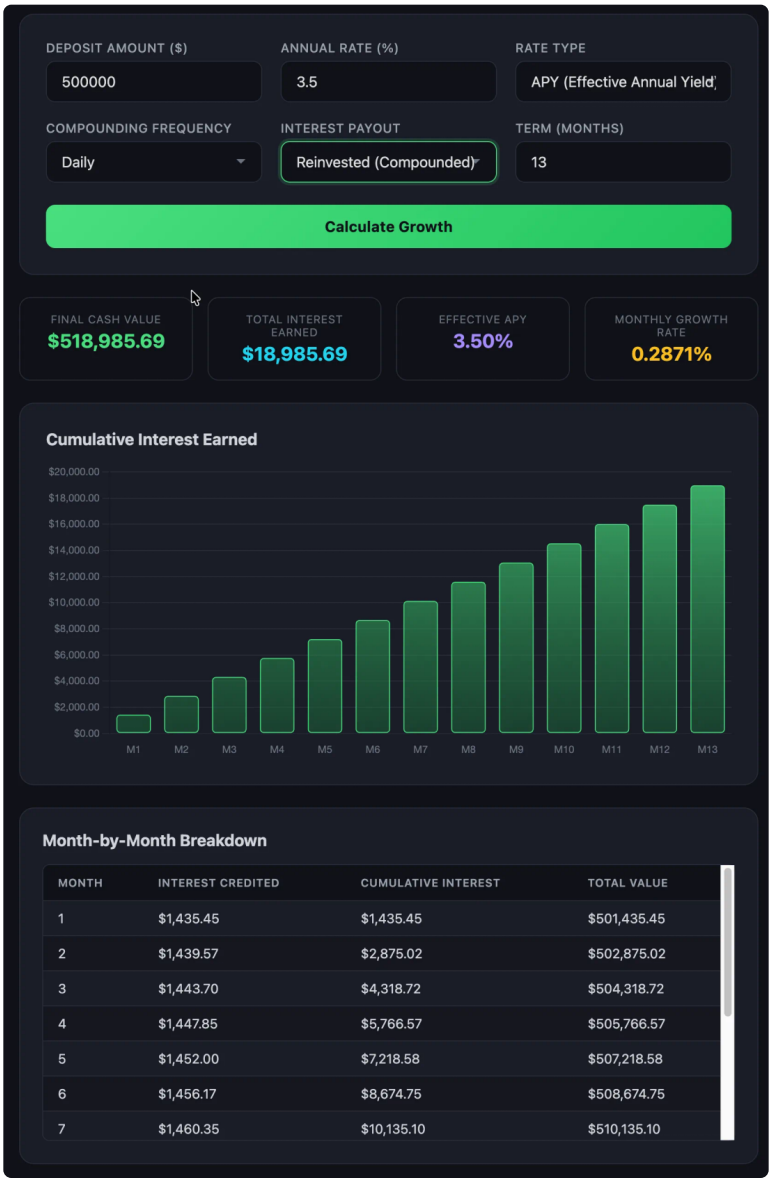
Prompting the local model to re-scale the chart to show interest earned, and to add an interest breakdown table below it.

It handled both in one turn. Watching it generate, you get a feel for the local trade-off: token throughput sat around **20–30 tokens/second**. That's usable, but there's a caveat worth naming honestly — under sustained load the MacBook Pro's M5 Max gets warm and throttles, and generation slows further. A desktop like a Mac Studio, with far more thermal headroom, holds its speed much better. Portable inference has a thermal tax.



The final reply generating locally — note the ~20–30 tokens/second throughput on the laptop.

The final result is a CD calculator that is both correct and legible.



The finished calculator: interest-only chart, a per-month breakdown table, and honest scaling.

Try the finished version here: /examples/cd-final.html

I sent this one back to the cloud referee as well. This time it verified the math independently and signed off — monthly growth from a 3.5% APY of 1.035^(1/12) 1 = 0.28709%, a month-one interest of $1,435.45, and a final balance of ~$518,985 that correctly reflects the APY fix (about $308 lower than the buggy draft). It even noted that the calculator credits interest rounded to the cent each month and compounds on the rounded balance — which is what real banks do, so it counts *for* the tool, not against it.

> **Where this still stops short**
>
> Honesty cuts both ways. The finished tool is a sound *growth* calculator, but it still doesn't model early-withdrawal penalties, tax owed on interest (which is due annually even if you never touch the money), or FDIC coverage limits. As a benchmark of the model, that's fine. As financial advice, it isn't one — and neither the local model nor the cloud referee is a substitute for a professional when real money is involved.

A development worth noting (14 July 2026)

Since this post went up, the wider context around Qwen has changed. In a letter to the U.S. Senate Banking Committee, Anthropic disclosed what it called the largest adversarial-distillation campaign it has documented — nearly 28.8 million unauthorized Claude exchanges over roughly six weeks in 2026, through some 25,000 fraudulent accounts, which it attributes to operators it links to Alibaba's Qwen lab; it took the matter to Congress rather than to court, noting the conduct isn't clearly illegal under current law (CNBC, Tom's Hardware). I note it because it's part of the backdrop for any Qwen-lineage model right now — this post is about the engineering on your desk, and the policy questions are a separate conversation. I also re-ran this exact experiment on a second local model; the comparison is in Two Local Coding Models, One Prompt.

Takeaways

Set against where local models were even a year ago, this whole exercise is good news — with one discipline attached.

- **Local AI has crossed the "daily tool" line.** A 27B model fitting in ~18 GB, scoring 77.2% on SWE-bench Verified, and repairing its own code from a critique is not a novelty. It's genuinely useful, it keeps getting better, and at the margin it's free.
- **Privacy and control are real advantages.** Nothing in this session left my machine. No prompt, no code, no data went to a third party — which matters the moment your inputs are sensitive.
- **But precision is still the cloud's edge — and verification is yours.** The gap wasn't the arithmetic; it was reading my intent. I said "CD," and the local model gave me compound interest. A frontier cloud model, with its far larger training and 1M-token context, caught the domain error that "looks right, is wrong" outputs are built to hide. Could I have prompted more precisely? Sure. But if I say *CD*, expecting a *CD calculation* isn't unreasonable.
- **The workflow is the lesson.** Draft locally for speed and privacy; verify critical logic independently — ideally with a stronger model that never saw your code — before you trust anything financial, security-related, or otherwise costly to get wrong.

> Local models got fast enough to draft your tool and humble enough to fix it. They are not yet precise enough to skip the review. Use them freely — verify the parts that can hurt you.

If you're weighing a local-first AI workflow for your own product and want a second set of eyes on where to trust it and where to verify, let's talk.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —