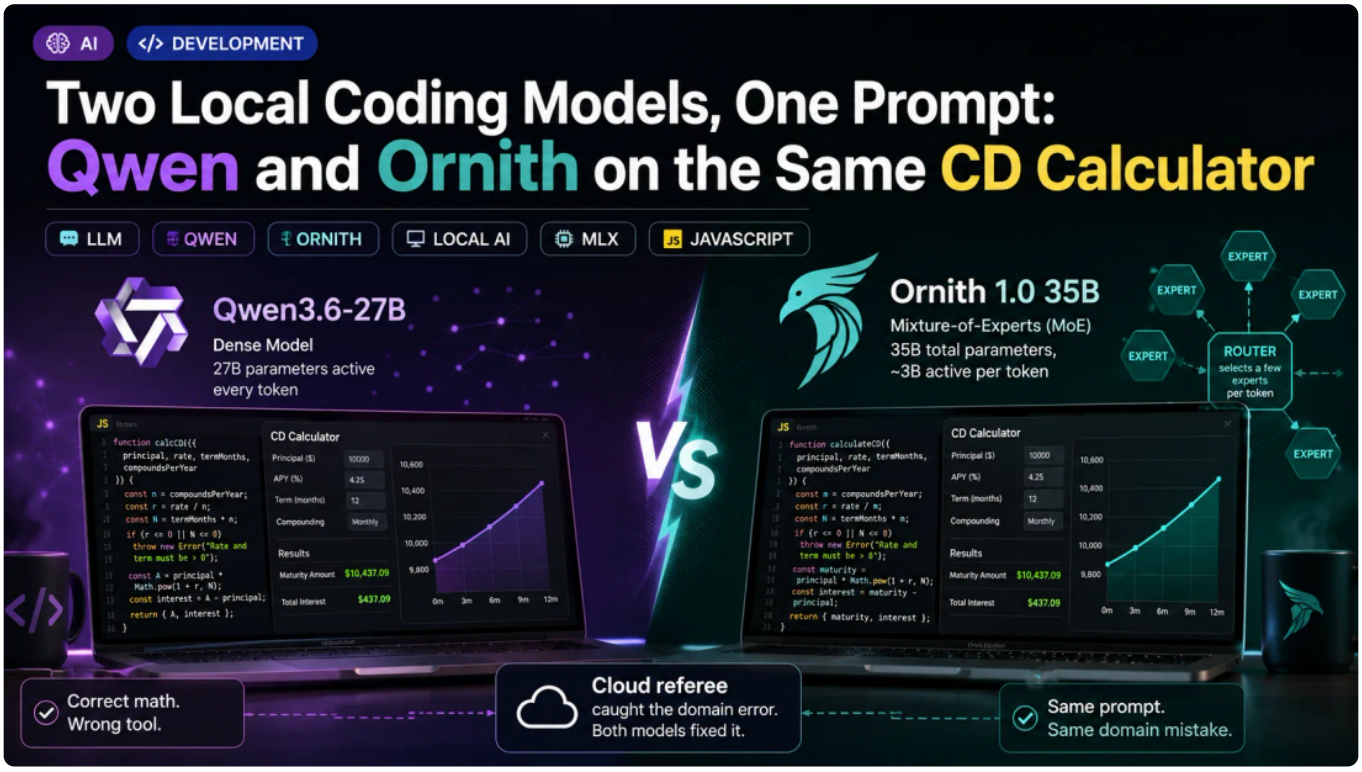
NEWS · JULY 15, 2026

Qwen vs. Ornith: Two Local AI Models, One CD Calculator

By Dr. Holger Flick



Last time, a 27-billion-parameter model on my laptop wrote a slick, charted CD calculator from a single sentence — correct math, wrong tool — and a cloud referee caught the domain error. This time I ran the *exact same* experiment with a completely different local model, Ornith 1.0 35B, to see whether the story holds when you swap the engine underneath it.

Ornith and Qwen are both models you can run entirely offline on a Mac, both genuinely good at code — and yet they're built on very different ideas. One is a dense model; the other is a self-improving mixture-of-experts. So the question isn't "which one wins." It's: *given the same casual prompt, where does each one shine, where does each one stumble, and which one fits which job?*

The short version: both nailed the arithmetic, both made the identical domain mistake, and both repaired themselves from the same critique. The differences that mattered were elsewhere — in polish, in detail, and in how they got there. Here's the whole run, side by side.

> **This is a companion to an earlier post**
> The Qwen half of this story — the model, the quantization math, the referee loop, and the full critique — lives in A Local AI Wrote a CD Calculator. This post assumes you've skimmed that one and focuses on

> the **comparison**. Both experiments use the same one-line prompt and the same cloud referee, so the only variable is the model.

Two philosophies of "runs on your desk"

Before the head-to-head, it's worth understanding *why* these two models feel different, because the architecture is the story.

Qwen3.6-27B — the model from my earlier post — is a **dense** model: all 27 billion parameters are active on every token. Ornith 1.0 35B, released on 25 June 2026 by DeepReinforce under the permissive MIT license, takes the other road. It's a **mixture-of-experts** (MoE) model: 35 billion parameters *total*, but only a small slice — about **3 billion** — is active per token, routed through a handful of specialized "expert" sub-networks instead of the whole model at once (model card).

That one design choice changes how the two models feel on the same laptop.

Read the diagram this way: Qwen keeps a smaller model in memory and runs *all* of it every step. Ornith keeps a *larger* model resident — you need more unified memory to hold it — but because only ~3B of it activates per token, it "feels" quicker than its 35B total suggests. Neither approach is better in the abstract; they trade memory for speed differently. On my machine I run the **4-bit** quantized build (mlx-community/Ornith-1.0-35B-4bit), which peaks around **20 GB** of unified memory — the full-precision bf16 build would need roughly **72 GB** — and I measured generation at a genuinely striking **100–120 tokens/second** — several times the 20–30 tok/s I measured for the dense Qwen build in the earlier post. That's exactly what the MoE design is *for*: because only ~3B of the 35B fires per token, it generates far faster than its total size suggests.

There's one more lineage detail that makes this comparison delightfully recursive: Ornith is **post-trained on a Qwen base** (reported here). So this isn't quite "Qwen vs. a rival" — it's closer to "Qwen vs. a specialized descendant of Qwen." Both teams deserve credit: Alibaba's Qwen for the foundation, DeepReinforce for the agentic-coding specialization layered on top.

> **What 'self-improving' actually means here**
> Ornith's model card calls it a "self-improving MoE." Concretely, that describes how it was *trained* — it "jointly optimizes search scaffolds and solution rollouts via Reinforcement Learning to discover superior code trajectories" (model card). It's a training method, not a model that rewrites itself on your laptop. Worth scoping, because the phrase invites a bigger claim than the evidence supports.

I ran it on Apple's MLX

The way I ran Ornith is itself part of the comparison, so it deserves a sentence.

I used the MLX build of Ornith — MLX is Apple's open-source machine-learning framework built specifically for Apple Silicon, exploiting the unified memory architecture directly rather than adapting CUDA-style operations to Metal. For MoE models on a Mac, that native path is a real advantage. The community `mlx-community` org publishes ready-to-run 4-, 5-, 6-, 8-bit and full-precision MLX conversions, so getting started is a one-line download. If MLX isn't your stack, Ollama and LM Studio run Ornith on the Metal backend too.

The benchmarks: close, and they disagree by task

Neither model is strictly "smarter." The published scores are close, and — importantly — they trade places depending on what you measure.

Here are the numbers, each from a primary or first-party source so you can check them yourself:

The takeaway from those bars: on **SWE-bench Verified** — a standard agentic-coding benchmark — Qwen3.6-27B posts **77.2%** (source) and Ornith 35B posts **75.6** (source); a hair apart. But on **Terminal-Bench 2.1**, which measures longer-horizon terminal work, Ornith reports **64.2** against **41.4** for its same-weight Qwen 3.5-35B peer, and DeepReinforce notes it even beats Alibaba's much larger Qwen 3.5-397B on that benchmark (source). Honest scoping: Ornith *trails* on SWE-Bench Pro in the same comparison. So — close, and it depends on the task. That's a healthy result for anyone shopping for a local model.

> **The wider landscape (June 2026) — context, not a verdict**
> Both models trace to Alibaba's Qwen lineage, which is very much in the news. 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. Anthropic took the matter to Congress rather than to court, noting the conduct isn't clearly illegal under current law (CNBC, Tom's Hardware). I mention it because it's part of the backdrop for *any* Qwen-lineage model right now, and readers deserve the full picture. This post is about the engineering on your desk; the policy questions are a separate conversation, and I'll leave those to the people having them. The tools themselves remain impressive pieces of work.

The task: identical prompt, identical trap

To keep the comparison fair, I gave Ornith the *exact* prompt from the Qwen run — same casual phrasing, same one-liner.

Here it is again, verbatim:

```
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
```

Ornith produced a single self-contained page — HTML, CSS, and JavaScript in one file, monthly compounding, a hand-rolled bar chart. Clean and immediate.

Ornith's first working version — good at first glance, and carrying the exact same problem Qwen had.

And here's the first genuinely interesting result of the comparison: **Ornith made the identical domain mistake Qwen did.** It built a mathematically correct *compound-interest* calculator and labeled it a *CD* calculator — the same "textbook formula, wrong domain" error. Both models pattern-matched "CD" to "compound the rate monthly" and skipped what actually makes a CD a CD.

You can open Ornith's untouched first attempt and see for yourself:

- Ornith's first draft: /examples/ornith/cd-first.html

Same referee, same verdict

Rather than re-derive the critique, I handed Ornith the *same* review Claude produced for the Qwen version — the one that pinpoints why "CD" ` "compound interest."

The core of that critique bears repeating, because it applies word-for-word to both models:

> **The referee's verdict (applies to both models)**
> The arithmetic checks out — it's the financial modeling that's shaky. Banks advertise CDs by **APY**,
>
> not a nominal rate; compounding a quoted APY monthly double-counts the compounding. There's no compounding-frequency selector, no simple-interest payout option, and the chart's y-axis starting at the principal dramatizes a small gain into a staircase. It implemented the formula it pattern-matched to and skipped the domain.

If you want the full explanation of the APY-vs-nominal bug — with sources and a diagram — it's in the companion post. What matters here is that a *second, differently-built* model fell into the *same* pit. That tells you the failure isn't a quirk of one model; it's a property of how these models reason about under-specified, domain-loaded prompts. Which is the whole reason to verify.

Where Ornith diverged: the repair

This is where the two runs stopped being identical — and where Ornith earned real respect.

In a single follow-up, I asked Ornith to switch to a real charting library *and* handed it the referee's full critique — the library swap and the domain fix in one prompt:

```
please use chart.js for the chart

The arithmetic checks out — it's the financial modeling that's shaky.
… [the full APY-vs-nominal / compounding-frequency / FDIC critique] …
everything except early withdrawal seems to be a valid criticism
```

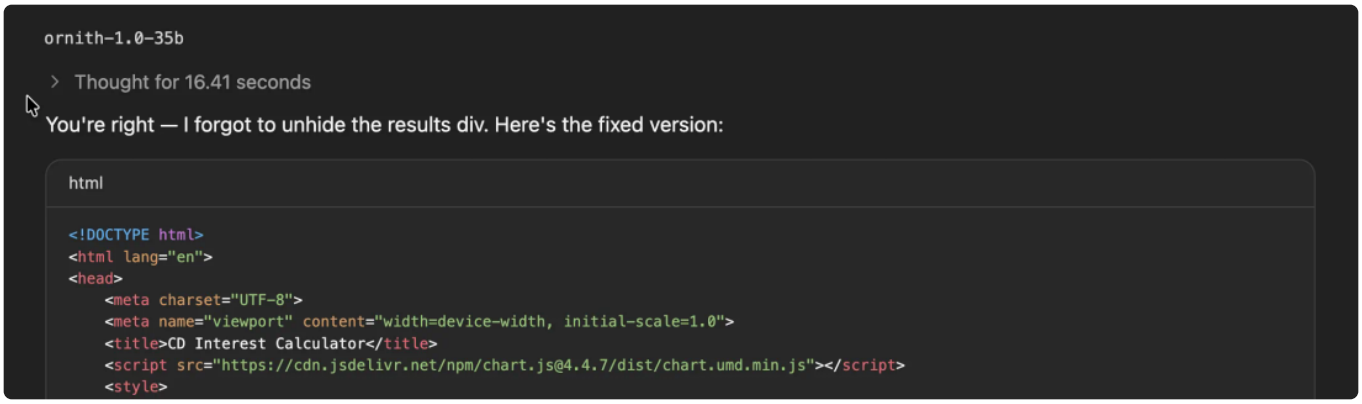
Ornith didn't get defensive; it wired up a properly configured Chart.js bar chart *and* owned the domain miss outright — the same "I know this library so you don't have to" leverage Qwen showed, plus an honest acknowledgment of the deeper error.

That combined revision, though, produced a page that rendered but did *nothing* on click — a real bug, not a subtle one. So I told it exactly that, in plain words:

```
this broke the functionality. nothing happens when I click Calculate Returns.
```

It diagnosed its own mistake immediately:

```
You're right — I forgot to unhide the results div. Here's the fixed version:
```

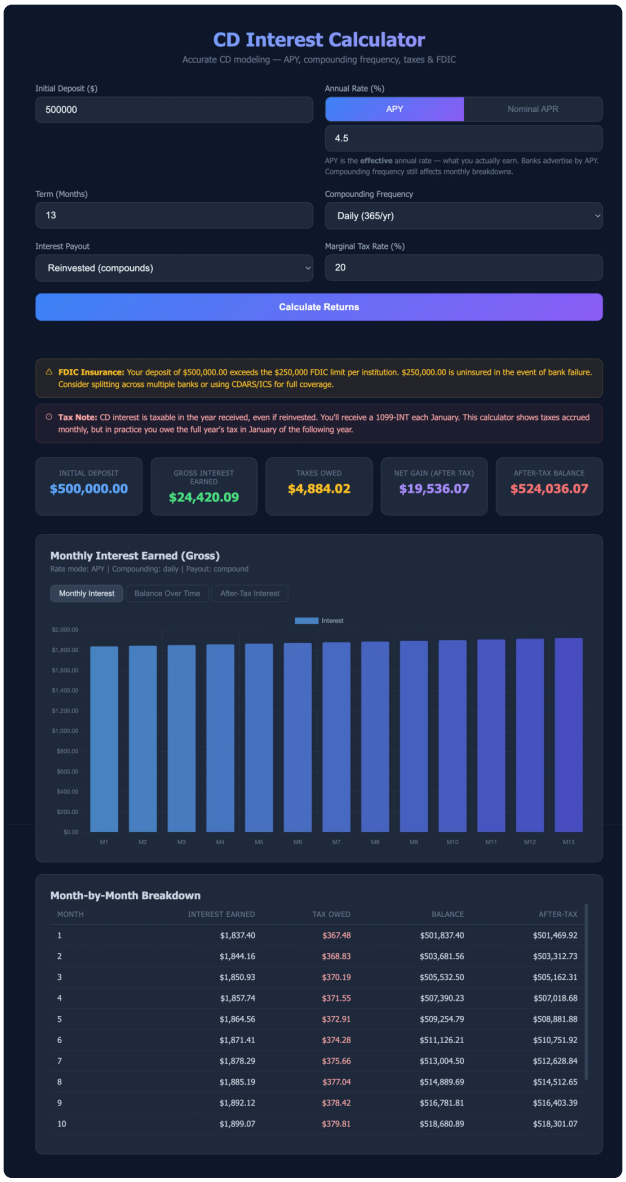


Ornith reading the critique and acknowledging the error with a fix ready.

And *then* it delivered — a version that went beyond the fix. Without me asking, Ornith added a full monthly-breakdown table, inline tax-treatment and FDIC notes, and warning messages about the tool's limits. Grepping the final file confirms it: an APY-vs-nominal selector, compounding logic, and dozens of references to tax and FDIC that simply weren't in the first draft.

Read that path honestly: Ornith took an *extra* stumble Qwen didn't — it shipped a broken build mid-way — but it recovered from a plain-language bug report and then over-delivered on detail. The intermediate and final versions are here so you can trace every step:

- Ornith after adding Chart.js and the correctness fixes: /examples/ornith/cd-second.html
- Ornith's final version: /examples/ornith/cd-third.html



Ornith's finished solution — a monthly-breakdown table it added unprompted, tax and FDIC warnings, and honest limits. The UI is a touch plainer than Qwen's, but it carries more detail and is more correct.

Qwen vs. Ornith: where each one fit

With both runs done from the same starting line, the differences are clear — and they're differences of *character*, not of a winner beating a loser.

Here's the fair, side-by-side read:

Interpreting that split: if you want the **prettiest result on the least memory**, dense Qwen is a lovely fit — and it edged Ornith on the pure SWE-bench score. If you want **more domain detail, faster tokens, and stronger long-horizon behavior**, and you have the unified memory to hold a 35B MoE, Ornith is compelling — it produced the *more correct and more complete* CD tool here, table and warnings included. Both are excellent; the right pick is the one that matches your machine and your job.

> **You're not limited to these two**
> The local-model landscape is rich, and naming the neighbors only builds trust. Meta's Llama, Mistral, and Google's Gemma all ship strong same-class builds, and the tooling is open: Ollama, llama.cpp, LM Studio, and Apple's MLX all load these models locally. Pick by fit — license, language strength, memory budget — not by hype.

Takeaways

Running the same experiment twice, on two very different engines, turned a one-off anecdote into something closer to a pattern.

- **The domain error is systemic, not model-specific.** Two independently built models — one dense, one MoE — made the *identical* "CD ` compound interest" mistake from the same casual prompt. That's the clearest signal in the whole exercise: under-specified, domain-loaded prompts trip up capable models regardless of architecture. Verification isn't optional insurance; it's the workflow.
- **Both models repaired themselves from a critique — one even overshot.** Fed the same review, both produced genuinely corrected tools. Ornith took an extra stumble (a broken intermediate build) but then added a breakdown table, tax notes, and FDIC warnings without being asked. Self-repair from plain feedback is now table stakes for local models, and that's remarkable.
- **Architecture is a fit decision, not a ranking.** Dense Qwen: smaller footprint, prettier UI, a shade higher on SWE-bench. MoE Ornith: bigger footprint, far faster tokens (I measured 100–120 tok/s), more domain detail, stronger long-horizon scores. Neither "wins." You choose by your hardware and your task.
- **The verification loop still holds.** Draft locally for speed and privacy; verify anything financial, security-related, or costly-to-get-wrong with an independent second opinion — ideally a model that never saw your code — before you trust it.

> Swap the local model and the lesson doesn't move: these tools are fast enough to draft your work and humble enough to fix it, but not yet precise enough to skip the review. The engine is a matter of fit. The verification is a matter of discipline.

If you're deciding which local model fits your product — and where to trust it versus verify it — 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 —