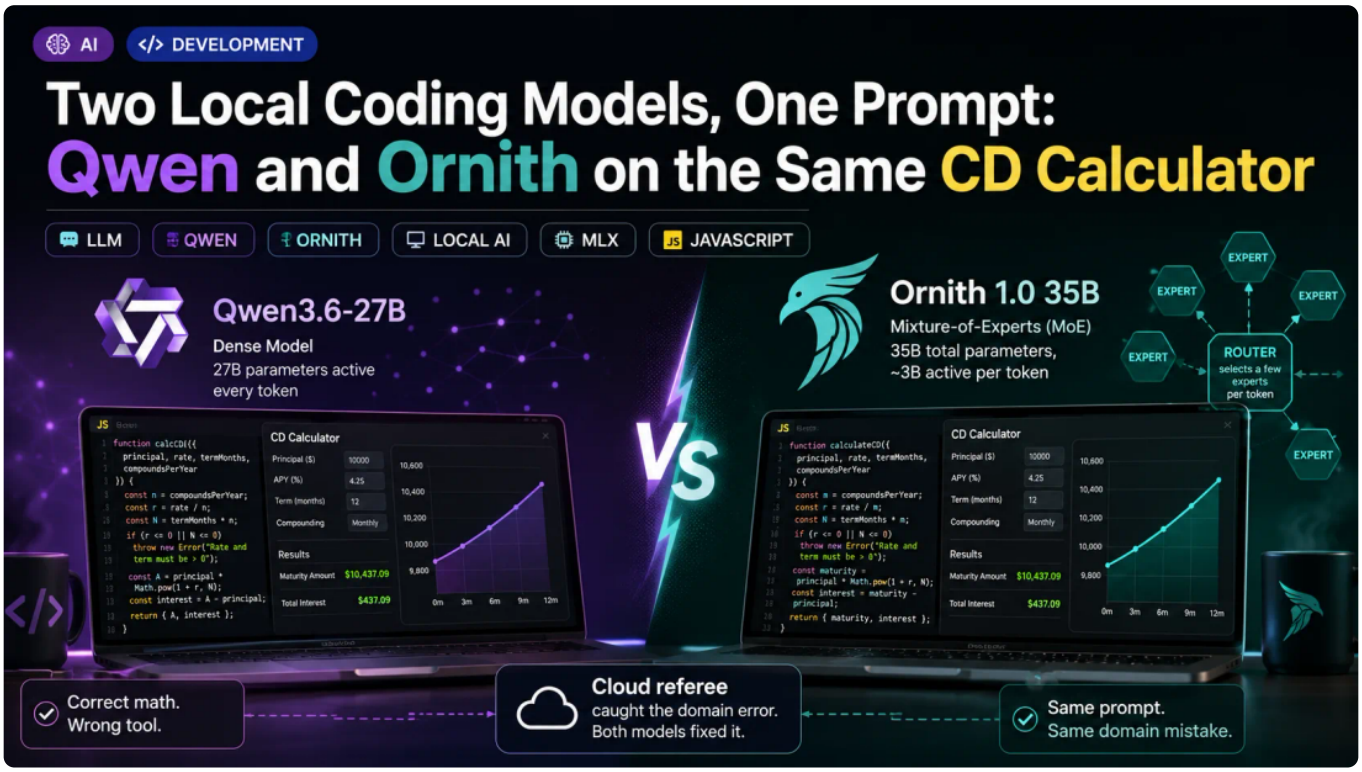
NEWS · JULY 15, 2026

Zwei lokale Coding-Modelle, ein Prompt: Qwen und Ornith am selben Festgeld-Rechner

By Dr. Holger Flick



Letztes Mal hat ein Modell mit 27 Milliarden Parametern auf meinem Laptop aus einem einzigen Satz einen schicken, bebilderten Festgeld-Rechner geschrieben — richtige Mathematik, falsches Werkzeug — und ein Cloud-Schiedsrichter fing den Fachfehler. Diesmal habe ich *genau dasselbe* Experiment mit einem völlig anderen lokalen Modell gemacht, Ornith 1.0 35B, um zu sehen, ob die Geschichte hält, wenn man den Motor darunter austauscht.

Ornith und Qwen sind beide Modelle, die man vollständig offline auf einem Mac laufen lassen kann, beide wirklich gut in Code — und doch bauen sie auf sehr unterschiedlichen Ideen auf. Das eine ist ein dichtes Modell; das andere ein selbstverbesserndes Mixture-of-Experts. Die Frage lautet also nicht „welches gewinnt". Sie lautet: *Wo glänzt jedes von beiden bei demselben beiläufigen Prompt, wo stolpert jedes, und welches passt zu welcher Aufgabe?*

Die Kurzfassung: Beide trafen die Arithmetik, beide machten denselben Fachfehler, und beide reparierten sich aus derselben Kritik heraus selbst. Die Unterschiede, auf die es ankam, lagen woanders — in der Politur, im Detail und darin, wie sie dorthin kamen. Hier ist der ganze Durchlauf, Seite an Seite.

> **Dies ist eine Ergänzung zu einem früheren Beitrag**
> Die Qwen-Hälfte dieser Geschichte — das Modell, die Quantisierungs-Mathematik, die Schiedsrichter-Schleife und die vollständige Kritik — steht in Eine lokale KI hat einen Festgeld-Rechner geschrieben. Dieser Beitrag setzt voraus, dass Sie jenen überflogen haben, und konzentriert sich auf den **Vergleich**. Beide Experimente nutzen denselben einzeiligen Prompt und denselben Cloud-Schiedsrichter, sodass die einzige Variable das Modell ist.

Zwei Philosophien von „läuft auf deinem Schreibtisch"

Vor dem direkten Vergleich lohnt es sich zu verstehen, *warum* sich diese beiden Modelle unterschiedlich anfühlen, denn die Architektur ist die Geschichte.

Qwen3.6-27B — das Modell aus meinem früheren Beitrag — ist ein **dichtes** Modell: Alle 27 Milliarden Parameter sind bei jedem Token aktiv. Ornith 1.0 35B, veröffentlicht am 25. Juni 2026 von DeepReinforce unter der freizügigen MIT-Lizenz, geht den anderen Weg. Es ist ein **Mixture-of-Experts**-Modell (MoE): 35 Milliarden Parameter *insgesamt*, aber nur ein kleiner Ausschnitt — etwa **3 Milliarden** — ist pro Token aktiv, geleitet durch eine Handvoll spezialisierter „Experten"-Teilnetze statt durch das ganze Modell auf einmal (Modellkarte).

Diese eine Designentscheidung ändert, wie sich die beiden Modelle auf demselben Laptop anfühlen.

Lesen Sie das Diagramm so: Qwen hält ein kleineres Modell im Speicher und führt bei jedem Schritt *das gesamte* aus. Ornith hält ein *größeres* Modell resident — man braucht mehr Unified Memory, um es zu halten — aber weil nur ~3B davon pro Token aktiviert werden, „fühlt" es sich schneller an, als seine 35B insgesamt vermuten lassen. Keiner der Ansätze ist im Abstrakten besser; sie tauschen Speicher und Geschwindigkeit unterschiedlich. Auf meiner Maschine nutze ich den **4-Bit**-quantisierten Build (mlx-community/Ornith-1.0-35B-4bit), der bei rund **20 GB** Unified Memory seinen Spitzenwert erreicht — der Build in voller bf16-Genauigkeit bräuchte etwa **72 GB** — und ich habe eine Generierung von wirklich beeindruckenden **100–120 Token/Sekunde** gemessen — ein Mehrfaches der 20–30 Token/s, die ich im früheren Beitrag für den dichten Qwen-Build gemessen habe. Genau dafür ist das MoE-Design da: Weil nur ~3B der 35B pro Token feuern, generiert es weit schneller, als seine Gesamtgröße vermuten lässt.

Es gibt noch ein Abstammungsdetail, das diesen Vergleich herrlich rekursiv macht: Ornith ist **auf einer Qwen-Basis nachtrainiert** (hier berichtet). Das ist also nicht ganz „Qwen vs. ein Rivale" — es ist eher „Qwen vs. ein spezialisierter Nachkomme von Qwen". Beide Teams verdienen Anerkennung: Alibabas Qwen für das Fundament, DeepReinforce für die darauf aufgesetzte Spezialisierung auf agentisches Programmieren.

!!! note "Was „selbstverbessernd" hier tatsächlich bedeutet" Orniths Modellkarte nennt es ein „selbstverbesserndes MoE". Konkret beschreibt das, wie es *trainiert* wurde — es „optimiert gemeinsam Such-Gerüste und Lösungs-Rollouts per Reinforcement Learning, um bessere Code-Trajektorien zu entdecken" (Modellkarte). Es ist eine Trainingsmethode, kein Modell, das sich auf Ihrem Laptop selbst umschreibt. Das ist eine wichtige Eingrenzung, denn der Begriff lädt zu einer größeren Behauptung ein, als die Belege hergeben.

Ich habe es auf Apples MLX laufen lassen

Wie ich Ornith laufen ließ, ist selbst Teil des Vergleichs und verdient daher einen Satz.

Ich habe den MLX-Build von Ornith verwendet — MLX ist Apples quelloffenes Machine-Learning-Framework, eigens für Apple Silicon gebaut, das die Unified-Memory-Architektur direkt ausnutzt, statt CUDA-artige Operationen an Metal anzupassen. Für MoE-Modelle auf einem Mac ist dieser native Pfad ein echter Vorteil. Die Community-Organisation `mlx-community` veröffentlicht direkt lauffähige MLX-Konvertierungen in 4, 5, 6, 8 Bit und voller Genauigkeit, sodass der Einstieg ein einzeiliger Download ist. Wenn MLX nicht Ihr Stack ist, laufen auch Ollama und LM Studio Ornith über das Metal-Backend.

Die Benchmarks: knapp, und sie widersprechen sich je nach Aufgabe

Keines der Modelle ist strikt „schlauer". Die veröffentlichten Werte liegen dicht beieinander und — wichtig — sie tauschen die Plätze, je nachdem, was man misst.

Hier sind die Zahlen, jede aus einer Primär- oder Erstquelle, damit Sie sie selbst prüfen können:

Die Erkenntnis aus diesen Balken: Auf **SWE-bench Verified** — einem gängigen Benchmark für agentisches Programmieren — erreicht Qwen3.6-27B **77,2 %** (Quelle) und Ornith 35B **75,6** (Quelle); ein Haar auseinander. Auf **Terminal-Bench 2.1** jedoch, das langfristigere Terminal-Arbeit misst, berichtet Ornith **64,2** gegen **41,4** für seinen gleichgewichtigen Qwen-3.5-35B-Peer, und DeepReinforce merkt an, dass es dabei sogar Alibabas viel größeres Qwen 3.5-397B schlägt (Quelle). Ehrliche Eingrenzung: Ornith *liegt zurück* auf SWE-Bench Pro im selben Vergleich. Also — knapp, und es hängt von der Aufgabe ab. Das ist ein gesundes Ergebnis für alle, die nach einem lokalen Modell suchen.

> **Die weitere Lage (Juni 2026) — Kontext, kein Urteil**
>
> Beide Modelle führen sich auf Alibabas Qwen-Abstammung zurück, die derzeit stark in den Nachrichten ist. In einem Brief an den Bankenausschuss des US-Senats legte Anthropic offen, was es als die größte je dokumentierte gegnerische Distillationskampagne bezeichnete — knapp 28,8 Millionen unautorisierte Claude-Austausche über rund sechs Wochen im Jahr 2026, über etwa 25.000 betrügerische Konten, die es Betreibern zuschreibt, die es mit Alibabas Qwen-Labor in Verbindung bringt. Anthropic brachte die Sache vor den Kongress statt vor Gericht und merkte an, dass das Verhalten nach geltendem Recht nicht eindeutig illegal sei (CNBC, Tom's Hardware). Ich erwähne es, weil es zum Hintergrund *jedes* Modells der Qwen-Abstammung gehört und die Leser das vollständige Bild verdienen. Dieser Beitrag handelt vom Engineering auf Ihrem Schreibtisch; die politischen Fragen sind ein eigenes Gespräch, und das überlasse ich denen, die es führen. Die Werkzeuge selbst bleiben beeindruckende Arbeit.

Die Aufgabe: identischer Prompt, identische Falle

Um den Vergleich fair zu halten, gab ich Ornith den *exakten* Prompt aus dem Qwen-Durchlauf — dieselbe beiläufige Formulierung, derselbe Einzeiler.

Hier ist er noch einmal, wortwörtlich:

```
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 erzeugte eine einzelne, in sich geschlossene Seite — HTML, CSS und JavaScript in einer Datei, monatliche Verzinsung, ein selbst gebautes Balkendiagramm. Sauber und unmittelbar.

Orniths erste funktionierende Version — auf den ersten Blick gut, und mit genau demselben Problem, das Qwen hatte.

Und hier ist das erste wirklich interessante Ergebnis des Vergleichs: **Ornith machte denselben Fachfehler wie Qwen.** Es baute einen mathematisch korrekten *Zinseszins*-Rechner und beschriftete ihn als *Festgeld*-Rechner — derselbe „Lehrbuchformel, falsche Domäne"-Fehler. Beide Modelle assoziierten „CD" mit „Zinssatz monatlich verzinsen" und übersprangen das, was ein Festgeld tatsächlich zu einem Festgeld macht.

Sie können Orniths unangetasteten ersten Versuch öffnen und selbst nachsehen:

- Orniths erster Entwurf: /examples/ornith/cd-first.html

Derselbe Schiedsrichter, dasselbe Urteil

Statt die Kritik neu herzuleiten, gab ich Ornith *dieselbe* Rezension, die Claude für die Qwen-Version erstellt hatte — jene, die genau bestimmt, warum „CD" ` „Zinseszins".

Der Kern dieser Kritik verdient eine Wiederholung, denn er gilt wortwörtlich für beide Modelle:

> **Das Urteil des Schiedsrichters (gilt für beide Modelle)**
> Die Arithmetik stimmt — es ist die Finanzmodellierung, die wacklig ist. Banken bewerben Festgeld mit dem **effektiven Jahreszins (APY)**, nicht mit einem nominalen Satz; einen ausgewiesenen APY monatlich zu verzinsen, zählt die Verzinsung doppelt. Es gibt keine Auswahl der Verzinsungshäufigkeit, keine Option für einfache Zinsen mit Auszahlung, und die bei der Anlagesumme beginnende y-Achse des Diagramms bläht einen kleinen Gewinn zu einer Treppe auf. Es implementierte die Formel, auf die es assoziierte, und übersprang die Domäne.

Wenn Sie die vollständige Erklärung des APY-vs.-Nominal-Fehlers wünschen — mit Quellen und einem Diagramm — steht sie im Begleitbeitrag. Worauf es hier ankommt: Ein *zweites, anders gebautes* Modell fiel in *dieselbe* Grube. Das sagt Ihnen, dass der Fehler keine Eigenart eines Modells ist; er ist eine Eigenschaft dessen, wie diese Modelle über unterspezifizierte, fachlich beladene Prompts nachdenken. Was der ganze Grund ist, zu prüfen.

Wo Ornith abwich: die Reparatur

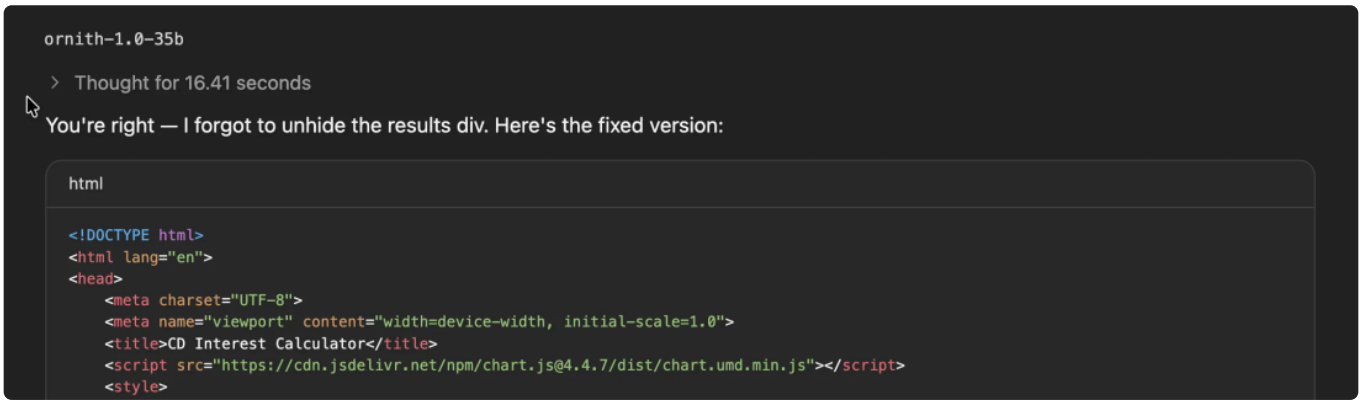
Hier hörten die beiden Durchläufe auf, identisch zu sein — und hier verdiente sich Ornith echten Respekt.

In einer einzigen Folgeanweisung bat ich Ornith, auf eine echte Diagrammbibliothek umzustellen, *und* übergab ihm die vollständige Kritik des Schiedsrichters — der Bibliothekswechsel und die fachliche Korrektur in einem Prompt:

```
please use chart.js for the chart

The arithmetic checks out — it's the financial modeling that's shaky.
… [die vollständige APY-vs.-Nominal- / Verzinsungshäufigkeit- / FDIC-Kritik] …
everything except early withdrawal seems to be a valid criticism
```

Ornith wurde nicht defensiv; es verdrahtete ein korrekt konfiguriertes Chart.js-Balkendiagramm *und* gestand den fachlichen Fehltritt geradeheraus ein — dieselbe „Ich kenne diese Bibliothek, damit du es nicht musst"-Hebelwirkung, die Qwen zeigte, plus ein ehrliches Eingeständnis des tieferliegenden Fehlers.



Ornith liest die Kritik und erkennt den Fachfehler an, statt seinen ersten Entwurf zu verteidigen.

Diese kombinierte Überarbeitung allerdings erzeugte eine Seite, die zwar gerendert wurde, aber beim Klick *nichts* tat — ein echter Fehler, kein subtiler. Also sagte ich ihm genau das, in klaren Worten:

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

Es diagnostizierte seinen eigenen Fehler sofort:

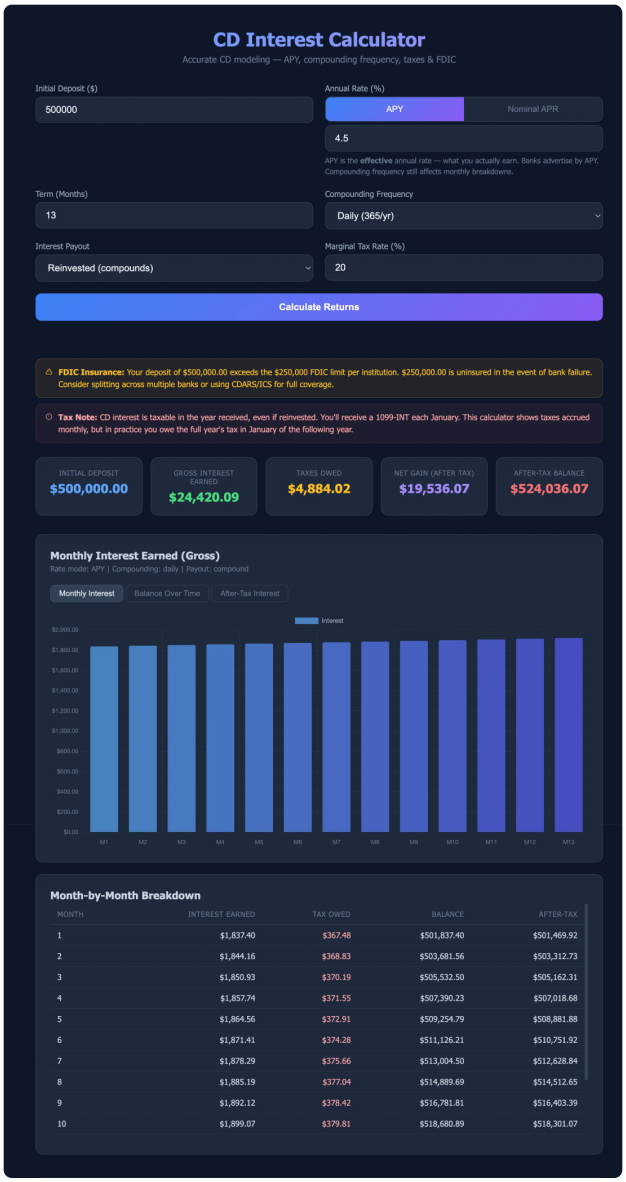
```
You're right — I forgot to unhide the results div. Here's the fixed version:
```

Und *dann* lieferte es — eine Version, die über die Reparatur hinausging. Ohne dass ich darum bat, fügte

Ornith eine vollständige monatliche Aufschlüsselungstabelle, eingebettete Hinweise zur Steuerbehandlung und zu FDIC-Grenzen sowie Warnmeldungen zu den Grenzen des Werkzeugs hinzu. Ein Blick in die finale Datei bestätigt es: eine APY-vs.-Nominal-Auswahl, Verzinsungslogik und dutzende Verweise auf Steuern und FDIC, die im ersten Entwurf schlicht nicht vorhanden waren.

Lesen Sie diesen Pfad ehrlich: Ornith machte einen Stolperer, den Qwen nicht machte — es lieferte auf halbem Weg einen defekten Build — aber es erholte sich aus einer alltagssprachlichen Fehlermeldung und lieferte dann beim Detail über. Die Zwischen- und Endversionen finden Sie hier, sodass Sie jeden Schritt nachvollziehen können:

- Ornith nach dem Einbau von Chart.js und den Korrekturen: /examples/ornith/cd-second.html
- Orniths finale Version: /examples/ornith/cd-third.html



Orniths fertige Lösung — eine monatliche Aufschlüsselungstabelle, die es unaufgefordert ergänzte, Steuer- und FDIC-Warnungen sowie ehrliche Grenzen. Die Oberfläche ist eine Spur schlichter als Qwens, trägt aber mehr Detail und ist korrekter.

Qwen vs. Ornith: wo jedes von beiden passte

Mit beiden Durchläufen von derselben Startlinie sind die Unterschiede klar — und es sind Unterschiede im *Charakter*, nicht ein Gewinner, der einen Verlierer schlägt.

Hier die faire Gegenüberstellung, Seite an Seite:

So ist der Split zu deuten: Wenn Sie das **hübscheste Ergebnis bei geringstem Speicher** wollen, ist das dichte Qwen eine schöne Passung — und es lag bei der reinen SWE-bench-Punktzahl vor Ornith. Wenn Sie **mehr Fachdetail, schnellere Token und stärkeres Langfristverhalten** wollen und den Unified Memory haben, um ein 35B-MoE zu halten, ist Ornith überzeugend — es lieferte hier das *korrektere und vollständigere* Festgeld-Werkzeug, Tabelle und Warnungen inklusive. Beide sind ausgezeichnet; die richtige Wahl ist die, die zu Ihrer Maschine und Ihrer Aufgabe passt.

> **Sie sind nicht auf diese zwei beschränkt**
>
> Die Landschaft der lokalen Modelle ist reich, und die Nachbarn zu nennen schafft nur Vertrauen. Metas Llama, Mistral und Googles Gemma bringen allesamt starke Builds derselben Klasse, und das Werkzeug ist offen: Ollama, llama.cpp, LM Studio und Apples MLX laden diese Modelle alle lokal. Wählen Sie nach Passung — Lizenz, Sprachstärke, Speicherbudget — nicht nach Hype.

Fazit

Dasselbe Experiment zweimal, auf zwei sehr unterschiedlichen Motoren, verwandelte eine einmalige Anekdote in etwas, das eher einem Muster gleicht.

- **Der Fachfehler ist systemisch, nicht modellspezifisch.** Zwei unabhängig gebaute Modelle — eines dicht, eines MoE — machten aus demselben beiläufigen Prompt den *identischen* „CD ` Zinseszins"-Fehler.

  Das ist das klarste Signal der ganzen Übung: Unterspezifizierte, fachlich beladene Prompts bringen fähige Modelle unabhängig von der Architektur ins Straucheln. Prüfung ist keine optionale Versicherung; sie ist der Arbeitsablauf.
- **Beide Modelle reparierten sich aus einer Kritik heraus selbst — eines schoss sogar über.** Mit derselben Rezension gefüttert, erzeugten beide echt korrigierte Werkzeuge. Ornith machte einen zusätzlichen Stolperer (einen defekten Zwischen-Build), fügte dann aber unaufgefordert eine Aufschlüsselungstabelle, Steuerhinweise und FDIC-Warnungen hinzu. Selbstreparatur aus alltagssprachlichem Feedback ist inzwischen Standard bei lokalen Modellen, und das ist bemerkenswert.
- **Architektur ist eine Frage der Passung, keine Rangfolge.** Dichtes Qwen: kleinerer Bedarf, hübschere Oberfläche, ein Stück höher bei SWE-bench. MoE-Ornith: größerer Bedarf, weit schnellere Token (gemessen 100–120 Token/s), mehr Fachdetail, stärkere Langfristwerte. Keines „gewinnt". Sie wählen nach Ihrer Hardware und Ihrer Aufgabe.
- **Die Prüfschleife hält weiterhin.** Entwerfen Sie lokal für Geschwindigkeit und Privatsphäre; prüfen Sie alles Finanzielle, Sicherheitsrelevante oder Teure-falsch-zu-machen mit einer unabhängigen zweiten Meinung — idealerweise einem Modell, das Ihren Code nie gesehen hat — bevor Sie ihm vertrauen.

> Tauschen Sie das lokale Modell aus, und die Lehre bewegt sich nicht: Diese Werkzeuge sind schnell genug, um Ihre Arbeit zu entwerfen, und bescheiden genug, um sie zu reparieren, aber noch nicht präzise genug, um die Prüfung zu überspringen. Der Motor ist eine Frage der Passung. Die Prüfung ist eine Frage der Disziplin.

Wenn Sie entscheiden, welches lokale Modell zu Ihrem Produkt passt — und wo Sie ihm vertrauen versus prüfen sollten — sprechen wir.

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 —