Skills in the Weights: The LoRA Answer to the Prompt Tax

August 31, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Four days ago I measured what compiling a skill into a prompt costs: 1,383.9 extra tokens on every call, forever, and a result that turned out to be noise. Macaron-V1 puts the skill in a LoRA adapter instead, so there is no prompt tax at all. Here is what it costs, and the question nobody has asked about it.

📚 **More on this topic:** Agent Skill Compilation Tested · LoRA Training on Consumer Hardware · Fine-Tuning LLMs on Consumer Hardware · Gemma 4 vs Qwen 3.6 on the 3090

Three of the guides on this site end the same way. You train a LoRA, you run `llama-export-lora`, you merge the adapter into the base, you ship one file. The consumer-hardware LoRA guide says it. The QLoRA guide says it. The MLX one says it.

That advice is correct, and I would give it again for one adapter. It is also exactly what the interesting architecture of 2026 exists to avoid.

The question underneath all of this

How do you install a skill into a model you did not train?

Not fine-tune it into a new model. Install it: hand an existing model a capability it did not have, and keep the model. There are three substrates for that, and only one of them has been measured.

You can put the skill in the **prompt**. You can put it in the **weights, offline**. Or you can put it in the **weights, at inference**, which is the exotic one.

I measured the first substrate four days ago. This is about the second.

Substrate one: the prompt, and what it actually costs

Last week I had a frontier model read my own production logs and write procedural skills for a local 27B doing intent classification. The skill goes in the system prompt. That is the whole mechanism: string concatenation.

Two numbers came out of it.

The first is the tax. Baseline inference on that task took 651.7 prompt tokens. With the skill appended, 2,035.6. **Every inference from then on pays 1,383.9 extra tokens**, forever, and the cost scales with the length of the skill. Compilation was a one-time $4.97 for twenty skills; divide that by the tax and you break even at 188 local calls.

The second number is worse. I compiled ten skills from byte-identical input and scored them separately. They ranged from 42.55% to 59.57% against a 48.94% baseline. Mean effect **+1.91 points against a compiler-noise floor of 4.65**, with a paired bootstrap of [−4.26, +8.30] straddling zero. The best one looked like a 10.6-point win and was a draw from a distribution.

So the prompt substrate is cheap to produce, immediate, readable, and taxes you on every call forever. Its artifact is about 5,800 characters of text. And at least on my task, what you get back is a sample, not a result.

Substrate two: the weights, compiled offline

Macaron-V1 came out of Mind Lab on 10 August 2026, 74 authors, and takes the other option. Freeze the base model. Train specialist LoRA adapters. Select one per user turn.

The flagship, Venti, is a 744B GLM-5.2 base with four adapters covering chat, agent, coding and generative UI. The smaller one, Tall, uses **Qwen3.6-35B-A3B** as its base and is explicitly billed for local deployment. Everything is MIT licensed and on Hugging Face, and people are downloading it — Tall has 744 downloads, Venti 541.

The appeal for our purposes is structural. **There is no prompt tax, by construction.** The skill lives in the adapter, not the context window, so a skilled inference costs exactly what an unskilled one costs. The 1,383.9 tokens I am paying forever on the intent classifier simply do not exist in this design.

You pay somewhere else.

The artifact is **7.55 GB**, not 5,800 characters. Producing it needs a training run on collected trajectories, not one API call. And using it needs a serving layer that can route to the right adapter, rather than a Python f-string.

That last one turns out to be less of a problem than I expected.

What Macaron claims, and what it admits

Before going further, the honest accounting, because the paper is unusually candid and the coverage of it is not.

The eye-catching numbers (58.3 on Macaron ChatBench, 64.0 on LivingBench, 87.8 on UI4A-Bench) are on **benchmarks Mind Lab built themselves**, judged by a privately deployed GLM-5.2, which is the same model family as Venti's base. They flag the risk in their own words: the common judge "may favor outputs from the same model family," and there is "no human agreement or cross-family judge calibration." No confidence intervals, no hypothesis tests.

On the external suite the picture is plainer. Venti "leads the listed TerminalBench 2.1 values and **trails the largest listed values on VitaBench, ClawGym, SWE-Verified, DeepSWE, and SWE Atlas QnA**." If you see "outperforms GLM on every benchmark" anywhere, that is not what the paper says.

They report two negatives outright. Their observe-before-commit REPL substrate scores **49.5% against 54.0% for plain function calling** on BFCL v4. And long-session character stability "degrades after multiple preference-drift events," which they describe as documented but not quantitatively measured.

And the motivation for the whole architecture is itself unmeasured. Cross-task interference is the reason to split capabilities into separate adapters, but: "The present release does not include the budget-matched single-LoRA comparison needed to quantify that interference, so we treat it as **a design motivation rather than an empirical finding** of this report."

Can you run it

Three findings, and the first one surprised me.

llama.cpp already ships the serving primitive

Mixture-of-LoRA needs one thing from a serving stack: hold several adapters and pick one per turn. `llama-server` has done this for a while and I have not seen anyone connect it to this architecture.

```
--lora a.gguf,b.gguf,c.gguf     # load several
--lora-init-without-apply        # load at scale 0, activate later
```

```
POST /completion
{ "prompt": "...", "lora": [{"id": 0, "scale": 1.0}] }
```

There is a `GET` / `POST /lora-adapters` endpoint for setting scales at runtime, and the per-request `lora` field takes adapter IDs with scales, defaulting everything unlisted to 0. **Per-request adapter selection is one-LoRA-per-turn.** That is the mechanism Macaron's MoL Proxy needs.

What is missing is only the routing policy, the small model call that reads the turn and picks a label. That lives in their Python proxy, and Mind Lab open-sourced it as Mixture-of-LoRA-Harness under MIT. The serving half is already on your machine.

One documented catch worth knowing: requests with different LoRA configurations are not batched together, which costs throughput on a busy server.

The converter probably cannot take Tall's adapters

Getting a PEFT adapter into GGUF means `convert_lora_to_gguf.py`. I read it looking for how it handles mixture-of-experts bases, since Tall is an MoE.

There is no expert branch. In 546 lines there is no handling for the stacked `ffn_gate_exps` / `ffn_up_exps` / `ffn_down_exps` tensors that llama.cpp builds for MoE models. The script delegates to the base architecture's converter with `super().modify_tensors(...)`, then asserts the result is a `LoraTorchTensor` — and that class raises `NotImplementedError` on exactly the reshape and row-size cases expert stacking would need.

Here is the irony. Venti's adapters, per the paper, "use rank 16, LoRA alpha 32, and the same attention/MLP target modules", which a converter can map. Tall's use "rank 64, LoRA alpha 128, and **expert-specific target parameters**."

The 744B model has the convertible adapters. The one billed for local deployment does not.

The arithmetic does not work on a 3090

I already have the base. `qwen3.6-35b-a3b-Q4_K_M.gguf` is 21.17 GB on disk, which is 20,187 MiB of weights. My 3090 has 24,576 MiB total and 20,065 MiB free with a desktop running, so the base alone needs a headless boot before it fits.

Each adapter is 7.55 GB in the released F16 form.

	3090 (24 GB)	3060 (12 GB)
Q4 base alone	fits headless, ~4 GB spare	does not fit
Base + one adapter	**does not fit**	no
Base + four adapters	not close	no

Routing requires the adapters to be resident, because the point is to switch between them mid-conversation. Base plus four is roughly 51 GB. You could quantize the adapters down, and you could merge one into the base and requantize — but merging is where this article started, and it costs you four separate 21 GB models instead of one base and four adapters. **The routing design and consumer VRAM are in direct tension**, and no amount of clever serving fixes that.

Substrate three: the weights, at inference

The exotic option deserves a mention because its cost is the one nobody quotes.

TTT-E2E, from Astera Institute, Stanford, Berkeley, UC San Diego and NVIDIA in December 2025, keeps training the model on your context as it reads, compressing what it sees into the weights. Constant inference latency regardless of context length, 2.7x faster than full attention at 128K on an H100.

The paper is honest about what that buys and what it costs. Table 2, RULER needle-in-a-haystack, 3B models at 128K context:

Method	Pass-key	Number	UUID
Full attention	**0.99**	**0.86**	**0.64**
Sliding window alone	0.07	0.05	0.05
TTT-E2E	0.06	0.05	0.03

Their own summary: full attention “dramatically outperforms the other methods, including ours,” because “the key mechanism in our method is compression, which leaves out seemingly irrelevant details, such as the target string.”

Two things worth adding that the paper does not say outright. **TTT-E2E is statistically indistinguishable from the plain sliding window it sits on**: 0.06/0.05/0.03 against 0.07/0.05/0.05. On retrieval, the test-time-training mechanism contributes approximately nothing over the 8K window underneath it. An outside researcher noticed the same thing from a reproduction and asked about it in issue #8 on 2 June. It has had no reply.

And the collapse is with **depth**, not in general. At 8K context TTT-E2E scores 1.00 / 0.99 / 0.77, and on the UUID task it beats full attention. It is a long-context failure, not a broken model.

For local work that distinction does not rescue it. RAG, document QA and codebase search are all needle-shaped, and the number that matters is 0.06.

The artifact situation settles the rest. The repo has **five commits, the last on 15 February 2026**, no LICENSE file, and one external contribution which was a README typo fix. Checkpoints went to a Requester-Pays Google Cloud bucket rather than Hugging Face, after a request from Hugging Face's own open-source team was closed without explanation. And there is no way to run it: asked for standalone inference code in April, the author replied that they have an eval mode, but "we don't have an entirely separated inference code at the moment."

Why you cannot combine them

The obvious thought is to take Macaron's machinery and get TTT's effect: swap adapters for skills, and also let the adapter absorb the conversation. It does not work, and the reason is precise.

Macaron's adapters hold competence, not memory. Swap one and the model gets better at coding; the conversation still arrives as tokens, which is why the paper has an entire section on KV cache reuse. There is no inner loop and no per-token update — parameter changes happen offline, "when trajectory evidence warrants a parameter update," after validation.

The deeper obstacle is the meta-learning. TTT-E2E's contribution is not the update rule, it is that W_0 was **trained to be receptive** to test-time updates. The paper ablates the alternative and calls it TTT-naive: applying test-time training to a conventionally-trained model. It "performs only slightly better than the toy baseline," and their toy baseline is a Transformer with no attention at all.

So anything TTT-shaped bolted onto a frozen Qwen base is TTT-naive by construction. Both architectures solve forgetting by freezing something, and both compress into weights. Neither can borrow from the other, because the freeze happens at a different time: Macaron freezes before deployment so the adapter is safe, TTT-E2E trains its base to be unfrozen so the inner loop works.

The question nobody has asked

I ran the prompt compiler ten times because the paper behind it reported one run per configuration, and I wanted to know the spread before trusting a number. The spread was 4.65 points, the effect was 1.91, and the win I was pleased with turned out to be the top of ten draws.

Nobody has done that on the weights side. **Nobody has trained the same adapter ten times on identical trajectories to see whether LoRA compilation is reproducible.**

I looked hard for it in Macaron's paper. Every `±` in the report is evaluation-sampling spread ("100 tasks per seed," five eval seeds on one already-trained adapter), and they qualify even that as not establishing equivalence. Every other mention of "seed" is a data-generation seed for building scenarios. There are no training seeds, no repeated runs, no re-training variance anywhere.

They say the gap themselves, in three places. Component attribution "remains open and is not resolved by any controlled experiment in this release." The evaluation "measures the configuration-search stage, **not repeated transfer across adapter generations**." And the report "does not provide a complete per-specialist training specification," with reproducing the checkpoints listed as a material limitation.

That is the same omission I found in the prompt substrate, in a different material. It matters more here, not less. When a compiled prompt turns out to be a draw from a distribution, you rerun it for 24 cents and take the average. When a 7.55 GB adapter is a draw from a distribution, you do not get to find out cheaply, and every benchmark number attached to it is a single sample nobody has bounded.

The architecture is sound and the serving path is already on your machine. What is missing is not engineering. It is the boring experiment where you do the same thing ten times and look at the spread.