MEMORANDUM
To: Wakai Stakeholders
From: Dirk Harms-Merbitz
Date: Feb 21, 2026
Re: Toasted — Running Inference on Apple Silicon

Toasted

The first memo described `toast` — AI as a Unix pipe. The second described the ecosystem of composable tools around it, showing how the user can create an iMessage bot in one line of code. The third talks about `jam`, the AI native shell. This memo provides a glimpse into the engineering behind the `toast` ecosystem.

TLDR: `toasted` is our from-scratch local inference daemon for Apple Silicon (first built in C against the MLX C API, then rewritten in C++ against MLX's C++ API). On a MacBook Pro with an M4 Max chip, running 4-bit quantized weights, it reads prompts and conversation history at nearly 400 tok/s and writes responses at around 100 tok/s, integrated with every tool in the `toast` ecosystem.

This memo is about the engineering journey from before to after:

Metric	Before	After
Prompt reading (17K tokens)	7 tok/s	394 tok/s
Time to first word (20K history, cache hit)	75 s	0.6 s
Response writing (short history)	23 tok/s	107 tok/s
Response writing (28K-token history)	~23 tok/s	~80 tok/s
Response writing (68K-token history)	N/A	~60 tok/s
Memory needed (11-turn chat)	11 GB	1 GB

(For reference: a token is a short chunk of text; 100 tok/s is roughly 70–80 English words per second.)

Tool complexity

Toast routes inference through cloud providers. Anthropic, OpenAI, Groq — text goes out over the network, tokens come back. Every request has a cost, a round trip, and data leaving the machine.

For local inference, we support tools like Ollama and LM Studio. Both are good products. Both sit on the same foundation: a Python runtime loads the model into GPU memory, compiles Metal shaders, and generates tokens in a loop. Ollama wraps llama.cpp. LM Studio wraps MLX.

Managing these tools is a lot to ask of a user that just wants to `toast`. Python has a half second startup time and a fragile ecosystem with env issues. Importing frameworks means having to learn yet another thing. Instantiating the model, wait, which model? Maybe compiling shaders. Default context windows are small, suitable for chatting but not for work. For a terminal-literate non-developer who just wants local inference, this complexity becomes a wall.

We want `toast` to be a reflex, like grep or curl. What if we had a local inference server, `toasted`, with a single file AI model, not a hidden directory with mystery content?

The Model

We used Qwen3-Coder-Next because it is an unusual model: 4-bit quantized, 42.7 GB on disk, and designed for efficient inference. It should run on any Mac with 64 GB or more of RAM.

512 experts, 8 active. Each token routes through a small router network that selects the 8 most relevant feed-forward experts out of 512. The other 504 experts sit idle—their weights are not touched on that step. The model *knows* as much as if all 512 ran (training saw all of them), but inference only *pays* for 8.

36 DeltaNet layers, 12 attention layers. Most LLMs are 100% transformer attention—every token attends to every other token, with cost growing with context length. Qwen3-Coder-Next uses DeltaNet, a linear recurrent architecture, for 75% of its layers. DeltaNet maintains a fixed-size state matrix (constant with respect to context length). Only 12 layers use traditional attention with KV caches that grow with context.

What can we expect? ~5 GB of weight/state data read per decode step. At ~546 GB/s unified-memory bandwidth, that implies a bandwidth floor of ~9.1ms per token, or ~110 tok/s. This is a 42 GB model running at speeds typically associated with a 7B dense model. In practice, overheads like kernel dispatch and cache updates, and performance degradation at long context lengths, tend to push measured throughput slightly below this theoretical floor. But we can try to get as close as possible.

`toasted` in C (MLX C API)

`toasted` is our from-scratch implementation of transformer inference in C. We wrote the attention mechanism, the feed-forward layers, the rotary position embeddings, the RMS normalization, the top-p sampling — all directly against Apple's MLX C API. It reads the usual model directory (safetensors weights, config, tokenizer) and runs inference natively. No Python anywhere.

After a bit of iteration it worked and generated tokens correctly at 21 tokens per second on a MacBook Pro with M4 Max and 128GB of RAM.

Wait, our C code is slower than Apple's MLX? Turns out that MLX C bindings are not fully supported and we had to call each operation individually: multiply here, add here, softmax here. The GPU spent more time waiting for instructions than doing math.

Pre-Computed Graphs

What can we do at compile time? Maybe we do not need to *rebuild* the computation graph at runtime. Instead trace it once using Apple's MLX software and serialize it to disk. When `toasted` starts it loads the pre-computed graph once. Same fused operations. Same optimized dispatch. Same speed.

Using MLX we run the model's forward pass, and instead of executing it, capture the entire computation graph — every matrix multiply, every attention head, every expert routing decision — and write it as a serialized function to disk.

Because each trace is shape-specific we have to *bake* each model that we want to support. But that happens only once and we do it, not our users.

$$f(\text{token}, \text{cache}_1, \text{cache}_2, \ldots, \text{cache}_{96}) \rightarrow (\text{logits}, \text{cache}'_1, \text{cache}'_2, \ldots, \text{cache}'_{96})$$

To support a context window of n tokens, we need n traces — one for each possible position. For the full 262,144-token context, that is 262,144 individual computation graphs, plus 2,048 chunk-prefill graphs for processing prompts in batches of 128.

Sounds like a lot but it is doable. The model weights — which dominate the file size at 42 GB — are stored once and referenced by every trace. Each additional trace adds only its graph topology, which is negligible. The full export for this model is approximately 45 GB regardless of how many traces it contains.

With pre-computed graphs we get around **80 tokens per second.** Parity with Apple's MLX!

Our `bake` workaround got us to ~80 tok/s, but it required a day long export step. We wanted MLX-level performance *without* baking.

`toasted` rewritten in C++ (MLX C++ API)

Better might be possible so we decided to rewrite `toasted` in C++. Hopefully the better MLX bindings would get us more performance. Our first test showed that prefill performance had regressed! While prefill dropped from 40 tok/s to 7 tok/s, generation was acceptable now, at over 90 tok/s.

GPU sync barriers. The code called `mx::eval(hidden)` every 4 layers during prefill—12 synchronization points that flushed the GPU pipeline each time. For a 14-token prompt through 48 layers, the GPU spent more time waiting than computing.

Cross-stream interference. Prefill ran on a separate GPU stream, but the custom DeltaNet Metal kernel didn't explicitly inherit it. This caused implicit synchronization between streams—the worst kind of performance bug because it's invisible.

The fix was two lines: skip intermediate evals for short sequences, drop the separate prefill stream. Prefill jumped from 7 tok/s back to 40.

Stuck at 20 tok/s

Even the C++ implementation was stuck at 21 tok/s. We tried a number of things but no matter what we did, the number of tokens per second stayed in the 20s. We looked into Apple's source code to see what Apple's implementation was doing. The numbers did not move. Profiling showed our MoE layer was still 2.4× slower in C++ — 0.90ms per layer versus 0.29ms — despite calling the same `gather_qmm` under the hood.

We suspected an fp16 vs fp32 type leak. That was it. The MoE gate's quantized weights use `bfloat16` scales and biases. When MLX dequantizes them, the output promotes to `float32`. Python's `softmax(precise=True)` computes in float32 but casts the result back to `float16` before feeding it downstream. Our C++ code didn't cast back. The `float32` gate output propagated through expert selection, into `gather_qmm`, through the expert projections, through the residual add — every subsequent operation inherited the wider and slower type.

The effect was invisible in the code and invisible in the output. The model produced correct tokens. But every tensor that touched the MoE path doubled in size, doubling the memory bandwidth consumed per token. On a bandwidth-bound workload, that is fatal.

Making sure everything stayed fp16 got us generate speed in the 50 tokens per second range.

Prompt (and context) reading: from 7 to 40 to 394 tokens per second

Prompt reading (often called *prefill*) is the model processing its prompt and context before it can write the first word. After the disappointing 7 tok/s, our more reasonable 40 tok/s became the baseline. However, for a

17,000-token conversation, that's 7 minutes of staring at a blank screen before the first word appears.

The bottleneck was that prefill processed the entire prompt as one massive batch through `model_forward`. The 12 attention layers used `scaled_dot_product_attention` with a causal mask over all 17K tokens simultaneously—a huge matrix operation.

We restructured prefill into 32-token chunks. Each chunk processes through all 48 layers, updates the cache, and moves on. The attention layers only see their chunk (plus cached KV from previous chunks), keeping the matrix operations small.

Result: 394 tok/s prefill. The same 17K prompt now prefills in ~44 seconds instead of 7 minutes.

The Road to generate 100 tok/s

At ~93 tok/s at short context (and ~82 tok/s at 12K context), we were leaving performance on the table. The gap analysis:

- 9.1ms: Reading ~5GB gives us the bandwidth floor, can't optimize
- 0.5ms: GPU kernel dispatch overhead
- 1.0ms: CPU-side graph construction (building the MLX computation graph each token)
- 0.4ms: Miscellaneous (KV cache ops, synchronization)

Compiled step functions. MLX supports `mx::compile()`—trace a computation graph once, cache the compiled version, replay it for subsequent calls. We wrapped the 36 DeltaNet layers and 12 attention projection stages into compiled functions, eliminating the per-token graph construction overhead.

Result: 89.6 tok/s at 12K context (up from 81.7), with a flatter degradation curve as context grows.

Speculative Decoding

How about speculative decoding? We tried two approaches—n-gram prompt lookup and self-speculative early exit. Both failed catastrophically. The n-gram approach got 11–14% acceptance rates (you need >70% to break even). Self-speculative got 0%—early layers of this hybrid architecture produce representations too different from the final output.

All speculative code was reverted. Sometimes the best optimization is knowing when to stop.

Remembering what we read

This was the big one. Every request from toast included the *entire* conversation history. At 20K tokens, that meant 60+ seconds of redundant prefill—re-processing identical messages that hadn't changed since the last turn. The user types "thanks" and waits over a minute.

The insight: on each turn, only the last message is new. Everything before it was already processed on the previous turn. If we save the model's internal state (KV caches for 12 attention layers, recurrent states for 36 DeltaNet layers), we can restore it and only prefill the new message.

The implementation:

1. Hash `messages[0..n-2]` (everything except the last message) using FNV-1a

2. Check the hash against an LRU cache of saved sessions
3. On hit: restore cached state, tokenize only the last message (~30–80 tokens), prefill just that
4. On miss: full tokenize, full prefill
5. After generation: save the new state with a hash that includes the generated response

The save hash is constructed so the *next* request's lookup hash will match it—because the next request's `messages[0..n-2]` will include this response as the second-to-last message.

Result: 75 seconds → 600 milliseconds. A 125× improvement in time-to-first-token.

The Memory Problem

The first version of the cache stored every session state. In a single chat, each turn created a new entry. After 8 turns, we had 8 snapshots of essentially the same conversation at different points—~11GB of dead KV cache data that would never be hit again.

The fix was parent eviction: when saving a new session that was restored from an existing one, evict the parent. A single chat now holds exactly one session in cache. Two independent conversations hold two. Memory tracks active conversations, not history.

Becoming a Daemon

A CLI tool that processes one prompt isn't useful for chat. Toast needed a persistent daemon—something that loads the 42.7GB model once, listens on a Unix socket, and handles requests.

This session was pure engineering: Unix socket server, JSON wire protocol, signal handling, daemonization with `fork()`/`setsid()`, debug mode, chat template support for multi-turn conversations. The kind of work that's straightforward but tedious. AI wrote the entire daemon infrastructure in one pass.

One subtle bug emerged: `fork()` after initializing Metal breaks Apple's XPC connection to the GPU. The fix was loading the model *before* forking. Small detail, hours of debugging compressed into a suggestion.

The Model Reviews Its Own Code

Once we had a working daemon, we asked the model (running on `toasted`) to review the engine's code and suggest optimizations.

Paged KV Cache (vLLM-style): It proposed replacing our `concatenate`-based KV growth with a block allocator. Sounds impressive. In practice, `concatenate` runs once every 256 tokens, copies 12MB at 546 GB/s, and costs 0.02ms. The actual bottleneck—SDPA reading the entire KV cache—costs 1.5ms per token. The optimization would be 20,000× less impactful than the problem it claims to solve.

Cache the last hidden state from prefill: It claimed `gen_step` redundantly recomputes the prefill's last hidden state. It doesn't. Prefill produces logits → sample token T_1. `gen_step(`T_1`)` embeds a *different token* and runs it through all 48 layers to update the cache. No redundancy.

Enable MetalFX: That's Apple's game graphics upscaler. Has nothing to do with compute shaders.

Pin threads to performance cores: The CPU thread just submits work to the GPU and reads back one integer. It's not the bottleneck.

After several pages of wrong suggestions, and really trying hard to find something to improve, the model eventually reasoned itself to the correct conclusion:

> "Your code is optimal. The only path forward is a faster Mac."

It knew *about* inference optimization—it had seen thousands of blog posts and papers during training. But it had zero visibility into its own architecture, the runtime, or the hardware. It pattern-matched on code structure rather than tracing execution.

The system prompt said "be brief, as if text messaging." The model reviewing its own engine responded with excited 2,000+ token essays and emoji section headers. This is perhaps the most relatable LLM behavior of all.

Where We Landed

Metric	Before	After
Prompt reading (17K tokens)	7 tok/s	394 tok/s
Time to first word (20K history, cache hit)	75 s	0.6 s
Response writing (short history)	23 tok/s	107 tok/s
Response writing (28K-token history)	~23 tok/s	~80 tok/s
Response writing (68K-token history)	N/A	~60 tok/s
Memory needed (8-turn chat)	11 GB	1 session

The final `toasted.cpp` is ~1,800 lines of C++ with:

- Full hybrid DeltaNet/attention inference with a hand-tuned Metal kernel
- MoE routing through 512 experts via `gather_qmm`
- Chunked batch prefill
- Compiled step functions for all 48 layers
- Session cache with LRU, parent eviction, and FNV-1a hashing
- Unix socket daemon with JSON protocol
- Temperature sampling, multi-turn chat templates, audit logging

What We Learned

The biggest wins aren't where you expect. Compiled step functions—the "obvious" optimization—gave 10%. The session cache—an architectural change to avoid redundant work—gave 12,500%. The best optimization was not making the hot loop faster, but not entering it at all.

LLMs are better at writing code than reviewing it. AI wrote the DeltaNet kernel, the MoE routing, the session cache, all correctly on first or second attempt. But when asked to *optimize* existing code, it pattern-matched against training data rather than reasoning about the specific system. Writing is synthesis; optimization is analysis. They're different skills.

Unified memory changes the optimization landscape. Most inference optimization literature targets discrete GPUs with separate CPU/GPU memory. On Apple Silicon, there's no PCIe transfer, no memory

copies, no staging buffers. Half the standard playbook doesn't apply. You need to reason from first principles about bandwidth and compute, not cargo-cult techniques from GPU server deployments.

The model that can't taste the soup. Qwen3-Coder-Next, running on the very engine we built, couldn't identify a single valid optimization for that engine. It had read every cookbook but couldn't tell you if the soup needed salt. There's a lesson in there about the difference between knowledge and understanding—one that feels important as we build systems that are very good at the former.

A full 256K-context *bake* for `toasted` in C takes about 18 hours. With `toasted` in C++ we no longer need that process.

How it works

`toasted` loads the model once. For 45 GB this takes about sixty seconds. It then listens on a Unix domain socket, accepts requests using the same wire protocol `toast` and `toasted` already speak, generates a response, and sends it back. The model stays hot in GPU memory between requests. Metal shaders stay compiled. Cache initialization arrays stay loaded. Unlike naive Python based command line inference tools that may load the model every time.

`toasted`, with an `e`, is listed as a provider in the `toastd` router. It opens a Unix socket to toasted, forwards the request, and relays the response. No HTTP parsing, no headers, no serialization overhead. The integration was ten lines of code.

When `toasted` is running, `toast` will automatically default to local inference. Other models can be used as well but have to be specified.

The result:

```
toast "explain quicksort"
```

A 30-billion-parameter model running on local hardware with the same interface as every cloud provider, delivering around 100 tokens per second. The pipe does not care where the intelligence comes from.

`toasted` gives `toast` users a local, on their laptop, inference path with zero marginal cost, zero latency, and zero data exposure. Air-gapped environments, regulated industries, and security-conscious teams that cannot send code to cloud APIs now have a fully local option that works identically to the cloud providers — same flag, same pipe, same output. Without the additional complexity of local LLM tools.

Ideas being considered

Speculative decoding — a smaller draft model proposes, verified by the large model. Expected 2–3× throughput.

Unix for AI Ops. Now with a local brain.

https://linuxtoaster.com