Locai Link vs Ollama — Gemma 4 benchmark (macOS, Apple M2, 16 GB)

Same Gemma 4 (Q4_K_M) weights, both runtimes on an Apple M2 / 16 GB host. Reasoning off on both for an apples-to-apples comparison.

- **Link wins prefill and TTFT by ~2×.** Conversation TTFT 2.3× shorter; document-analysis TTFT 1.8× shorter; prefill 2× faster in both scenarios.
- **Decode is tied at ~30-40 tok/s** — both backends bottleneck on the M2's unified-memory bandwidth, which is much lower than a discrete NVIDIA GPU.
- **End-to-end latency is essentially tied** for both scenarios; the prefill advantage is small in absolute terms because the M2's compute is the limit, not the prompt-processing setup.
- **Memory cost:** Link uses about 3 GB more RAM than Ollama (6.5 GB vs 3.3 GB on conversation, 6.8 GB vs 3.4 GB on document analysis).

Backends used: Link and Ollama both running with Apple's Metal-accelerated llama.cpp builds. On Apple Silicon there is no separate VRAM — the GPU uses the unified memory pool, so RSS captures the total memory footprint.

What we tested

Model	Gemma 4 (E2B-it, Q4_K_M quantisation, 2.9 GB GGUF, 4.6 B params) — **identical bytes** loaded by both backends
Hardware	Apple Mac, M2 SoC, 16 GB unified memory
GPU usage	Metal acceleration on both backends (Apple Silicon default). Unified memory — no separate VRAM column
Reasoning mode	**Off on both** — Link via `LLAMA_ARG_REASONING=off`, Ollama via `think: false` on `/api/chat`
Sampling	temperature=0, seed=42, top_p=1, streaming — same on both
Scenarios	**Conversation** (short prompt → ~300-token reply, 20 runs) and **Document analysis** (~9.5 k-token excerpt from *From the Earth to the Moon* → reply up to 500 tokens, 10 runs)
Warm-up	3 throwaway requests per scenario per backend before measurement
Errors	0 / 60 measured requests

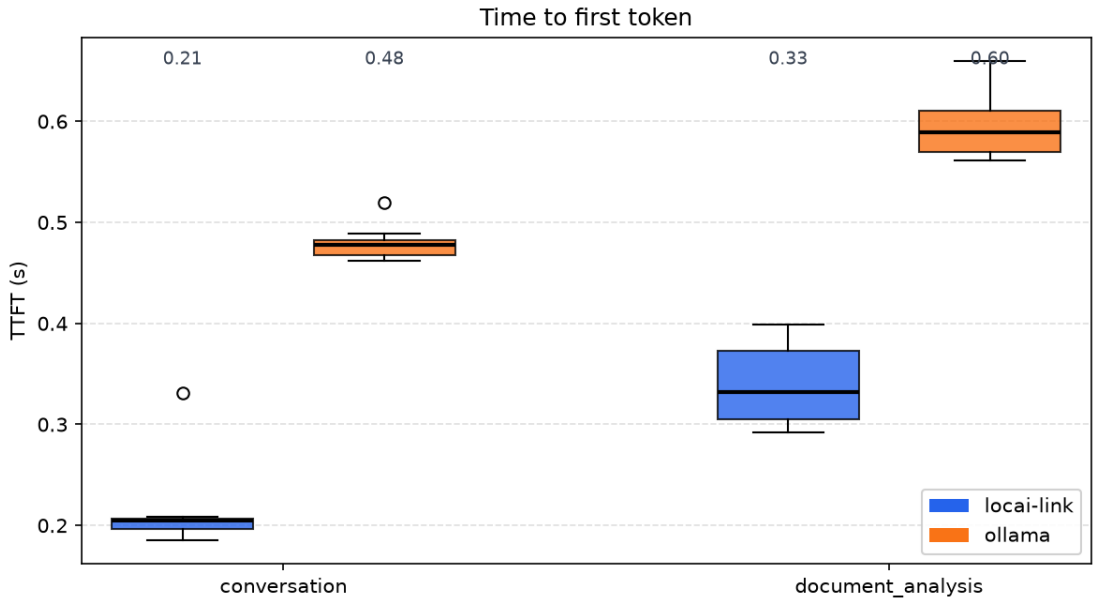
How to read the numbers

Metric	What it means	Why it matters
TTFT (Time to First Token, ms)	Latency from request sent to first word of the reply	The “snappiness” of a chat
Prefill tok/s	How fast the model reads your input, in tokens/sec	Dominates document-analysis latency
Decode tok/s	How fast the model produces its reply, in tokens/sec	The streaming speed users see
E2E latency	Total wall time from question to last word of reply	What users actually wait for
Peak RSS (MB)	System RAM held by the runner process (includes GPU usage on Apple Silicon)	Memory budget on the host

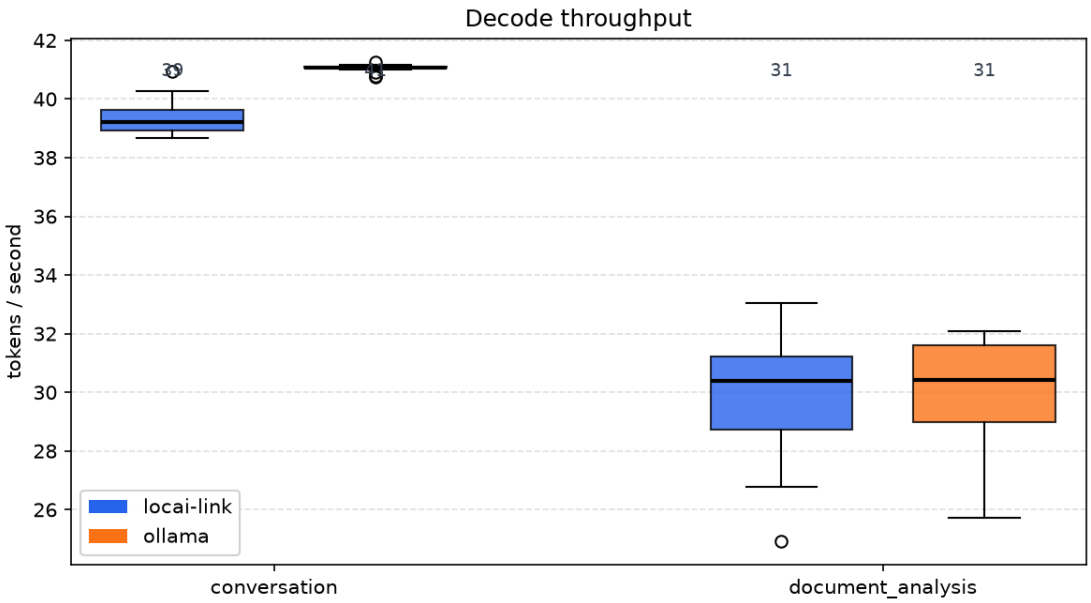
Conversation scenario

Short prompt (~30 tokens), 300-token reply, 20 runs.

Metric	Locai Link	Ollama	Ratio
TTFT median (ms)	**205**	478	Link **2.3×** faster
TTFT p95 (ms)	**215**	490	Link **2.3×** faster
Prefill tok/s	**141**	60	Link **2.3×** faster
Decode tok/s (median)	39	**41**	Tied (Ollama 5 % ahead)
Decode tok/s p95	40	**41**	Tied
E2E latency median (s)	7.9	**7.8**	Tied
Peak RSS (MB)	6 493	**3 332**	Ollama uses 3.2 GB less



TTFT distribution — conversation & document analysis

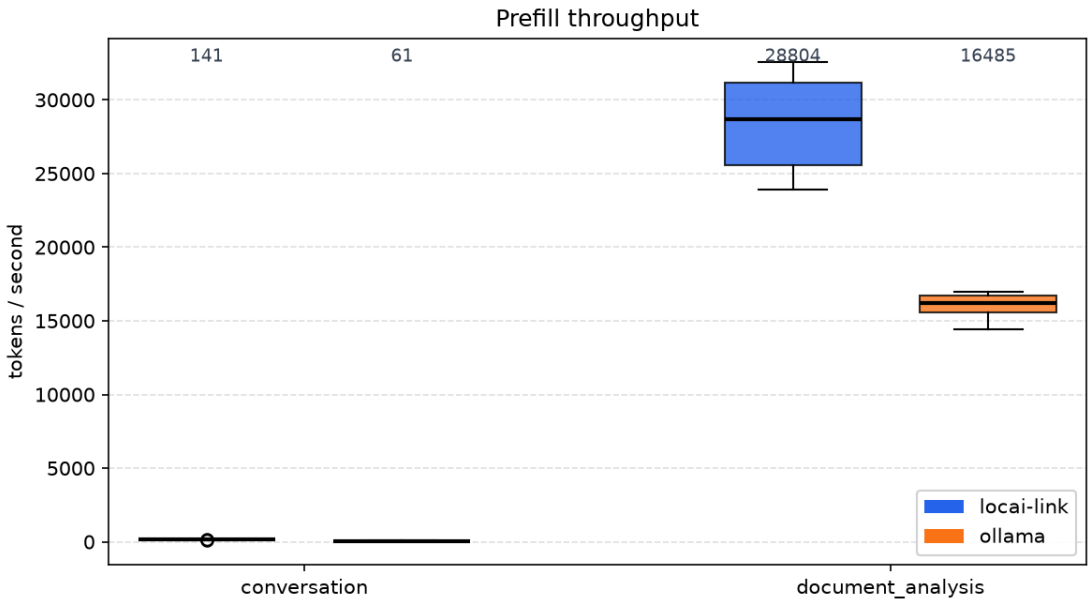


Decode throughput — tokens generated per second

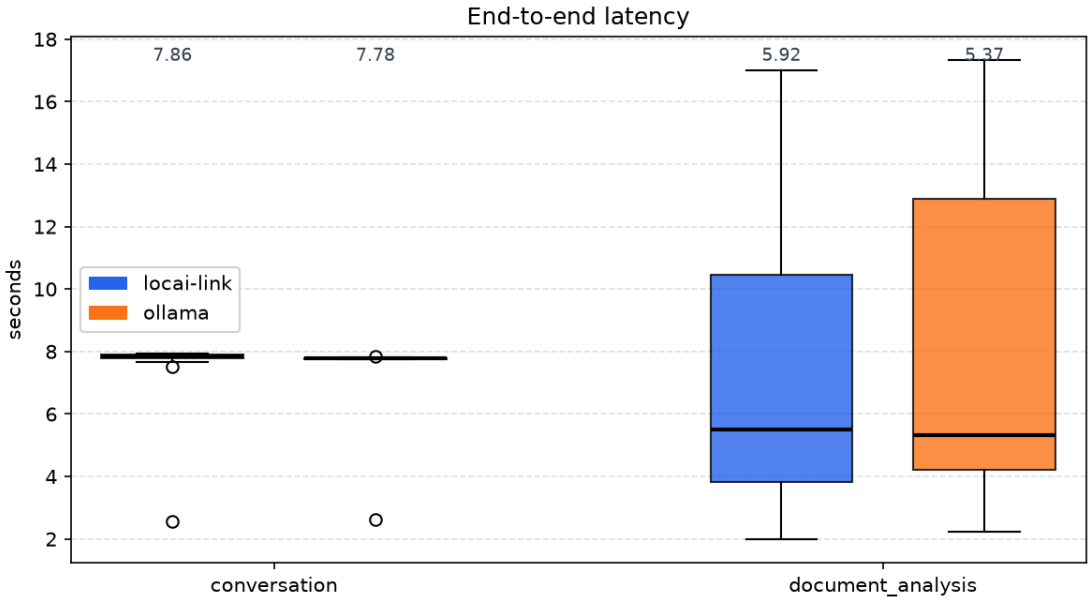
Document-analysis scenario

~9 500-token prompt, reply up to 500 tokens, 10 runs.

Metric	Locai Link	Ollama	Ratio
TTFT median (ms)	**332**	589	Link **1.8×** faster
Prefill tok/s	**28 693**	16 178	Link **1.8×** faster
Decode tok/s (median)	**30.4**	30.4	Tied
Decode tok/s p95	**32.3**	31.9	Tied
E2E latency median (s)	5.5	**5.3**	Tied (Ollama 0.2 s ahead)
Peak RSS (MB)	6 759	**3 429**	Ollama uses 3.3 GB less



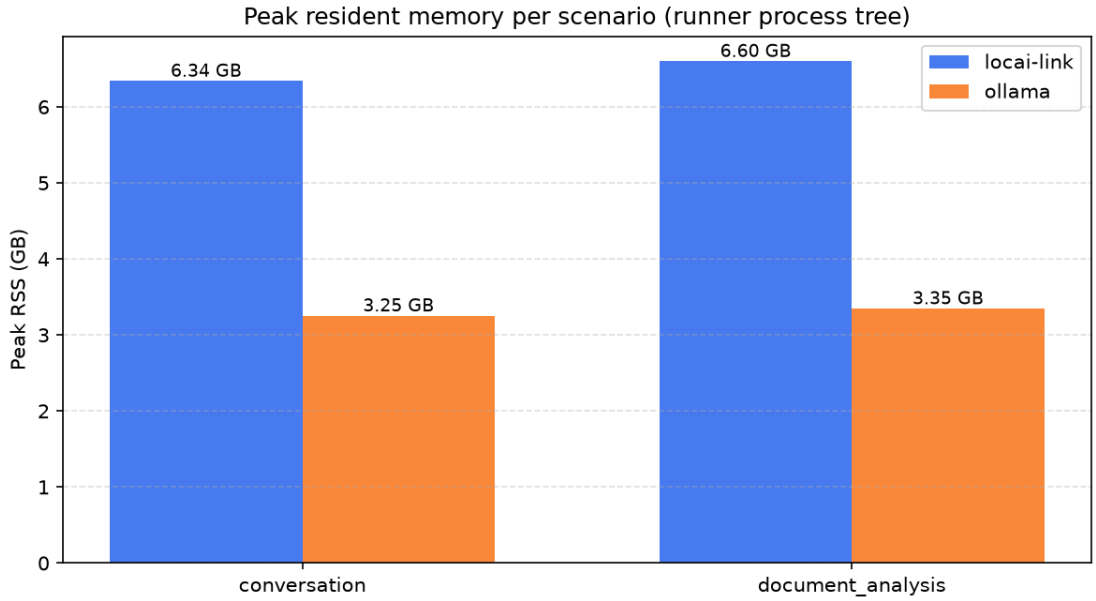
Prefill throughput — reading the document



End-to-end latency

Memory footprint

Peak system RAM held during each scenario.



Peak RSS per backend × scenario

On Apple Silicon the GPU shares the system memory pool — there is no separate VRAM. The RSS chart therefore captures the whole memory footprint per backend, including the Metal-side allocations. Link uses ~3 GB more across both scenarios, which is the cost of its larger KV cache + full-offload approach.

Run-to-run stability

Both backends are very consistent — variance between iterations is small and p95 stays close to the median.



Decode tokens/sec per iteration

What this means for product

- **For Apple Silicon deployment, Link wins prompt processing by ~2×** but decode is at parity with Ollama. The widest gap is on TTFT and prefill — anything where the input is substantial sees Link processing roughly twice as fast.

- **Decode is the same on both** because both backends are bottlenecked by the same constraint — the M2's unified-memory bandwidth. At ~30-40 tok/s, neither runtime can outrun the other on token generation.
- **End-to-end is essentially tied** for typical chat scenarios, because decode time dominates and the prefill advantage is small in absolute milliseconds when the prompt is short.
- **Memory trade is ~3 GB more RAM on Link.** On a 16 GB M2 with both backends loaded simultaneously the headroom is tight; on a 32 GB or 64 GB machine it's negligible.

Caveats worth knowing

1. **Single-stream measurement.** One request in flight at a time per backend. Concurrent load (e.g., 4 parallel chats) wasn't measured and would have different characteristics on both sides.
2. **Different stream protocols.** Link uses OpenAI-compatible SSE on `/v1/chat/completions`; Ollama uses NDJSON on `/api/chat`. This is necessary because Ollama's OpenAI-compat endpoint ignores `think: false`. Both are normalised to the same metric shape by the bench harness.
3. **One model, one quantisation.** Numbers may shift for larger / different-architecture models. Gemma 4 has sliding-window attention and shared KV layers, which affect both runtimes' KV cache strategies.
4. **No per-process VRAM on Apple Silicon.** Apple's unified memory architecture doesn't separate GPU memory from system memory, so the VRAM chart is omitted. RSS captures the full picture.

Reproducibility

Everything lives in `locai-link/bench/`. To re-run end-to-end on macOS:

```
cd locai-link/bench

# Cross-platform launcher (recommended)
uv run --script run.py

# or the bash equivalent:
./run.sh
```

Required on the machine:

- `uv` on PATH
- `ollama` on PATH (Mac native build — uses Metal automatically)
- The Gemma 4 GGUF at `locai-link/models/gemma-4-E2B-it-Q4_K_M.gguf`
- A working Locai Link install (`uv run main.py setup` etc.) — its installer downloads the Metal-accelerated llama.cpp prebuilt on macOS

Graphs regenerate with `uv run --script graphs.py`. Full methodology in `link-vs-ollama-benchmark-proposal.md`.