Local LLM Performance on an Apple M4 Max

Ivan Murashko

Contents

1 From Quality to Performance

Part one of this series, Can Local LLMs Analyze Algorithmic Complexity?, studies quality. It checks whether local models can follow a small C++ program and derive its asymptotic complexity. This part asks how quickly current local packages process the same long prompt and generate text on one Apple M4 Max. Input speed measures prompt processing, output speed measures answer generation, loaded latency measures the time for one request after the model is in memory, and resident memory is the memory reported for that loaded model. A sustained run shows how the output speed changes during repeated requests without cooling. None of these measurements establishes answer quality, so there is no composite score.

The comparison contains seven current packages. A package is the installed set of model weights and metadata distributed under one Ollama tag. Each package was refreshed with `ollama pull`, and its complete SHA-256 digest—a fingerprint of the package bytes—was pinned before collection. Every request sent `think=false`, which asks Ollama to disable a separate reasoning channel and return only the visible answer. The audit confirmed that the published requests contained no hidden generated text.

2 Hardware, Packages, and Protocol

The machine is a MacBook Pro with an Apple M4 Max, 16 CPU cores, 40 GPU cores, and 128GB of unified memory. It runs macOS 26.6.2 on arm64

and Ollama 0.32.15 from the official release. Before collection, the campaign recorded the full SHA-256 digest, advertised context, capabilities, and license metadata of each scheduled package.

Table 1 describes complete installed artifacts, meaning the package weights and their on-disk representation. A GiB is a gibibyte, 2^{30} bytes. A dense model uses one weight path for every token; a mixture-of-experts (MoE) model selects only part of its expert weights. The MLX tag identifies a package prepared for Apple's MLX execution path; GGUF and safetensors are weight containers. Quantization compresses model weights: NVFP4 and Q4_K_M are low-bit representations, while BF16 stores 16-bit floating-point values. The table uses short IDs for the 0.32.15 results.

ID	Installed tag	Architecture	Artifact	Parameters	Quantization	Package GiB
Q38-27	`qwen3.8:27b-mlx`	dense	MLX/safetensors	27.8B	NVFP4	16.927
Q36-27	`qwen3.6:27b-mlx`	dense	MLX/safetensors	27.4B	NVFP4	18.406
Q36-35	`qwen3.6:35b-mlx`	MoE	MLX/safetensors	35.1B	NVFP4	20.405
G4-26	`gemma4:26b`	dense	GGUF	25.8B	Q4_K_M	16.752
G4-31	`gemma4:31b-mlx`	dense	MLX/safetensors	31.7B	NVFP4	17.352
Q35-122	`qwen3.5:122b`	MoE	GGUF	125.1B	Q4_K_M	75.782
Q36-A3B	`qwen3.6:35b-a3b-coding-bf16`	MoE	BF16/safetensors	35.1B	unreported*	65.411

Table 1: The seven artifacts in the published comparison. Values come from Ollama metadata; parameter counts are totals, not active-expert counts. *MLX* and *BF16* come from the tags. *No quantization value is reported.

These labels describe complete packages; they do not isolate the effect of one format while keeping the model unchanged.

We reuse two established tasks. The short task asks for one coherent story. The long-input task contains 180 numbered copies of the same synthetic benchmark note and asks for exactly five summary bullets. Every package receives the same characters, which is why the long prompt can be used as one fixed-text input test. A response cap is the maximum number of tokens that Ollama may generate: 768 for the story and 180 for the summary.

A non-streaming request waits for one complete response instead of delivering answer chunks while they are generated. Temperature zero minimizes random token sampling. Requests use those settings and run one at a time. The context limit is 16,384 tokens, so the complete long prompt fits without clipping. The loaded model process, called the runner, is retained for 30 minutes unless the protocol explicitly unloads it.

For an isolated measurement, the collector unloads the previous runner, loads the required package, and waits until macOS reports 60 continuous seconds in its nominal thermal state, meaning state 0 with no reported thermal pressure. It then verifies the pinned digest, the context limit, one loaded runner, and full GPU residency. Full residency means that Ollama reports the complete runner on the GPU, with no CPU offload. Each package completed both tasks ten times, giving 140 isolated measurements. Package and task order varied so that no package always ran first on the coolest machine.

A sustained measurement keeps one package loaded and sends ten story requests back to back, without a cooling interval. The seven sustained sequences add 70 requests, giving 210 published measurements in total. A structural audit checked every request, timing, digest, runner state, and raw response. All 210 published measurements passed the audit; a thermally contaminated isolated attempt was discarded and repeated after cooling.

3 Isolated Input and Output Performance

Ollama reports prompt processing and answer generation separately. We calculate both rates by the same elementary rule:

$$\text{speed in tokens/s} = \frac{\text{number of tokens}}{\text{operation time in seconds}}.$$

For input speed, the numerator is the prompt-token count and the denominator is prompt-processing time. For output speed, they are generated-token count and generation time. Ollama reports durations in nanoseconds; we convert them to seconds before applying this rule. The first rate measures how quickly the model reads; the second measures how quickly it writes. They are not added or averaged. The short story prompt is too small for a stable input-rate comparison. The three primary rates are therefore story output, long-prompt input, and long-prompt output. Table 2 reports the mean over ten trials. The value after ± is the sample standard deviation, which shows how much the trials varied around that mean.

ID	Story output	Long input	Long output
Q38-27	45.36 ± 1.06	228.17 ± 2.89	39.88 ± 0.89
Q36-27	26.73 ± 0.06	239.96 ± 3.87	24.67 ± 0.21
Q36-35	114.38 ± 0.62	1676.20 ± 7.28	103.96 ± 0.73
G4-26	97.50 ± 0.76	1270.12 ± 16.44	89.57 ± 0.62
G4-31	37.40 ± 0.43	179.79 ± 1.58	26.37 ± 0.41
Q35-122	44.64 ± 0.44	443.73 ± 33.24	41.88 ± 0.18
Q36-A3B	63.26 ± 0.31	352.53 ± 44.23	59.57 ± 0.39

Table 2: Isolated throughput at `think=false`; values are mean ± sample standard deviation over ten accepted trials.

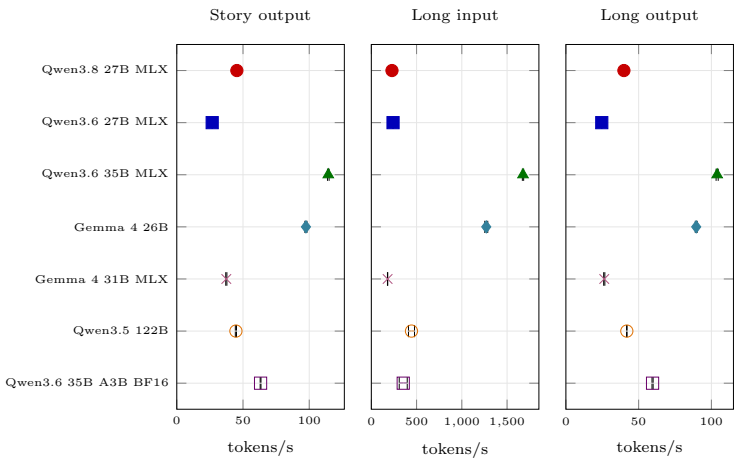

Figure 1: Isolated rates with sample-standard-deviation error bars. The current Qwen3.8 27B package is the filled red point; every point comes from the current Ollama 0.32.15 campaign.

Within this campaign, Qwen3.6 35B MLX leads all three rates: 114.38 story-output, 1676.20 long-input, and 103.96 long-output tokens/s. Gemma4 26B is second at 97.50, 1270.12, and 89.57. The Qwen model is an MoE, so its 35.1B total parameters are not a dense compute count; package size and total parameters alone do not predict throughput.

The near-size dense Qwen pair is more direct. Qwen3.8 reaches 45.36 ± 1.06 story and 39.88 ± 0.89 long-output tokens/s, versus 26.73 ± 0.06 and 24.67 ± 0.21 for Qwen3.6. Qwen3.6 ingests the long prompt about 5% faster. Every Qwen3.8 story trial lies between 43.77 and 46.61 tokens/s.

Qwen3.5 122B illustrates the other gap between size and speed. It generates a story at 44.64 tokens/s, almost exactly Qwen3.8's rate, despite occupying more than four times as much resident memory. Its MoE architecture and GGUF package make the total parameter count a poor proxy for active computation.

A tokenizer is the package-specific rule that converts text into model tokens. The same synthetic 180-note prompt therefore produces different token counts for different packages. Table 3 reports both that token count and the seconds required to process the identical text.

ID	Prompt tokens	Input seconds	Summary output tokens
Q38-27	9223	40.43 ± 0.51	80
Q36-27	9223	38.45 ± 0.63	86
Q36-35	9223	5.50 ± 0.02	84
G4-26	9404	7.41 ± 0.10	66
G4-31	9404	52.31 ± 0.46	51
Q35-122	9223	20.89 ± 1.57	81
Q36-A3B	9223	26.53 ± 3.30	88

Table 3: Processing the same synthetic 180-note prompt over ten accepted trials. Token counts depend on the package tokenizer.

For the same text, Qwen tokenizers produce 9223 tokens; Gemma tokenizers, 9404. Qwen3.6 35B MLX processes it in 5.50 seconds, Gemma4 26B in 7.41, and Gemma4 31B MLX in 52.31. Input tokens/s compares the tokenizer-specific sequences; input seconds compares the time required for identical text. All 70 summaries returned exactly five visible bullets and ended before the 180-token cap. The separate story task used a 768-token cap. Every package reached that cap except Gemma4 31B, which stopped at 711 ± 14 tokens. This is completion behavior, not a quality score.

4 Practical Latency and Resident Memory

Fresh-runner preload time is the wall-clock time required to load a package before a request. Loaded request time starts after that runner is already in memory and uses Ollama's server-reported total duration. Inference time is the part of that duration spent processing the prompt and generating the answer. Resident GiB is Ollama's reported loaded-runner memory, not peak unified memory for the entire computer. Visible characters/s is the number of answer characters divided by generation time; it supplements the tokenizer-dependent token rate but also depends on the generated text.

ID	Preload wall s	Loaded request s	Inference s	Visible char/s	Resident GiB	Prompt/output tokens
Q38-27	2.02 ± 1.27	17.47 ± 0.56	17.45 ± 0.56	189.8 ± 4.4	17.27	38/768
Q36-27	2.12 ± 1.37	29.20 ± 0.07	29.14 ± 0.06	118.9 ± 0.2	18.74	38/768
Q36-35	2.63 ± 1.49	7.00 ± 0.06	6.96 ± 0.04	502.9 ± 2.7	20.53	38/768
G4-26	4.27 ± 3.12	8.08 ± 0.09	8.06 ± 0.09	431.1 ± 3.4	16.42	39/768
G4-31	2.35 ± 1.26	19.50 ± 0.51	19.48 ± 0.51	168.9 ± 2.4	16.96	$39/711 \pm 14$
Q35-122	17.48 ± 0.08	17.63 ± 0.19	17.62 ± 0.19	190.3 ± 1.9	74.16	38/768
Q36-A3B	21.36 ± 1.32	35.81 ± 2.18	35.58 ± 2.18	265.8 ± 1.3	65.53	38/768

Table 4: Fresh-runner cost, loaded story-request time, visible output, and runner-reported residency.

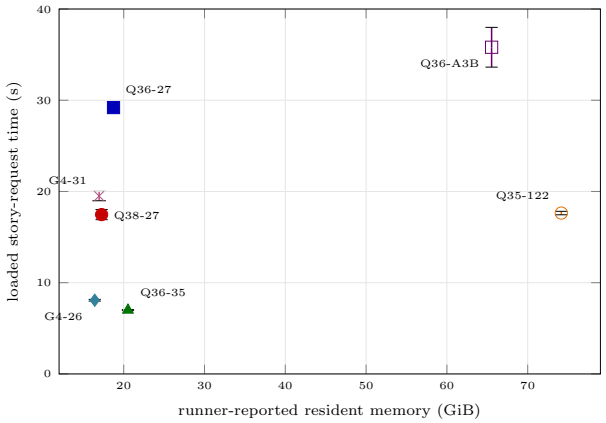

Figure 2: Loaded story-request time versus runner-reported resident memory. Each point is the ten-trial mean; vertical bars show sample standard deviation. Lower and farther left are better. Gemma4 31B produced 711 ± 14 tokens, while the other packages produced 768, so this is request time rather than length-normalized generation time.

Qwen3.6 35B MLX has the lowest loaded request time at 7.00 seconds, followed by Gemma4 26B at 8.08. Qwen3.5 122B takes 17.48 seconds to preload and occupies 74.16 GiB, yet its loaded story latency is similar to Qwen3.8's. The BF16 package needs 21.36 seconds to preload and 65.53 GiB after a story. Its 63.26 output tokens/s do not imply low latency: Ollama attributes 23.44 ± 2.17 seconds to evaluating the 38-token story prompt before the 768-token output begins.

The smaller MLX and Gemma4 26B runners occupy roughly 16–21 GiB. Preload here means a fresh runner, not cold storage: unloading does not flush the macOS filesystem cache.

5 Sustained Performance

The sustained test starts with a cooled machine and sends ten story requests without a cooling interval. To measure the slowdown, we average the output speeds of requests 1–3, average the speeds of requests 8–10, and calculate

$$\text{speed retained} = 100 \times \frac{\text{average speed of requests 8–10}}{\text{average speed of requests 1–3}}.$$

Thus, 100% means no slowdown. A result of 80% means that the closing requests ran at four fifths of the opening speed, or 20% slower. Three-request averages prevent one unusually fast or slow request from dominating the result. For Qwen3.8 27B, the opening average is

$$\frac{46.73 + 47.78 + 40.34}{3} = 44.95 \text{ tokens/s},$$

and the closing average is

$$\frac{19.65 + 21.72 + 23.27}{3} = 21.55 \text{ tokens/s}.$$

Its retained speed is therefore $100 \times 21.55/44.95 = 47.9\%$.

macOS reports four thermal-pressure states: nominal, fair, serious, and critical. The table records the first request in which the state changed from nominal to any higher state, together with the elapsed time from the start of the sequence.

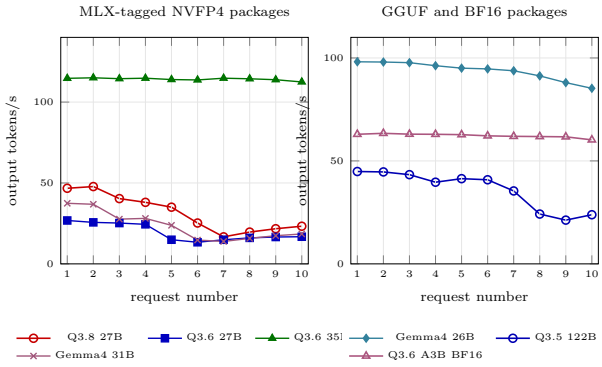

Figure 3: Ten back-to-back story rates. Thermal changes are retained.

ID	Request 10 tokens/s	Speed retained	First thermal transition
Q38-27	23.27	47.9%	request 4 at 54.2s
Q36-27	16.78	63.6%	request 3 at 83.4s
Q36-35	112.44	99.0%	not observed
G4-26	85.25	90.0%	request 8 at 61.8s
G4-31	18.49	50.7%	request 4 at 79.6s
Q35-122	23.86	52.2%	request 5 at 87.5s
Q36-A3B	60.23	97.1%	request 9 at 134.8s

Table 5: Final-request rate, percentage of opening speed retained, and first thermal-state transition.

Qwen3.6 35B MLX stays between 112.44 and 115.02 tokens/s, retains 99.0%, and never leaves the nominal state. BF16 falls gently from 62.93 to 60.23 and retains 97.1%. Gemma4 26B declines from 98.14 to 85.25 and retains 90.0%, the best sustained result among the dense packages in the current campaign.

Qwen3.8 begins at 46.73, enters the fair state during request 4, and ends at 23.27. Dense Qwen3.6 begins at 26.82 and ends at 16.78. Gemma4 31B and Qwen3.5 122B retain roughly half of their initial throughput. No package

reached a serious or critical thermal state. The OS signal is coarse: nominal does not imply equal die temperature, clock, or power, so these are workload curves, not a physical thermal model.

6 Claude and GPT-5.6 Sol API Speed Reference

The local campaign cannot measure the hardware behind a proprietary service. It can nevertheless ask a narrower question: is local output generation in the same practical range, and what input-side latency information is public? Table 6 records a dated snapshot from Artificial Analysis. Its independent API benchmark uses streaming and varied prompts. The values are rolling 72-hour medians retrieved on 29 August 2026. The Claude row uses the first-party Anthropic endpoint. The OpenAI model-side reference uses GPT-5.6 Sol through OpenAI's API. GPT-5.6 Sol is the model-side reference for Codex here; the row does not measure a complete Codex agent task with tool calls.

The provider APIs do not expose prompt-evaluation time directly. Artificial Analysis reports the median time to the first visible answer token for target prompt lengths of approximately 1000 and 10,000 tokens. For a reasoning model, that delay includes hidden reasoning before the answer. We call the following inverse first-answer-latency slope the *artificial input speed*:

$$\text{artificial input speed} = \frac{10000 - 1000}{t_{10\text{K}} - t_{1\text{K}}}.$$

For example, GPT-5.6 Sol takes 13.29 seconds for the 1K workload and 21.10 seconds for the 10K workload. The additional 9000 target tokens therefore add 7.81 seconds, giving an artificial input speed of 1151.9 target tokens/s. This subtraction removes most fixed request delay, but the two workloads use different prompts and rolling medians, and their hidden reasoning may also differ. The result is not a direct measure of model prefill, meaning prompt processing before generation. We use it only for an approximate comparison with Ollama's measured input speed, not as a substitute for that measurement.

External reference	Artificial input speed (target tokens/s)	Output speed (tokens/s)
Claude Opus 5, high (Anthropic)	1003.1	52.6
GPT-5.6 Sol, high (OpenAI)	1151.9	73.3

Table 6: The artificial input speed is calculated from Artificial Analysis's 1K and 10K first-answer times; output speed is measured by Artificial Analysis through each provider's own API. Source pages: Claude and GPT-5.6 Sol. The artificial input speed is a latency-derived estimate, not a direct prefill measurement.

Artificial Analysis counts output with one common tokenizer; a tokenizer is the rule that converts text to tokens. It calculates output speed after the first answer chunk, whereas Ollama uses each local package's native token count over its complete generated sequence. The prompts, temperature, network path, and sampling window also differ. The external values therefore belong in a reference table, not as new points in the controlled local figure.

7 Conclusion

The results compare complete installed packages; they do not isolate the effect of MLX, GGUF, precision, or model architecture.

The practical conclusions are as follows:

- **Qwen3.6 35B MLX is the best overall performer.** It has the fastest input and output rates, uses 20.5 GiB, and retains 99.0% of its opening speed.
- **Gemma4 26B is the best dense package.** It is second in all three throughput measurements, uses 16.4 GiB, and retains 90.0% under repeated load.
- **Qwen3.8 27B is a reasonable middle choice, but sustained speed is its weakness.** Its isolated story speed is 45.36 tokens/s with 17.3 GiB resident, but it retains only 47.9% under repeated load.
- **Qwen3.6 27B and Gemma4 31B MLX are weak for their memory class.** They use 17–19 GiB but have lower output rates and substantial sustained slowdown.
- **The two large-memory packages are poor speed-per-memory choices on this machine.** Qwen3.5 122B uses 74.2 GiB for Qwen3.8-like output speed. Qwen3.6 A3B BF16 reaches 63.26 tokens/s but uses 65.5 GiB and handles the prompt slowly.
- **Gemma4 26B is the closest local speed match to GPT-5.6 Sol, the model-side Codex reference.** Its long-prompt input and output rates are 1270.12 and 89.57 tokens/s, compared with the artificial input rate of 1151.9 and the measured output rate of 73.3 for GPT-5.6 Sol; the gaps are 10.3% and 22.2%. Claude has no similarly close match on both rates. Giving input and output equal weight, Qwen3.5 122B has a combined absolute percentage gap of 76.2 points, slightly below 78.0 for Qwen3.6 A3B BF16. Its rates are 443.73 input and 41.88 output tokens/s versus Claude's artificial 1003.1 and measured 52.6. Qwen3.6 A3B BF16 is closest to Claude on output alone at 59.57 tokens/s. The artificial input rates are latency-derived estimates, so these comparisons are approximate.

This comparison covers speed, not complete working ability. Claude and Codex agents also reason, call tools, and correct their work; part one studies the quality difference. Here the narrower result is that local throughput is not the main obstacle for the two fastest packages.