?? Optimization & Model Theory

Focus: Improving the model's raw intelligence and speed. The "Making it Better" stage. Technical mechanics of the LLM.

- Agentic LLM Inference Tuning Reference for Qwen 3.6 and Gemma 4
- Qwen 3.6 27B and 35B MTP vs Standard on 16GB GPU

Agentic LLM Inference Tuning Reference for Qwen 3.6 and Gemma 4

This page is a practical reference for agentic LLM inference tuning (temperature, top_p, top_k, penalties, and how they interact in multi-step and tool-heavy workflows).

It sits alongside the broader LLM performance engineering hub and matches best with a clear LLM hosting and serving story—throughput and scheduling still dominate when the model is starved, but unstable sampling burns retries and output tokens before the GPU does.

This page consolidates:

- Vendor recommended parameters
- Embedded defaults from GGUF and APIs
- Real-world community findings
- Agentic workflow optimizations

Current Focus:

- Qwen 3.6 (dense and MoE)
- Gemma 4 (dense and MoE)

If you run terminal agents such as OpenCode, pair this reference with local LLM behavior in OpenCode so workload-level results and sampler defaults stay aligned.

> **Goal:** Provide a single place to configure models for agent loops, coding, and multi-step reasoning.

TLDR Reference Table: Agentic Defaults

Model	Mode	Temp	Top_p	Top_k	Presence Penalty
Qwen 3.5 27B	thinking general	1.0	0.95	20	0.0
Qwen 3.5 27B	coding	0.6	0.95	20	0.0
Qwen 3.5 35B MoE	thinking	1.0	0.95	20	1.5
Qwen 3.5 35B MoE	coding	0.6	0.95	20	0.0
Gemma 4 31B	general	1.0	0.95	64	0.0
Gemma 4 31B	coding	1.2	0.95	65	0.0
Gemma 4 26B MoE	general	1.0	0.95	64	0.0
Gemma 4 26B MoE	coding	1.2	0.95	65	0.0

What "Agentic Inference" Actually Means

Most parameter guides assume chat, single-shot completion, or human interaction. Agentic systems are different; they require multi-step reasoning, tool calling, consistent outputs, and low error propagation.

Core Shift in Priorities

Use Case	Priority
Chat	Natural language quality
Creative	Diversity

Use Case	Priority
Agentic	Consistency + reasoning stability

Qwen 3.6 Tuning

Dense vs MoE

The 27B dense and 35B-A3B MoE models differ significantly regarding MTP (Multi-Token Prediction) speculative decoding: MTP is worthwhile for the 27B dense model but collapses the usable context window on 16 GB VRAM for the 35B MoE. See the Qwen 3.6 MTP vs Standard benchmark for measured generation speeds and VRAM overhead.

Qwen is one of the few families where **MoE requires different penalties**.

Dense (27B)

- Stable and predictable; no routing complexity.
- **Recommended:** `presence_penalty = 0.0`

MoE (35B-A3B)

- Expert routing per token creates a risk of repetition loops.
- **Recommended:** `presence_penalty = 1.5` (general), `0.0` (coding).

Why this matters: MoE models can get stuck reusing the same experts. Presence penalty helps diversify token paths and improve reasoning exploration.

Qwen Agentic Coding Setup

Correct setup:

- `temperature = 0.6`
- `top_p = 0.95`
- `top_k = 20`
- `presence_penalty = 0.0`

Why low temperature works: Coding agents need deterministic outputs, repeatable tool calls, and stable formatting. Higher temperature often breaks JSON and introduces hallucinated APIs.

Gemma 4 Tuning

Gemma behaves differently. Because official model cards are often implicit, real tuning comes from Google AI Studio, GGUF defaults, and community benchmarks.

The Counter-Intuitive Finding

Gemma 4 performs better with **higher temperature**.

Temp	Result
0.5	Poor reasoning
1.0	Stable baseline
1.2 to 1.5	Best coding performance

Hypothesis: Training distribution favors exploration, and the model compensates for a lack of explicit chain-of-thought control by expanding the solution search space.

Gemma Agentic Coding Setup

Recommended:

- `temperature = 1.2`
- `top_p = 0.95`
- `top_k = 65`
- `penalties = 0.0`

Note: Do not apply the traditional "low temp for code" rule blindly to Gemma.

Thinking Mode and Agent Systems

Both Qwen and Gemma support reasoning modes. Agent loops require intermediate reasoning, error recovery, and multi-step planning.

Practical rule: Always enable thinking mode for coding agents, tool use, and multi-step tasks.

Parameter Strategy by Use Case

- **Coding agents:** Prioritize determinism, minimize penalties, and use stable sampling.
- **Reasoning agents:** Use moderate temperature to allow exploration while preserving structure.
- **Tool calling:** Require strict formatting, low randomness, and consistent token patterns.

Schema and JSON tooling are orthogonal to logits; combine these rules with structured output patterns for Ollama and Qwen3 to reduce validator retries.

Vendor Defaults vs Reality

Vendor defaults are generally safe and generic, but not optimized. Community findings often show better task-specific tuning:

- **Gemma:** Official guidance is sparse; community suggests high temp for coding.
- **Qwen:** Official sections are inconsistent; community values have converged.

Practical Deployment Notes

Under concurrency, queueing and memory splits interact with retries as much as sampling does. Read how Ollama handles parallel requests to tune `OLLAMA_NUM_PARALLEL`.

- **Ollama:** Works well for both families; verify GPU compatibility.
- **vLLM:** Stable for production with support for advanced sampling.
- **llama.cpp:** Requires careful sampler ordering; always enable Jinja for modern models.

Key Takeaways

- There is no universal parameter set.

- Architecture matters more than model size.
- Agentic systems require different tuning than chat.
- Community benchmarks often lead vendor documentation.

Final Opinion: Treat inference tuning as a first-class system design problem, not just a config file. Most outdated guides assuming "low temp for code" do not apply to modern models.

Qwen 3.6 27B and 35B MTP vs Standard on 16GB GPU

I tested Speculative decoding (Multi-Token Prediction, MTP) performance in Qwen 3.6 27B and 35B on an RTX 4080 with 16 GB VRAM.

For a broader view of token speeds and VRAM trade-offs across more models on the same hardware, see 16 GB VRAM LLM benchmarks with llama.cpp.

What MTP (Multi-Token Prediction) Is

Multi-Token Prediction is a form of speculative decoding built directly into certain model checkpoints. Instead of predicting one token per forward pass, the model carries extra "MTP heads" that propose several future tokens in a single step --- then verifies them in parallel. If the guesses are accepted, the effective throughput rises without changing the output quality.

The Qwen 3.6 family ships both standard GGUF files and MTP-enabled variants. In llama.cpp, MTP is activated through:

```
--spec-type draft-mtp --spec-draft-n-max 3
```

`--spec-draft-n-max` is the key tuning knob. It sets how many speculative tokens the MTP head proposes at each step. Higher values give a potential speed boost but cost extra VRAM for the draft buffers --- a real constraint on 16 GB cards.

What and How I Tested

I tested how the two Qwen 3.6 models behave with MTP enabled versus standard decoding on a GPU with 16 GB VRAM (RTX 4080).

To fit model weights and KV cache into VRAM I used heavily quantised variants:

- `Qwen3.6-27B-UD-IQ3_XXS` and `Qwen3.6-27B-UD-IQ3_XXS-MTP`

- `Qwen3.6-35B-A3B-UD-IQ3_S` and `Qwen3.6-35B-A3B-UD-IQ3_S-MTP`

Two context budgets are tracked per run:

- Avg Ctx --- the context size at which llama.cpp occupies ~14.8 GB VRAM, leaving other apps (Xorg, GNOME Shell, Cursor) a comfortable ~500 MB buffer.
- Max Ctx --- the largest context llama.cpp could allocate given that the same desktop apps already hold ~500 MB VRAM.

A key reason for keeping the average context at a practical target is that Hermes Agent --- which I use as the primary AI assistant connecting to llama.cpp on this machine --- requires at least 64 K context by default and will reject models with a smaller window at startup. Models below that threshold cannot maintain enough working memory for multi-step tool-calling workflows. For llama.cpp this means passing `--ctx-size 65536` or larger. Any MTP configuration that compresses the average usable context significantly below 64 K is therefore unsuitable for daily Hermes workloads, which is why the Avg Ctx numbers in the tables below are the most decision-relevant ones.

Both KV cache quantisation levels were tested: q8 (higher quality, more VRAM) and q5 (lower VRAM, longer context). Be aware that moving from q8 to q5 KV cache can cause a noticeable quality drop --- in my testing the degradation was significant enough to make q5 unsuitable for my workloads. The speed and context numbers for q5 are included for completeness, but you should test response quality on your own tasks before committing to it.

Qwen 3.6 27B MTP vs Standard

KV Cache q8

	MTP max 1	MTP max 2	MTP max 3	MTP max 4	Standard (IQ3_XXS)
Prompt Speed	148 t/s	151 t/s	148 t/s	147 t/s	200 t/s
Gen Speed	65 t/s	75 t/s	73 t/s	75 t/s	45 t/s
Avg Ctx	40 K	40 K	40 K	30 K	80 K
Max Ctx	60 K	60 K	60 K	50 K	100 K

With q8 KV cache, MTP at `--spec-draft-n-max 2` delivers ~67 % faster generation (75 vs 45 t/s) at the cost of halving the average context window from 80 K to 40 K. Prompt ingestion speed drops from 200 to ~150 t/s because MTP requires device-to-host transfers during the prefill phase.

KV Cache q5

	MTP max 1	MTP max 2	MTP max 3	MTP max 4	Standard (IQ3_XXS)
Prompt Speed	145 t/s	144 t/s	141 t/s	139 t/s	191 t/s
Gen Speed	57 t/s	62 t/s	67 t/s	66 t/s	41 t/s
Avg Ctx	70 K	60 K	60 K	50 K	130 K
Max Ctx	100 K	100 K	90 K	80 K	160 K

Switching to q5 KV cache recovers meaningful context: `--spec-draft-n-max 1` gives 70 K average context at 57 t/s --- a 39 % generation speedup over standard decoding while still keeping the context window at a useful size. At `--spec-draft-n-max 3` the context drops to 60 K but generation reaches 67 t/s (+63 %).

Qwen 3.6 27B Takeaway

MTP is genuinely useful for the 27B dense model. The sweet spot on 16 GB VRAM is:

- q8 KV + `--spec-draft-n-max 2` --- best raw speed (75 t/s), context down to 40--60 K
- q5 KV + `--spec-draft-n-max 1` --- best speed-vs-context balance (57 t/s, 70 K avg context)

Qwen 3.6 35B MTP vs Standard

The 35B model is a Mixture-of-Experts (MoE) architecture (`35B-A3B` means 35B total parameters, ~3B active per token). MoE models usually benefit more from MTP because the sparse routing keeps the MTP head computationally cheap relative to a full forward pass.

KV Cache q8

	MTP max 1	MTP max 2	MTP max 3	MTP max 4	Standard (IQ3_S)

Prompt Speed	277 t/s	277 t/s	265 t/s	275 t/s	368 t/s
Gen Speed	186 t/s	189 t/s	180 t/s	171 t/s	146 t/s
Avg Ctx	15 K	10 K	---	---	80 K
Max Ctx	80 K	70 K	60 K	50 K	150 K

The MoE architecture delivers impressive raw generation speed with MTP (+27 % at max 1, +29 % at max 2 vs standard 146 t/s). But the practical problem is the average context. With q8 KV cache, even `--spec-draft-n-max 1` only gives 15 K average context --- barely enough for modest tasks. Higher draft depths have no viable average context at all on a 16 GB card.

This is the central VRAM cost question for MTP on consumer hardware: the extra draft buffers eat into the remaining VRAM budget directly, and the 35B-A3B model with q8 KV cache leaves very little headroom.

KV Cache q5

	MTP max 1	MTP max 2	MTP max 3	MTP max 4	Standard (IQ3_S)
Prompt Speed	264 t/s	266 t/s	270 t/s	264 t/s	343 t/s
Gen Speed	151 t/s	147 t/s	137 t/s	131 t/s	122 t/s
Avg Ctx	10 K	---	---	---	120 K
Max Ctx	120 K	110 K	110 K	80 K	200 K

q5 KV cache only marginally improves the average context story. `--spec-draft-n-max 1` gives 10 K average context at 151 t/s. Standard decoding at q5 gives 122 t/s with 120 K average context.

Qwen 3.6 35B Takeaway

On a 16 GB GPU the 35B MoE model with MTP faces a hard wall: the average usable context collapses to 10--15 K tokens, making it impractical for real workloads. Standard decoding at 122--146 t/s with 80--120 K context is significantly more useful.

If you have 24 GB+ VRAM, the 35B + MTP combination becomes much more attractive --- the context window issue disappears and you keep the speed benefit.

Choosing the Right `--spec-draft-n-max` Value

The question of how many speculative tokens to propose per step (`--spec-draft-n-max`) does not have a single right answer --- it depends on both model architecture and available VRAM:

- For 27B dense on 16 GB: `--spec-draft-n-max 2` with q8 KV is the fastest, `--spec-draft-n-max 1` with q5 KV is the most context-friendly.
- For 35B MoE on 16 GB: `--spec-draft-n-max 1` is the only option that keeps any usable context, and even then only marginally.
- Higher values (`3`, `4`) increase VRAM pressure without proportional speed gains --- at max 4 you're spending roughly the same extra VRAM as max 2 but gen speed doesn't keep pace.

How to Enable MTP in llama.cpp

Make sure you use an MTP-enabled GGUF (the filename contains `MTP`). If you are new to llama.cpp flags, the llama.cpp Quickstart with CLI and Server covers all the fundamentals. Then launch llama-server or llama-cli with:

```
llama-server\
  --model Qwen3.6-27B-UD-IQ3_XXS-MTP.gguf\
  --ctx-size 40000\
  -ngl 99 --flash-attn on\
  --cache-type-k q8_0 --cache-type-v q8_0\
  --spec-type draft-mtp\
  --spec-draft-n-max 2
```

For q5 KV cache, replace `q8_0` with `q5_1` or `q5_0` and adjust `--ctx-size` upward:

```
llama-server\
  --model Qwen3.6-27B-UD-IQ3_XXS-MTP.gguf\
  --ctx-size 80000\
  -ngl 99 --flash-attn on\
  --cache-type-k q5_1 --cache-type-v q5_1\
  --spec-type draft-mtp\
```

```
  --spec-draft-n-max 1
```

MTP is activated automatically once llama.cpp sees the MTP heads in the GGUF file and `--spec-type draft-mtp` is set. So standard Qwen3.6-27B-UD-IQ3_XXS.gguf will not work in MTP mode, you will need Qwen3.6-27B-UD-IQ3_XXS-MTP.gguf. But the Qwen3.6-27B-UD-IQ3_XXS-MTP.gguf can work in both Speculative decoding mode and autoregressive one.

Conclusion

On a 16 GB GPU (RTX 4080), with these quants, llama.cpp's MTP is a clear win for Qwen 3.6 27B and a net negative for Qwen 3.6 35B in practical use:

Qwen 3.6 27B (IQ3_XXS) --- MTP is worthwhile:

- q8 KV + MTP max 2 ? ~67 % faster generation, context 40--60 K (vs 80--100 K without MTP)
- q5 KV + MTP max 1 ? ~39 % faster generation, context 70--100 K (vs 130--160 K without MTP)
- Good balance of speed and VRAM efficiency at `--spec-draft-n-max 2`

Qwen 3.6 35B (IQ3_S) --- MTP is not practical at 16 GB:

- Generation speed is 27--29 % higher but average context collapses to 10--15 K at q8, 10 K at q5
- Standard decoding at 122--146 t/s with 80--120 K context is more useful for real tasks
- The situation improves substantially on 24 GB+ VRAM

On paper, q5 KV cache is the obvious answer for maximising context window while keeping MTP speed gains --- but in practice the quality drop moving from q8 to q5 can be significant. Test q5 on your own tasks before adopting it; for my workloads the degradation was unacceptable, and q8 with a tighter context budget remains the better trade-off.

For the wider picture of LLM serving options and infrastructure trade-offs, see the LLM Hosting in 2026 pillar and LLM Performance in 2026. If you are tuning Qwen 3.6 sampler settings alongside MTP, the Agentic LLM Inference Parameters Reference for Qwen 3.6 and Gemma 4 is a useful companion.