Best 24GB Backend Shootout: ik_llama vs BeeLlama vs llama.cpp

May 22, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** On my RTX 3090 with Qwen 3.6 27B, ik_llama.cpp with MTP and BeeLlama with DFlash finish the am17an 9-prompt harness in 22.38s and 23.00s respectively — both ~1.6x faster than mainline llama.cpp's 37.22s baseline. ik_llama hits 88.5% draft acceptance with small drafts (981 total tokens); BeeLlama hits 37.4% with 2,809 drafts (3x wider batches). Same wall clock, opposite strategies. BeeLlama peaks at 119 tok/s on Python code generation; ik_llama wins by 2x on short translations. Pick by workload, not by acceptance rate alone.

On my RTX 3090, both ik_llama.cpp with MTP and BeeLlama with DFlash just finished the same 9-prompt harness in 22 seconds. Mainline llama.cpp took 37 seconds on the same machine, same harness, same Qwen 3.6 27B model class. Two backends, two different speculative decoding strategies, near-identical wall clock. The question of “which backend should I run” depends entirely on what you’re running through it.

The surprise underneath the tie: ik_llama hit 88.5% draft acceptance with tight, small batches. BeeLlama hit 37.4% with batches three times wider. Both ended up at the same wall clock. That’s the editorial hook of this piece — and the reason a naive “higher acceptance is better” read of these numbers leads you somewhere wrong. Below: the three configs, the numbers, the per-prompt breakdown, and when each one wins.

The three configurations

Same Miu — single RTX 3090 24GB, Linux, CUDA 12.8, sm_86 — and the same 9-prompt harness that the May 19 MTP piece used. Each backend runs at its canonical recommended config from the community thread or quickstart docs that ship with it, paired with the model variant the community recommends for that backend.

Backend	Build / Commit	Model	Spec config
Mainline llama.cpp	`b9079-f9cd456ea` (May 8)	Unsloth UD-Q4_K_XL (17 GB)	None (baseline)
ik_llama.cpp	`4530` / `48a55f74` (May 21)	Ubergarm IQ4_KS-MTP (16 GB)	`--spec-stage mtp --draft 3`

Backend	Build / Commit	Model	Spec config
BeeLlama	`b9353-ba1fdce59` (May 21)	Q5_K_S target (18 GB) + DFlash 1.7B draft (986 MB)	`-md <draft> -ctk turbo4 -ctv turbo3_tcq -ub 256`

Common server flags across all three: `-ngl 99 -c 10000 --no-mmap -fa on -np 1`. Each run capped at `n_predict=192`, `temperature=0.0`, `seed=42` per the am17an harness. All numbers reproducible.

The headline numbers

Backend	Wall total (s)	Mean tok/s	Aggregate accept	Total draft tokens
Mainline baseline	37.22	40.16	n/a	0
ik_llama + MTP n=3	**22.38**	**74.08**	**88.5%**	981
BeeLlama + DFlash	**23.00**	**73.31**	**37.4%**	**2,809**

ik_llama.cpp wins wall clock by 0.62 seconds. BeeLlama is within shouting distance on both wall clock and mean throughput, but its acceptance rate is less than half of ik_llama's. Both beat mainline by roughly the same factor — 1.66x and 1.62x respectively.

The "Total draft tokens" column is where the strategy difference lives. BeeLlama generated 2,809 draft tokens to ik_llama's 981 — nearly 3x as many. BeeLlama's wider draft batches let it tolerate a much lower per-draft acceptance and still finish in the same time. Two paths, same destination.

Two strategies, same destination

Speculative decoding lets a fast draft model propose tokens that the slow target model verifies in parallel. The target either accepts the drafts as-is, which is cheap, or rejects them and falls back to one-token-at-a-time. Higher acceptance means more bypassed work, which usually means faster wall clock. Usually.

MTP (Multi-Token Prediction) attaches small prediction heads directly inside the target model. Each forward pass produces both the next "real" token and N draft tokens that share the heavy lifting (attention, MLP), so the draft is essentially free. The ik_llama.cpp implementation here uses 3 draft heads (`--draft 3`), so each verification cycle nominally produces 4 tokens — 1

verified plus 3 drafted-then-verified. Because the drafts come from the same model architecture as the target, acceptance is high. 88.5% on this harness.

DFlash uses a separate small drafter model. In BeeLlama's case it's the 986 MB "DFlash Drafter 3.6" file — a specialized 1.7B model with 5 layers that hooks into the target's layer indices [1, 16, 31, 46, 61]. The drafter produces wider batches and BeeLlama's adaptive controller auto-scales batch sizes based on real-time profit metrics during the run. On this bench the controller settled into pretty wide batches by the third prompt and stayed there. Wider batches means lower per-token acceptance (37.4% aggregate), but more accepted tokens per verification cycle to compensate.

Two different approaches to the same problem. The wall clock is the result.

When BeeLlama wins

The per-prompt detail is where the article earns its weight. Here's the full per-prompt breakdown across all three backends:

Prompt	Mainline tok/s	ik_llama tok/s	BeeLlama tok/s	ik MTP accept	BeeLlama DFlash accept
code_python	40.3	79.6	**118.9**	91.3%	65.8%
code_cpp	40.2	75.1	**93.3**	91.0%	45.0%
explain_concept	40.2	78.3	76.5	93.2%	32.7%
summarize	39.3	73.7	65.8	82.1%	31.9%
qa_factual	40.2	73.9	59.0	85.0%	33.3%
translation	41.5	68.0	34.2	75.0%	10.6%
creative_short	40.0	69.6	61.0	86.5%	39.1%
stepwise_math	40.1	84.0	**89.0**	**96.3%**	42.5%
long_code_review	39.6	64.5	62.1	82.2%	26.7%

The single highest per-prompt throughput number in the entire bench: BeeLlama on code_python at 118.9 tok/s. Even at 65.8% acceptance, the wider draft batches push it well past ik_llama's 79.6 on the same prompt. The Python Fibonacci-with-memoization task is dense with predictable token patterns. Common keywords, common identifier names, common syntactic shapes. DFlash's wider drafts have plenty of correct guesses to grab.

BeeLlama also wins code_cpp at 93.3 vs ik_llama's 75.1, and stepwise_math at 89.0 vs 84.0. The pattern: when the model commits to a single direction with predictable surface forms (code, math step-by-step), DFlash's wider drafts pay off.

For coding agents that generate primarily Python or other structured outputs at length, BeeLlama is the highest-throughput backend on a single RTX 3090.

When ik_llama wins

The flip side. The prompts where BeeLlama's strategy backfires:

translation: ik_llama 68.0 vs BeeLlama 34.2. Nearly 2x faster on ik_llama. The translation prompt asks for the French rendering of "The quick brown fox jumps over the lazy dog" — a 22-token response. BeeLlama's DFlash adaptive controller can't warm up in 22 tokens; the per-cycle setup cost dominates. ik_llama's MTP heads activate immediately from prompt context and produce verified drafts on the first cycle. Acceptance for BeeLlama on translation came in at 10.6%, which is essentially "the drafter guessed almost nothing right."

summarize at 73.7 vs 65.8, **qa_factual** at 73.9 vs 59.0, **creative_short** at 69.6 vs 61.0, **long_code_review** at 64.5 vs 62.1. All four are short-or-medium prompts where ik_llama beats BeeLlama by 8-25%. The pattern: when output is short, ambiguous, or stylistically variable (creative writing, factual answers with multiple valid phrasings, the back-and-forth structure of code review), MTP's tight high-acceptance drafts win.

For mixed chat workloads — agents that produce short answers, structured-but-variable responses, casual conversation — ik_llama is the more consistent performer. Lowest tok/s across all 9 prompts: ik_llama at 64.5 (long_code_review). BeeLlama's lowest: 34.2 (translation). Per-prompt range for ik_llama is 19.5 tok/s; for BeeLlama it's 84.7 tok/s. BeeLlama's variance is more than 4x larger. That's the consistency story underneath the wall-clock tie.

The acceptance-rate trap

If you read the aggregate numbers cold — "ik_llama 88.5% accept, BeeLlama 37.4% accept" — your instinct is to call ik_llama the winner. The wall clock says it isn't. The two backends finish 0.62 seconds apart on a 22-23 second bench. By any reasonable measure, they tie.

The trap: acceptance rate is a per-draft ratio. Wall clock is total work. They correlate but don't have to track. Consider what actually happened:

- ik_llama generated 981 drafts, accepted 868. That's 868 useful drafted tokens.
- BeeLlama generated 2,809 drafts, accepted 1,050. That's 1,050 useful drafted tokens — more in absolute terms despite the lower ratio.

BeeLlama did more useful work despite lower acceptance because it drafted more in total. The wider batches compensate for lower per-token accuracy. The bottleneck for speculative decoding isn't acceptance rate alone — it's whether useful-drafted-tokens-per-second clears the overhead of drafting plus verifying. Both backends produce roughly equivalent useful-token throughput on this bench. Different strategies, same effective output rate.

The May 6 DFlash vs MTP head-to-head reported DFlash at 2.56x mean speedup vs MTP's 1.60x on similar-but-not-identical hardware. That bench used a different commit for both backends and a different ik_llama variant; the numbers aren't directly comparable to today's run. The May 22 result here, on the same RTX 3090 with both backends at their May 21 HEAD commits, is the cleanest 3-way I've measured.

When reading speculative-decoding benchmarks, especially ones authored by groups with a stake in one backend or another, watch for the acceptance-rate-vs-wall-clock framing. The number that matters to users is the one they feel: how long from prompt to all output complete. Acceptance rate is part of how you get there. It isn't the destination.

Methodology + caveats

Three caveats matter for interpreting this bench. Editorial honesty about each:

Each backend uses a different model variant. Mainline runs Unsloth UD-Q4_K_XL (17 GB). ik_llama runs Ubergarm's IQ4_KS-MTP (16 GB) — an ik_llama-exclusive quant with MTP heads baked into the GGUF. BeeLlama runs Q5_K_S target (18 GB) plus the 986 MB DFlash drafter from Ardenzard's repo. Each pairing is what the respective backend's community recommends for production. Apples-to-apples comparison would require running the same GGUF across all three, but that GGUF doesn't exist — the MTP and DFlash variants are non-portable formats. The right framing for this bench is "if you set up each backend canonically today, here's what you get." Useful for real users; not pure spec-decoding-method comparison.

ik_llama's MTP acceptance numbers came from the server log, not the harness output. ik_llama doesn't expose MTP draft statistics in its HTTP timings response — the `draft_n` field is empty across all 9 prompts. The actual per-prompt acceptance rates were extracted from the server log

post-run via grep on the `draft acceptance rate` lines. BeeLlama and mainline both expose draft stats through the response API. This is an ik_llama-specific telemetry quirk worth knowing about if you're building bench tooling against it. Filed for the project to flag.

One MTP warmup error fired during long_code_review on ik_llama. The server log surfaced "Source hidden state size mismatch (have 3665920 floats, need 2621440)" between prompts 8 and 9, followed by "failed to warm up MTP state from prompt batch." MTP still produced 82.2% acceptance on long_code_review (the prompt that triggered the warning), so the warmup failure didn't compromise the data. But the rough edge exists, and the longest prompt in the harness is where it surfaces. Worth noting for anyone planning to run very long prompts through ik_llama's current MTP path.

The 9-prompt harness is am17an's published bench script — same prompts, same `n_predict=192` cap, same `temperature=0.0`, same `seed=42` as the May 19 MTP piece and the broader community thread on PR #22673. The orchestration script that handles server-launch plus harness-run plus CSV-out lives at `~/Desktop/lucebox-hub/backend-shootout.sh` on my workstation. I'll publish it as a gist alongside this article.

The `--draft 3` choice for ik_llama matches the wall-clock winner from the May 19 piece. n=2 was faster on per-prompt throughput for 8 of 9 prompts in that earlier bench, but n=3 won the wall-clock total — that's the metric the user actually feels. BeeLlama's adaptive controller auto-picks n_max during the run with no equivalent dial to fix at launch.

Recommendation: pick by workload

The headline finding is that no single backend wins this bench outright. The right answer is contextual:

For coding agents and structured-output workflows → BeeLlama. The 118.9 tok/s peak on code_python is the fastest single number in the harness. The 93.3 tok/s on code_cpp is the third-fastest. If your workload is dominated by Python, C++, JSON, structured technical writing — code reviewers, agentic code generators, structured-data pipelines — BeeLlama's wider draft batches turn the predictability into speed.

For mixed chat workloads and short-response generation → ik_llama. The consistent ~74 tok/s mean with the lowest variance — every prompt landed between 64.5 and 84.0 — makes it the safer default. Chat agents that handle short factual answers, casual conversation, translations, and creative bursts will feel ik_llama's consistency more than they'll miss BeeLlama's peaks.

For everything else → mainline llama.cpp at ~40 tok/s. That's not slow. It's just not as fast as the spec-decoding paths. The May 8 mainline binary I tested predates PR #22673 and doesn't have MTP. If you're on a stock distro install (`apt install llama-cpp`) or a prebuilt release, you're getting roughly mainline-like numbers anyway. Spec decoding requires building from a fork or PR branch today.

The 24 GB VRAM target is real for this model size. If you're on a card with less VRAM, Qwen 3.6 27B at Q4-class quant doesn't fit comfortably. The Q4-class quants here land in the 16-18 GB range without KV cache, and the bench's 10K context plus the BeeLlama drafter pushes total usage up against the 24 GB wall. The Qwen 3.6 35B MoE path trades model architecture for VRAM headroom and is the better fit for smaller cards in this generation.

Reproducibility

Hardware: Miu — single RTX 3090 24GB, Linux (Xubuntu 24.04, kernel 6.8.0-111), NVIDIA driver 580.159.03, CUDA toolkit 12.8.93. Cards at similar compute tiers (RTX 4090, RTX A5000, AMD 7900 XTX with appropriate fork support) should reproduce broadly similar numbers. Exact wall clocks will vary with memory bandwidth and CUDA core count.

Backend repos and commits:

- ik_llama.cpp at HEAD `48a55f74` (May 21) — https://github.com/ikawrakow/ik_llama.cpp
- BeeLlama at HEAD `ba1fdce59` (May 21) — https://github.com/Anbeeld/beellama.cpp
- Mainline llama.cpp at `b9079-f9cd456ea` (May 8) — https://github.com/ggml-org/llama.cpp

Model files:

- Ubergarm IQ4_KS-MTP for ik_llama — https://huggingface.co/ubergarm/Qwen3.6-27B-GGUF
- Unsloth UD-Q4_K_XL for mainline — https://huggingface.co/unsloth/Qwen3.6-27B-GGUF
- BeeLlama Q5_K_S target — https://huggingface.co/unsloth/Qwen3.6-27B-GGUF
- DFlash 1.7B draft — https://huggingface.co/Ardenzard/Qwen3.6-27B-DFlash-GGUF

Bench harness (am17an's published script): https://gist.github.com/am17an/228edfb84ed082aa88e3865d6fa27090

Build pattern that worked for both ik_llama.cpp and BeeLlama on this system:

```
ulimit -s unlimited
cmake -B build -DGGML_CUDA=ON -DCMAKE_CUDA_ARCHITECTURES=86 \
```

```
  -DCMAKE_CUDA_COMPILER=/usr/local/cuda-12.8/bin/nvcc \
  -DGGML_CUDA_FA=ON -DGGML_CUDA_FA_ALL_QUANTS=ON   # BeeLlama only
cmake --build build --config Release -j1 --target llama-server llama-cli
```

The `-j1` matters. nvcc on the flash-attention template instances can segfault under higher parallelism even with plenty of RAM available — the issue is stack depth on deeply-templated CUDA code, not memory pressure. The `ulimit -s unlimited` is what made the difference. Skipping the test targets (`--target llama-server llama-cli`) avoids a gcc 13.3 internal compiler error on nlohmann/json template instantiation that the test suite hits.

What's next

Qwen 3.7 27B is coming — the preview version scored 57 AAI in May, and the open-weight 27B should land within weeks. When it does, the same bench needs to run again. Open question: does the MTP-baked-in-the-GGUF approach carry forward to 3.7's architecture, or does the GGUF format need updating? Open question: does Qwen 3.7's tokenizer or layer count shift the acceptance-rate-vs-wall-clock balance between MTP and DFlash?

PR #22673 (the mainline llama.cpp MTP path) hasn't merged yet but is closer than it was on May 19. ggerganov is actively reviewing on the `gg-mtp-rebase` branch. When the merge lands, mainline gets MTP for free and the 3-way shootout becomes a 4-way — mainline-baseline, mainline-MTP, ik_llama-MTP, BeeLlama-DFlash. That re-run will tell us whether ik_llama's MTP implementation is genuinely faster than mainline's once mainline catches up.

For now, the answer to "which 24GB backend should I run on RTX 3090 for Qwen 3.6 27B" is: BeeLlama for code-heavy workloads, ik_llama for mixed chat. Both finish in 22-23 seconds on the am17an harness. Mainline at 37 seconds is fine if you can't build from a fork. Pick by workload, not by acceptance rate.

Bench data, server logs, and the orchestration script are reproducible from this article. The am17an harness, model URLs, and backend HEAD commits above let any RTX 3090 owner run the same 27-cell bench in ~25 minutes wall time. If you do, drop the numbers — the PR #22673 thread and the BeeLlama issue tracker are where the maintainers are watching.

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/best-24gb-backend-shootout-ik-llama-beellama-llamacpp/

Free guides for running AI locally