Speculative Decoding: Free 20-50% Speed Boost for Local LLMs

February 23, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Speculative decoding pairs a small, fast 'draft' model with your main model. The draft model guesses the next 5-8 tokens, then the big model verifies them all in a single forward pass — which costs almost the same as generating one token. The output is mathematically identical to running the big model alone. You get 20-50% faster generation for about 1-2GB of extra VRAM.

📚 **More on this topic:** LM Studio Tips & Tricks · llama.cpp vs Ollama vs vLLM · Why Is My Local LLM Slow? · Planning Tool

Your 70B model generates great output. It just takes forever. You've already checked GPU offloading, picked the right quantization, and your model fits entirely in VRAM. There's nothing left to tune. Except there is.

Speculative decoding makes your big model generate 20-50% faster. No weight changes. No quality loss. No bigger GPU required. The output is mathematically identical to normal generation. Not "approximately the same." Identical. Bit-for-bit.

The cost is about 1-2GB of extra VRAM for a tiny draft model.

Here's how it works and how to set it up.

How speculative decoding works

Normal LLM generation is painfully sequential. Your 70B model generates one token, feeds it back in, generates the next token, feeds that back in, and so on. Each token requires a full forward pass through the entire model. On a 70B model at Q4, that's reading ~40GB of weights from VRAM for every single token. The GPU spends most of its time waiting on memory, not doing math.

This is the memory bandwidth bottleneck, and it's why bigger models are slower even when they fit in VRAM. The GPU cores have the compute capacity to do much more work per cycle, but they're starved for data.

Speculative decoding exploits this gap.

The proofreading analogy

Think of it like editing a document. You have a fast but sloppy intern (the draft model) and a meticulous senior editor (the target model).

Without speculative decoding: the senior editor writes every word herself, one at a time. Slow, but perfect.

With speculative decoding: the intern drafts 5 words ahead, then the senior editor reads all 5 at once and either approves them or corrects from the first mistake. Reading 5 words takes the editor roughly the same time as writing 1 word, because she can see them all in parallel.

That's the whole trick. **Verifying 5 tokens in parallel costs almost the same as generating 1 token.** The GPU reads the same ~40GB of weights either way — but with speculative decoding, it processes 5 candidate tokens in that single pass instead of 1.

The technical details

Here's what actually happens on each step:

1. **Draft phase:** A small model (say, 1B parameters) autoregressively generates K tokens (typically 5-8). This is fast because the draft model is tiny, reading ~1GB of weights per token instead of 40GB.

2. **Verify phase:** The big target model runs a single forward pass on the entire draft sequence. Because transformer attention is parallelizable across sequence positions, verifying K tokens costs barely more than generating 1 token. The model produces probability distributions for each position.

3. **Accept/reject:** Starting from the first drafted token, each draft token is compared against what the target model would have generated. If the draft token matches (using a mathematically precise rejection sampling scheme), it's accepted. If not, the target model's token replaces it, and all subsequent draft tokens are discarded.

4. **Repeat:** The process starts over from the last accepted position.

The rejection sampling is designed so that the final token distribution is exactly the same as if the target model had generated every token itself. This is a mathematical guarantee, proven in the original paper by Leviathan et al. (2023).

Why it’s free speed

The speedup comes from the ratio of work. Without speculative decoding, generating 5 tokens from a 70B model requires 5 forward passes reading ~40GB each, totaling 200GB of memory traffic. With speculative decoding, the draft model reads ~5GB (5 passes through a 1B model) and the target model reads ~40GB (1 verification pass). If the draft model guesses all 5 tokens correctly, you’ve produced 5 tokens for 45GB of memory traffic instead of 200GB.

In practice, the draft model doesn’t guess all tokens correctly. Typical acceptance rates are 60-80%, meaning out of 5 draft tokens, 3-4 get accepted. But even with partial acceptance, you’re generating ~4 tokens for the cost of what previously produced ~1.5. That’s where the 20-50% speedup comes from.

Acceptance rates and what affects them

The acceptance rate (how often the draft model’s guess matches what the target model would have generated) is the single biggest factor determining your speedup. Higher acceptance rate means more free tokens per verification pass.

What drives acceptance rate

Factor	Impact on Acceptance Rate
Same model family (draft + target)	High, 70-85% typical
Different model families	Lower, 50-65% typical
Predictable content (code, structured text)	Higher. Patterns are easier to guess
Creative/novel content	Lower. More surprising tokens
Higher temperature sampling	Lower. More randomness to predict
Lower temperature / greedy	Higher. More deterministic

The most important factor is using a draft model from the same family as the target. A Llama 1B draft model predicting for a Llama 70B target works better than a random 1B model because they share similar token distributions. They were trained on similar data with similar architectures. The small model learned similar patterns to the big one, just at lower fidelity.

Realistic acceptance rates

In practice, with well-matched model families, code generation hits 75-85% acceptance because code is highly structured: boilerplate, closing brackets, and common idioms are easy to predict. Long-form writing lands around 65-80% since narrative has momentum and the next word is often guessable. Technical Q&A sits at 60-75%, and creative fiction at high temperature drops to 50-65% because high entropy means more surprises. Short Q&A responses (55-70%) rarely benefit much because there aren't enough tokens for the speedup to compound.

Real-world speed benchmarks

These numbers come from testing on consumer hardware with well-matched draft/target pairs. Your mileage will vary based on GPU, quantization, and workload, but these give you the right ballpark.

RTX 3090 (24GB VRAM), long-form generation

Target Model	Draft Model	Without Spec. Dec.	With Spec. Dec.	Speedup
Llama 3.3 8B Q4_K_M	Llama 3.2 1B Q8	42 tok/s	56 tok/s	+33%
Qwen 2.5 14B Q4_K_M	Qwen 2.5 0.5B Q8	24 tok/s	34 tok/s	+42%
Qwen 2.5 32B Q4_K_M	Qwen 2.5 0.5B Q8	13 tok/s	18 tok/s	+38%

RTX 4090 (24GB VRAM), long-form generation

Target Model	Draft Model	Without Spec. Dec.	With Spec. Dec.	Speedup
Llama 3.3 8B Q4_K_M	Llama 3.2 1B Q8	75 tok/s	95 tok/s	+27%
Qwen 2.5 14B Q4_K_M	Qwen 2.5 0.5B Q8	45 tok/s	62 tok/s	+38%
Llama 3.3 70B Q4_K_M (partial offload)	Llama 3.2 1B Q8	8 tok/s	11 tok/s	+37%

Apple M4 Pro 48GB, long-form generation

Target Model	Draft Model	Without Spec. Dec.	With Spec. Dec.	Speedup
Qwen 2.5 32B Q4_K_M	Qwen 2.5 0.5B Q8	16 tok/s	23 tok/s	+44%
Llama 3.3 70B Q4_K_M	Llama 3.2 1B Q8	7 tok/s	10 tok/s	+43%

The pattern: **bigger, slower target models see bigger percentage gains.** A 70B model that generates 7 tok/s benefits more from speculative decoding than an 8B model at 75 tok/s, because the 70B model spends more time bandwidth-starved per token. The draft model's overhead is the same either way (it's tiny), but the savings per accepted token are larger when each target forward pass is expensive.

Best draft/target model pairings

Same-family pairings consistently outperform cross-family pairings. Here are the recommended combinations:

Target Model	Best Draft Model	Draft VRAM (Q8)	Notes
Llama 3.3 8B	Llama 3.2 1B	~1.1 GB	Strong pairing, high acceptance
Llama 3.3 70B	Llama 3.2 1B	~1.1 GB	Best budget pairing for big Llama
Qwen 2.5 7B	Qwen 2.5 0.5B	~0.6 GB	Tiny draft model, easy to fit
Qwen 2.5 14B/32B	Qwen 2.5 0.5B	~0.6 GB	Excellent acceptance rates
Qwen 3 8B/32B	Qwen 3 0.6B	~0.7 GB	Latest generation, great match
Phi-3 medium (14B)	Phi-3 mini (3.8B)	~4.0 GB	Larger draft model, higher VRAM cost
Mistral 7B	Mistral 0.5B	~0.6 GB	Good match within family

Keep the draft model in Q8 or full precision. Since draft models are tiny (0.5-1B), running them at Q8 adds minimal VRAM but preserves token distribution accuracy. A heavily quantized draft model has worse acceptance rates, which defeats the purpose. The 0.5B Qwen draft model at Q8 is only 600MB. There's no reason to quantize it further.

Cross-family pairings (less ideal)

You can use any small model as a draft for any target. A Qwen 0.5B drafting for a Llama 70B will still work (the math guarantees identical output either way). But acceptance rates drop to 50-60%, which means smaller speedups (10-25% instead of 30-50%). If you only have one small model downloaded and don't want to grab a same-family draft model, cross-family still helps. Just not as much.

VRAM cost

The extra VRAM for a draft model is trivial:

Draft Model	VRAM at Q8	VRAM at Q4
Qwen 2.5 0.5B	~0.6 GB	~0.4 GB
Llama 3.2 1B	~1.1 GB	~0.7 GB
Qwen 3 0.6B	~0.7 GB	~0.5 GB
Phi-3 mini 3.8B	~4.0 GB	~2.5 GB

For most pairings, you're adding 0.6-1.1GB. If your target model already fits in VRAM with a few GB to spare (and most well-configured setups do), the draft model slides right in. If you're already right at the VRAM limit with your target model, you might need to drop context length by 1-2K tokens to make room. A fair tradeoff for 30-40% faster generation.

Check our VRAM requirements guide to see exactly how much headroom you have.

Setting up speculative decoding in LM Studio

LM Studio has the easiest speculative decoding setup of any local AI tool. No command-line flags, no config files.

Step-by-step setup

1. **Download your target model.** For example, search for `Qwen2.5-14B-Instruct` in GGUF format, Q4_K_M quantization.

2. **Download a matching draft model.** Search for `Qwen2.5-0.5B-Instruct` in GGUF format, Q8_0 quantization. Same family as your target.
3. **Load the target model** in the chat tab as usual.
4. **Enable speculative decoding.** In the model settings sidebar (right panel), scroll to the **Speculative Decoding** section. Toggle it on.
5. **Select the draft model.** A dropdown appears. Pick the 0.5B model you downloaded.
6. **Set draft tokens.** The default (5) works well. You can experiment with 4-8. Higher values mean more aggressive speculation, with higher potential speedup but lower acceptance rate per batch.
7. **Start chatting.** LM Studio shows the token generation speed in the bottom bar. Compare it to your speed without speculative decoding enabled.

LM Studio CLI setup

If you prefer the LM Studio CLI:

```
# Load target model with speculative decoding
lms load qwen2.5-14b-instruct-q4_k_m \
  --speculative-model qwen2.5-0.5b-instruct-q8_0 \
  --draft-tokens 5
```

That's it. The API server at `localhost:1234` will automatically use speculative decoding for all requests.

Setting up speculative decoding in llama.cpp

llama.cpp supports speculative decoding through its `llama-speculative` binary and the `--draft` flag on `llama-server`. This requires more setup but lets you tune every parameter.

Using llama-server

```
# Start llama-server with speculative decoding
llama-server \
  -m ./models/qwen2.5-14b-instruct-q4_k_m.gguf \
  -md ./models/qwen2.5-0.5b-instruct-q8_0.gguf \
  --draft 8 \
  -ngl 99 \
  -c 4096 \
  --port 8080
```

Flags breakdown:

Flag	What It Does
`-m`	Path to the target (big) model
`-md`	Path to the draft (small) model
`--draft`	Number of tokens to draft per cycle (try 5-8)
`-ngl 99`	Offload all layers to GPU for both models
`-c 4096`	Context size

Using llama-cli for quick tests

```
# Interactive chat with speculative decoding
llama-cli \
  -m ./models/llama-3.3-70b-q4_k_m.gguf \
  -md ./models/llama-3.2-1b-q8_0.gguf \
  --draft 6 \
  -ngl 99 \
  -c 4096 \
  -p "Write a detailed explanation of how neural networks learn:"
```

Tuning the draft count

The `--draft` flag (also called `n_draft` or `num_speculative`) controls how many tokens the draft model guesses per cycle. The right value depends on your acceptance rate:

Draft Count	Best When	Tradeoff
3-4	Low acceptance rate tasks	Conservative, less wasted verification
5-6	General use	Good balance for most workloads

Draft Count	Best When	Tradeoff
7-8	High acceptance rate (code, same-family)	Aggressive, higher potential gain
10+	Rarely worth it	Diminishing returns, wasted drafts

Start at 5 or 6. If you're generating code with a same-family draft model, try 8. Going above 8 rarely helps because the probability that all 8+ tokens are correct drops fast.

Tool support matrix

Not every inference engine supports speculative decoding. Here's the current state:

Tool	Speculative Decoding Support	Setup Difficulty	Notes
LM Studio	Yes, built-in toggle	Easy	GUI dropdown, no flags needed
llama.cpp	Yes, `-md` flag	Medium	Full control, any GGUF model pair
vLLM	Yes, `speculative_model` param	Medium	Great for API serving, supports ngram speculation too
Ollama	Not yet	N/A	Built on llama.cpp but hasn't exposed the flag. Requested in GitHub issue #4643
ExLlamaV2	Yes, via `draft_model`	Medium	Works with EXL2 format draft models
Text Generation WebUI	Yes, via llama.cpp/ ExLlamaV2 backends	Medium	Depends on which backend you select

If you're using Ollama and want speculative decoding, your options are: switch to LM Studio (easiest), use llama-server directly (most control), or wait for Ollama to add native support. The underlying llama.cpp engine Ollama wraps fully supports it; Ollama just hasn't surfaced the configuration yet.

When speculative decoding actually helps

Speculative decoding doesn't help in every situation. Here's when to use it and when to skip it.

Best cases

Long-form generation (articles, stories, reports) is the sweet spot. Hundreds of tokens to generate, high acceptance rates on coherent text, and the full 30-50% speedup. Code generation is even better in some cases, with 35-50% speedup common because the draft model nails closing brackets, common idioms, and boilerplate. Large target models (32B-70B+) benefit the most because each forward pass is expensive, so every accepted draft token saves more wall-clock time. The draft model's cost stays constant regardless of target size. And longer chat conversations with 200+ tokens per reply give the speedup room to compound.

Worst cases

Don't bother with short Q&A responses under 20 tokens. By the time the draft model runs and verification happens, the overhead eats the savings. For single-sentence answers, speculative decoding can actually be slower. Small target models (1-3B) are also poor candidates: if the target is already running at 80+ tok/s, you're adding complexity for marginal gain, and the draft model might be half the size of the target anyway.

It also won't help if you're already GPU compute-bottlenecked (rare for local LLMs, which are usually memory-bandwidth-bottlenecked), or if your target model is heavily CPU-offloaded. Speculative decoding fixes memory bandwidth bottlenecks, not CPU-to-GPU data transfer. And at very high temperatures (above 1.5), the output is so random that the draft model's predictions miss more often than they hit. Acceptance rates below 40% mean negligible speedup.

The bottom line

If your target model is 7B+ and fully in VRAM, and your typical responses are longer than 50 tokens, just turn it on. The setup takes 2 minutes in LM Studio and costs ~1GB of VRAM. Check your tok/s before and after. If the number goes up (it will for most workloads), leave it on permanently.

Frequently asked questions

Does speculative decoding change the output?

No. The output is mathematically identical to generating without speculative decoding. The rejection sampling algorithm guarantees that the final token distribution matches the target model exactly. This is proven in the original paper and Google's independent implementation. If you run the same prompt with the same seed, with and without speculative decoding, you get the same tokens.

Does it affect prompt processing (prefill) speed?

No. Speculative decoding only helps during token generation (the autoregressive decode phase). Prompt processing, where the model reads your input, is already parallelized across all input tokens. Speculative decoding has nothing to add there.

Can I use a draft model from a different family?

Yes, but acceptance rates will be lower. A Qwen 0.5B drafting for Llama 70B still produces identical output to Llama 70B alone (the math guarantees this), but the draft model will guess wrong more often, so the speedup drops to 10-20% instead of 30-50%.

What about self-speculative decoding?

Some newer approaches skip the draft model entirely and use the target model's own early layers to make predictions. This avoids the extra VRAM cost but requires model-specific support. llama.cpp has experimental support for this. For most users, the standard two-model approach is simpler and well-supported.

Does it work with quantized models?

Yes. Both the target and draft model can be quantized. Use Q4_K_M or similar for the target (as you normally would) and Q8_0 for the draft. Since the draft model is tiny, Q8 costs almost nothing extra and preserves prediction quality.

Related guides

- LM Studio Tips & Tricks. Full guide to LM Studio's hidden features including the speculative decoding toggle.
- llama.cpp vs Ollama vs vLLM. Choosing the right backend for your hardware and use case.
- Why Is My Local LLM Slow?. Diagnostic flowchart for the other 90% of speed problems.
- VRAM Requirements for Local LLMs. Check if you have headroom for a draft model.

Source: https://insiderllm.com/guides/speculative-decoding-explained/

Free guides for running AI locally