1.5 Speculative Decoding

Chapter 1.3 established the bandwidth ceiling as the binding constraint on LLM decode: 136.5 GB/s shared LPDDR5X, ~25 GB/s effective NPU quota, ~6–20 tok/s sustained throughput for 3B–8B INT4 models. The natural follow-up question is whether there's any way around that ceiling without changing hardware. For one specific class of decode optimization, the answer is yes — and OpenVINO 2026.0 made it available on NPU. This section is about **speculative decoding**.

The pitch is straightforward and faintly magical: produce two to three tokens per forward pass instead of one, at no quality cost. The cost is paid in extra compute (which NPU has spare capacity for) and a second smaller model (the "draft"), both of which fit comfortably under the bandwidth budget that limits ordinary decode. If your workload is bandwidth-bound — and on Intel NPU it almost always is — speculative decoding is the single largest throughput win available without changing your model.

The Core Idea

Ordinary decode generates one token per forward pass. Each pass reads the entire model's weights (roughly 4 GB for an 8B INT4 model) and the full KV cache, producing one new token. Throughput is bounded by how fast you can stream those weights through the MAC array. At 25 GB/s effective bandwidth and 4 GB per pass, the math gives you ~6 tokens per second. The compute is mostly idle; the bus is the bottleneck.

Speculative decoding turns this on its head. Two models are involved: a **draft** model (fast, small, lower quality) and a **target** model (slow, large, the one you actually trust). The procedure:

1. The draft model generates `K` tokens autoregressively (typically K = 4–8). Each draft step is cheap because the draft model is small — maybe a 0.5B or 1B model against an 8B target.
2. The target model is fed the prompt plus all `K` proposed tokens at once. This is a **single prefill-style forward pass**, parallelized across the K positions, computing target-model probabilities for each.
3. The algorithm compares draft-model probabilities against target-model probabilities at each position. Tokens that match (or pass a probabilistic acceptance test) are kept. The first token that doesn't match is replaced with the target's choice; tokens after the rejection are discarded.
4. On average, the loop accepts 2–4 tokens per target forward pass. Net throughput is multiplied by that acceptance rate.

The key insight: **one target forward pass produces multiple tokens of output**, instead of one. The bandwidth cost per token effectively drops by the acceptance rate. On NPU, where

bandwidth is the binding constraint, that's a direct speedup.

Why It Doesn't Cost Quality

The acceptance test is mathematically constructed so that the output distribution is *identical* to ordinary target-model sampling. If the draft proposes a token with probability `p_draft(t)` and the target assigns it probability `p_target(t)`, the token is accepted with probability `min(1, p_target(t) / p_draft(t))`. If rejected, the replacement is drawn from `max(0, p_target(t) - p_draft(t)) / Z`. The math (Leviathan et al., 2022; Chen et al., 2023) works out such that the sequence of accepted tokens is distributed identically to a sequence drawn directly from the target.

For greedy decoding — which is what you use on NPU — the math simplifies further. Greedy means you always pick the argmax. The acceptance rule becomes: accept the draft token if and only if it matches the target's argmax at that position. No probability comparison, no acceptance probability less than 1; it's a deterministic exact-match check. The output is **bit-for-bit identical** to ordinary greedy decoding on the target. No quality cost whatsoever.

This is a strong claim and worth restating: **speculative decoding under greedy sampling produces exactly the same output as ordinary greedy decoding**, just faster. It is not an approximation. It is not a quality tradeoff. If you implement it correctly, the unit tests pass.

The Speedup Math

The expected throughput multiplier is roughly the **acceptance rate**, which depends on how well the draft model approximates the target.

For a typical setup — Llama 3.2 1B drafting for Llama 3.1 8B, same training family, same tokenizer — acceptance rates of 60–80% are normal. With K = 4 draft tokens per cycle, you get on average 2.4 to 3.2 accepted tokens per target forward pass. **That's a 2.4× to 3.2× decode speedup**, taking a 6 tok/s baseline to 14–19 tok/s.

The acceptance rate degrades when:

- **Architectures diverge.** A draft model from a different family (different training data, different attention layout, different vocabulary) won't share enough probability mass to match well.
- **The target's outputs are unpredictable.** Code generation tends to have lower acceptance rates than chat because individual tokens have higher entropy. Creative writing with temperature > 0 has lower acceptance than fact-recall.
- **The draft is too small.** A 0.1B draft against a 70B target won't share much distribution; the draft just isn't smart enough to predict the target's behavior. There's a sweet spot — draft size around 5–15% of target size tends to work well.

In numbers, here's the rough decode ladder you can expect on Lunar Lake NPU:

Configuration	Acceptance	Decode tok/s
Llama 3.1 8B INT4, ordinary decode	n/a	~6
Llama 3.1 8B INT4, Llama 3.2 1B draft, K=4	60–75%	~14–17
Llama 3.1 8B INT4, Llama 3.2 1B draft, K=8	60–70%*	~16–22
Llama 3.1 8B INT4, n-gram draft, K=4	30–50%	~9–12

*K = 8 doesn't double the speedup over K = 4 because rejection probability compounds — late draft tokens are more likely to be rejected, and a rejection invalidates everything after it. K = 4 is usually a sweet spot.

The n-gram draft is worth a note: instead of a small neural model, you use a statistical bigram or trigram model. Even cheaper than a small model, no second inference engine needed, but acceptance rates are 30–50% rather than 60–80%. For workloads where the prompt has high repetitive structure (code completion against an existing codebase, document continuation), n-gram drafts can punch above their weight.

OpenVINO 2026.0 NPU Support

Speculative decoding landed in OpenVINO 2026.0 with NPU as a target. The API is exposed through OpenVINO GenAI's `LLMPipeline` via a draft-model parameter:

```
import openvino_genai as ov_genai

# The fast draft model
draft = ov_genai.LLMPipeline(
    "models/llama-3.2-1b-int4_npu",
    device="NPU",
)

# The slow target model
target = ov_genai.LLMPipeline(
    "models/llama-3.1-8b-int4_npu",
    device="NPU",
    draft_model=draft,                # tie the draft to the target
    num_assistant_tokens=4,           # K = 4 draft tokens per cycle
```

```
)

# Generate as normal; speculative decoding is transparent
result = target.generate(
    "Explain how speculative decoding works.",
    max_new_tokens=200,
)
```

Both pipelines need to be compiled separately. Both consume NPU SRAM and pull from the bandwidth budget. The total compiled footprint is the sum of draft + target weights — for our example, 4 GB target + 0.5 GB draft = 4.5 GB, well within Lunar Lake's 16 GB system memory but worth budgeting for. The draft model's compile time is dominated by the same cold-start overhead as the target; expect 30–60 seconds the first time, then warm-load from `CACHE_DIR`.

The OpenVINO release notes describe NPU speculative decoding as available; what's not yet well-documented is which target/draft pairs Intel has validated for it. The safest bets are same-family pairs (Llama 3.x target + smaller Llama 3.x draft, Qwen3 target + smaller Qwen3 draft) where vocabulary and tokenization match exactly. Cross-family pairs may need vocabulary adapters that aren't standard.

What Doesn't Speed Up

Speculative decoding helps **decode**. It does not help **prefill**. The prefill phase is already a single parallel forward pass over the whole prompt; there's nothing to speculate about, because the entire input is known up front.

This matters because for short-output / long-prompt workloads — RAG over a large retrieved context, document summarization, long-context translation — prefill dominates the total latency. Speculative decoding leaves that wall in place. For agent workloads where decode is the majority of latency (chat-style outputs, code generation, long-form reasoning), it helps a lot. For workloads where prefill is the majority, it doesn't.

Speculative decoding also doesn't help **encoder-decoder seq2seq** like M2M-100 directly. The standard speculative decoding paper handles decoder-only autoregressive generation; extending it to seq2seq with cross-attention has been published but isn't in OpenVINO's NPU implementation yet. The `LLMPipeline` API doesn't accept seq2seq targets. If your worked example is M2M-100, you don't get speculative decoding's speedup at this point — the workaround is to switch to a decoder-only model for the same task (Qwen3 has decent translation quality) or wait for OpenVINO to add `Seq2SeqPipeline` speculative support.

The other workload it doesn't help: **batch size > 1**. Speculative decoding assumes a single inference stream; the draft model predicts what the target wants next for that one stream. Multi-stream serving requires different techniques (continuous batching, paged attention). On NPU where

batch size 1 is already the recommended pattern (Chapter 1.3), this isn't a constraint you feel — you were going to be single-stream anyway.

Picking a Draft Model

The rules of thumb:

Same family beats cross-family. Llama 3.2 1B drafts for Llama 3.1 8B well; it shares the vocabulary, the tokenization, and a lot of architectural inductive bias. Trying to use Llama as a draft for Qwen, or vice versa, requires vocabulary mapping and tends to give acceptance rates in the 20–40% range.

5–15% of target size is the sweet spot. Smaller than that and the draft is too dumb to predict the target. Larger than that and the draft costs nearly as much as the target, eating into the speedup. For an 8B target, a 0.5B to 1.2B draft is typical. For a 14B target, a 1.5B to 2B draft.

Distilled drafts beat opportunistic drafts. If your target is a custom model, ideally you train a distilled draft against it — minimize KL divergence between target and draft on a calibration corpus. That gets you the highest acceptance rates. In practice most teams use whatever same-family small model is available, which is good enough.

Quantize the draft as aggressively as the target. INT4-sym group-128 for both. Draft quality matters less than target quality; you can quantize the draft more aggressively if you want.

The draft's KV cache also competes for bandwidth. Two simultaneous KV caches in the SRAM allocation. For 8K context, this is real overhead; for 1–2K context, it's negligible.

Failure Modes

Things that go wrong with speculative decoding:

Acceptance rate collapses on out-of-distribution prompts. A draft model trained on English chat won't predict the target's behavior on, say, Korean technical writing. Acceptance drops to 20%, and the speculative pass actively costs more than ordinary decode (because you paid for the draft's forward passes and got nothing). Mitigation: monitor acceptance rate as a runtime metric; fall back to ordinary decode if it drops below ~40%.

Draft model and target model use different tokenizers. This will look like a successful compile and total garbage at inference. Always check tokenizer identity before pairing models.

Out-of-memory at compile. Two compiled models in SRAM is more than one. If the target was at the edge of fitting, adding a draft pushes it over. Mitigation: smaller draft, or move the draft to CPU (the draft is fast enough that CPU is plausible) — the OpenVINO API doesn't currently allow device-

split between draft and target in the same `LLMPipeline`, but it's a feature on the roadmap.

Latency spikes on rejection. If the draft consistently misses, the target's K-position forward pass is wasted compute and the latency for those K tokens is worse than ordinary decode would have been. The average is better; the variance is higher. For agents with strict per-token SLAs, this matters.

Combining With Other Techniques

Speculative decoding is orthogonal to almost everything else covered in the book:

- **Combine with INT4 quantization.** The target's bandwidth cost drops 4× from FP16 to INT4 *and* you decode multiple tokens per pass. Multiplicative wins.
- **Combine with prefix caching.** Different optimization, different code path. They cooperate.
- **Combine with chunked prefill on the target.** Speculative decoding affects only the decode phase; chunked prefill is the prefill-phase mitigation. Both apply.
- **Combine with cascade-triage routing.** A tiny model decides whether the heavy model needs to run; if so, the heavy model uses speculative decoding to run faster. Double speedup on the cold path.

The one thing it doesn't combine well with is *very aggressive quantization on the draft*. INT4 channel-wise drafts tend to mispredict the target's argmax more often than INT4 group-128 drafts, and the acceptance rate drop wipes out the bandwidth saving on the draft. Keep the draft at INT4 group-128 at worst.

Honest Status

Speculative decoding on Intel NPU is genuinely new — OpenVINO 2026.0 added it; the documentation is thinner than for the older LLM pipeline features; very few production deployments have published acceptance-rate data on Intel hardware. The numbers in this section's tables are extrapolated from the GPU-side speculative decoding literature and from cross-platform measurements, not directly measured on Lunar Lake NPU.

If you build something around speculative decoding on NPU and your measured numbers contradict the table, your numbers win. The underlying math is solid; the implementation maturity is not. Treat this section as the right architecture to reach for, and expect to do your own benchmarking before depending on specific speedup figures.

What This Section Bought You

You should now understand:

- **Speculative decoding** runs a fast draft model to propose K tokens, then verifies them in one target forward pass
- **Quality is preserved exactly** under greedy decoding — output is bit-identical to ordinary greedy
- **Typical speedups are 2–3×** for same-family draft/target pairs with K=4
- **OpenVINO 2026.0 added NPU support** through `LLMPipeline`'s `draft_model` parameter
- **Same-family pairs work best** — Llama drafts for Llama, Qwen for Qwen; cross-family acceptance rates collapse
- **Draft model size sweet spot is 5–15%** of target size; quantize the draft as aggressively as the target
- **Speculative decoding doesn't help prefill** — for long-prompt / short-output workloads, the gain is limited
- **Doesn't yet help seq2seq models** like M2M-100; decoder-only targets only
- **Combines well with INT4 quantization, prefix caching, chunked prefill, and cascade routing**
- **Monitor acceptance rate as a runtime metric** — fall back to ordinary decode if it collapses

Chapter 2 turns from hardware to software. Given the bandwidth ceiling, the quantization budget, and the speculative-decoding mitigation, how should the agent itself be structured to fit?

Previous: *1.4 The Accuracy Cost of Quantization* **Next:** *Chapter 2: Agent State & Decision-Making*

Revision #1
Created 2026-05-12 19:35:57 UTC by Admin
Updated 2026-05-12 19:35:57 UTC by Admin