# Foundations of NPU-Optimized Agents

NPU architecture and computational constraints. Model quantization and optimization for NPU deployment. Latency profiles and throughput optimization. Hardware-aware agent design patterns.

# 1.1 Understanding NPU Architecture

Before talking about agents on NPUs, we need to talk about the NPU itself — what makes it a distinct class of accelerator, and why the architectural choices ripple all the way up to how you design an agent loop. This book uses **Intel Core NPU** as its primary anchor and **Facebook AI's M2M-100** as its primary worked-example model. Every concept in this chapter ladders back to those two.

## What an NPU Actually Is

Neural Processing Units are domain-specific accelerators built for the matrix-multiplication and activation workloads that dominate neural network inference. They sit between CPUs (which are flexible but inefficient at dense matmul) and GPUs (which are powerful but power-hungry and latency-spiky). The NPU's pitch is **sustained matrix throughput at a fraction of the GPU's power budget** — useful for always-on, on-device workloads where battery and thermal headroom matter more than peak FLOPS.

The internal recipe varies by vendor, but every modern NPU combines three things: a **MAC array** (the dense-matmul workhorse, typically 1K–4K multiply-accumulate units per cycle), **on-chip SRAM** sized to hold a tile of activations and weights without round-tripping to DRAM, and a **fixed-function or near-fixed-function activation pipeline** (GELU, Sigmoid, Tanh, sometimes Softmax in hardware). What an NPU does *not* have, by design, is a general-purpose programmable shader array. Generality is the GPU's job.

## How the Major Families Compare

Five NPU families currently matter in commercial deployments, and they descend from recognizably different lineages:

**Intel Core NPU** is the only x86-native NPU and the focus of this book. It inherits its architecture from Movidius — Intel acquired the company in 2016 — and pairs a MAC array with programmable SHAVE VLIW DSPs in the same compute engine. The SHAVEs handle transcendentals, type conversion, and FP32 fallback. Three generations exist: NPU 3720 (Meteor Lake, December 2023, ~11.5 TOPS INT8 claimed but measured at 9.5 TOPS at 1.16 GHz by Chips and Cheese), NPU 4 (Lunar Lake, September 2024, 48 TOPS INT8, on the same compute tile as the CPU and Xe2 iGPU), and NPU 5 (Panther Lake, CES 2026, 50 TOPS INT8, Intel 18A process, with native FP8 support).

**Apple Neural Engine** is a fixed-function tensor accelerator tightly bound to Core ML on macOS and iOS. The M4 family ships a 16-core Neural Engine at 38 TOPS. Developer access is gated through Core ML — there's no equivalent of OpenVINO that lets you reach the silicon directly.

**Qualcomm Hexagon NPU** descends from a phone DSP (Hexagon QDSP6) with a bolted-on Tensor Accelerator and Vector eXtensions. Snapdragon X Elite reaches 45 TOPS. The architecture is fundamentally optimized for power efficiency at phone scale; bringing it to laptops is a relatively recent push.

**AMD XDNA** descends from Xilinx Versal AI Engine tiles arranged in a 2D spatial array. XDNA 2 in Ryzen AI 300 (Strix Point) hits 50 TOPS INT8 plus 50 TOPS Block FP16. Unlike Intel and Apple, XDNA sits as a separate IP block rather than on the main compute die — a different integration model with implications for memory contention.

**Google Edge TPU** is a fixed-function systolic-array ASIC, primarily for Coral devices and on-device TensorFlow Lite. It's a different deployment story (small embedded modules) and outside the scope of consumer-PC agents.

# What's Distinctive About Intel

Four things set Intel apart, and each has practical consequences for agent design:

**OS support spans Windows and Linux.** The in-tree `intel/linux-npu-driver` makes Intel NPU usable on Ubuntu and other Linux distributions without proprietary blobs in user space. Apple's ANE is macOS-only; Qualcomm's NPU is largely Windows-on-Arm. This matters when your agent's deployment target isn't a consumer laptop — embedded kiosks, industrial edge boxes, server racks running Linux all become viable on Intel NPU.

**Developer access is ungated.** Every Core Ultra Series 2 or Series 3 SKU exposes the NPU. OpenVINO is Apache-2.0 open source. There's no equivalent of needing a Mac to develop for ANE or a specific Snapdragon SKU to access Hexagon at full capability.

**Single-die integration on Lunar Lake and Panther Lake.** CPU, Xe2 (or Xe3 on Panther Lake) iGPU, and NPU all sit on the same compute die, sharing an 8 MB memory-side L4 cache on Lunar Lake. AMD's XDNA, by contrast, is a separate block. The integration matters because **agents that hop between devices** — say, NPU for prefill and iGPU for decode — pay less for the hop on a single-die SoC.

**OpenVINO ecosystem coverage.** OpenVINO is the only unified toolkit that targets CPU, iGPU, NPU, dGPU (Arc), and Gaudi from the same source intermediate representation, with native Hugging Face Optimum-Intel integration. No competing vendor offers this breadth.

# The Intel NPU Generation Table

The differences between NPU 3, NPU 4, and NPU 5 are large enough that "the Intel NPU" is not one target — it's three. Code that runs well on NPU 4 may fail to compile on NPU 3, and FP8 paths that work on NPU 5 won't exist on either predecessor.

| Generation | SoC / launch | NCEs | SHAVE DSPs | INT8 TOPS (claimed) | INT8 TOPS (measured) | Distinctive feature |
|---|---|---|---|---|---|---|
| **NPU 3720** | Meteor Lake, Dec 2023 | 2 | 4 (Movidius SHAVE) | ~11.5 | **9.5 @ 1.16 GHz** | First Intel NPU; INT8/FP16; 4 MB total scratchpad |
| **NPU 4** | Lunar Lake, Sep 2024 | 6 | 12 (SHAVE-V, 4× wider) | **48** | back-calc ~1.95 GHz | 4× MACs; NF4 weight compression; single-tile integration |
| **NPU 5** | Panther Lake, CES 2026 | 3 (each 2× wider) | not disclosed | **50** | — | Native FP8 (E4M3/E5M2); programmable LUT for activations; Intel 18A |

Per-engine, the MAC array is 2048 INT8 MAC/cycle on every generation. What changes is the count of engines, the SRAM, and the supported data types. NPU 3 totals 4,096 INT8 MAC/cycle; NPU 4 totals 12,288; NPU 5 consolidates back to roughly 12,288 with wider per-engine units and the same Intel-18A area efficiency win.

The Copilot+ certification line (≥40 TOPS) draws cleanly across the table: NPU 4 and NPU 5 qualify; NPU 3 doesn't. If your agent depends on Phi Silica or other Copilot+ OS features, your floor is Lunar Lake or later.

# The Hidden Constraint: Memory Bandwidth

TOPS is the marketing number. Bandwidth is the engineering number. **Lunar Lake ships LPDDR5X-8533 on a 128-bit on-package bus, yielding 8,533 MT/s × 128 bits / 8 = 136.5 GB/s of total platform bandwidth shared among CPU, iGPU, and NPU.** There is no private DRAM for the NPU and no published per-device bandwidth quota. This is the single most important number for understanding why LLM decode tops out where it does on Intel hardware.

Intel does not say "decode is DRAM-bandwidth-bound on NPU" in marketing copy — that specific phrasing is a gap in vendor literature. The closest official analog is **Microsoft's Phi Silica blog** (Windows Experience Blog, December 2024): "*Context processing involves intense parallel computation, mainly matrix multiplications, requiring high computational power. In contrast, the*

*token iteration stage demands substantial memory for storing and accessing the KV cache for each token generation step. While it needs less computation, efficient memory access is crucial.*" That's the canonical quotable framing. The roofline becomes tangible on DeepSeek-R1-Distill-Llama-8B INT4: 4 GB of weights streamed at 6.10 tok/s equals about 24.4 GB/s of sustained DRAM read, roughly 18% of platform peak. The NPU does not saturate LPDDR5X; it saturates its scheduling-quota share plus driver overhead.

We'll return to this ceiling in Chapter 1.3 (where it sets the ITL floor) and Chapter 2.1 (where it sets the KV cache wall).

# Why M2M-100 as the Worked Model

A book about agentic AI on NPUs needs a concrete model to keep referencing, and we'll use **Facebook AI's M2M-100** — specifically the 418M and 1.2B variants. M2M-100 is a 100-language many-to-many translation model released by Meta in 2020 with three properties that make it a useful teaching example:

It is **encoder-decoder seq2seq**, which forces us to confront the asymmetric NPU/CPU partition that the rest of the field is converging on — encoders fit NPU constraints well (static shape, single forward pass), decoders do not (dynamic shape, autoregressive). M2M-100 makes the partition visible in code, not just in theory.

It uses **full multi-head attention with no GQA or MQA** — the architectural choice that defines its KV cache footprint. In Chapter 2.1 we'll show that M2M-100 1.2B has the same per-token decoder KV bandwidth as Phi-3-mini-3.8B, because Phi-3 uses GQA with one-quarter the KV heads. The KV cache wall is set by attention design, not parameter count, and M2M-100 makes this visceral.

It is **MIT-licensed** (unlike its successor NLLB-200, which is CC-BY-NC 4.0 and unusable in commercial products). The licensing distinction matters more than the technical successor relationship.

It is **not on Intel's validated NPU model list.** This is a feature for our purposes. Production NPU deployment guides usually anchor on validated models (Llama, Phi, Qwen) where everything works. Real engineering happens at the edge of the validated set, and M2M-100 is squarely there.

The honest deployment recommendation, which we'll build up to, is encoder on NPU and decoder on CPU or iGPU. The mechanics of that split are the through-line of the book.

# What This Section Bought You

You should now understand:

- **An NPU is a domain-specific matmul accelerator** with on-chip SRAM and fixed-function activations — not a general-purpose shader array
- **Intel Core NPU is distinctive** in OS coverage, ungated developer access, single-die integration, and ecosystem breadth
- **There are three Intel NPU generations** (3720, 4, 5) with materially different capabilities — "the Intel NPU" is not one target
- **Memory bandwidth, not TOPS, is the binding constraint** on Lunar Lake's 136.5 GB/s LPDDR5X-8533
- **M2M-100 is the worked model** for the book — encoder-decoder with full MHA, MIT-licensed, deliberately at the edge of NPU validation

The next section moves from architecture to consequence: given these properties, what computational constraints fall out, and what does optimization look like for an encoder-decoder model on Intel NPU specifically?

---

**Next:** *1.2 Computational Constraints & Model Optimization*

# 1.2 Computational Constraints & Model Optimization

The architecture from Chapter 1.1 sets the rules. This section is about playing inside them: what an Intel NPU will and won't accept, how to shape a model so it compiles, and how to quantize without quietly losing the quality you paid for in training. We anchor on M2M-100 throughout — partly because translation is a clean worked example, partly because M2M-100 is unforgiving enough about NPU constraints to be instructive.

## The Static-Shape Mandate

The first rule of Intel NPU is that **shapes are largely set at compile time, not run time**. The compiler tiles your graph across the NCEs and SHAVE DSPs, computes the SRAM allocation, and generates a binary blob. Change the shapes and you compile a new blob — which takes seconds to tens of seconds, and which Windows Update or a driver upgrade may invalidate.

For non-LLM workloads this is an absolute constraint. The encoder of M2M-100 has to be `reshape`d to a fixed sequence length before compile:

```
encoder_model.reshape({"input_ids":      [1, 128],
                       "attention_mask": [1, 128]})
encoder_npu = core.compile_model(encoder_model, "NPU")
```

Any input shorter than 128 gets padded; anything longer either truncates or forces a new compile. Pick the sequence length once, pick it to cover your real workload's 95th percentile, and live with the padding waste on short inputs.

For LLM workloads the story has loosened. OpenVINO 2025.3 introduced **dynamic prompts on NPU by default** through the `LLMPipeline` static-shape pipeline with `PREFILL_HINT=DYNAMIC` and `NPUW_LLM_PREFILL_CHUNK_SIZE=1024`. This isn't dynamic shape in the GPU sense — it's *chunked* static prefill, where the compiler emits a fixed-shape kernel and the runtime feeds chunks until the prompt is consumed. The illusion of dynamism, paid for by a fixed chunk granularity. There's no equivalent for `OVModelForSeq2SeqLM`, which is exactly why M2M-100's decoder doesn't get the same flexibility as a Llama-3 decoder.

## Intel NPU Operator Coverage

The canonical list of supported operations lives at `docs.openvino.ai/<version>/about-openvino/compatibility-and-support/supported-operations.html`, version-stamped per release. **Encoder-friendly ops are mature.** Transformer encoders compile reliably: `MatMul`, `Add`, `Multiply`, `LayerNormalization` (with decomposed fallback when the fused op isn't supported), `Softmax`, `Gelu`, `Reshape`, `Transpose`, `Concat`, `Gather` with static indices, `ScaledDotProductAttention`, `Convert`, and the `FakeQuantize`/`FakeConvert` ops for INT8/FP8 paths. OpenVINO 2025.2 explicitly added **QKV-projection and Multi-Head Attention graph-level fusions** for encoder-based LLMs, which is exactly the kind of optimization M2M-100's encoder benefits from.

**Decoder pain points have specific names**, and each is worth recognizing because they appear in real error messages:

- `DetectionOutput` still fails to compile on NPU and iGPU as of OpenVINO 2025.4 (Intel Community thread 1735991, Feb 2026)
- `ScatterNDUpdate` has been rejected by the VPU/NPU compiler historically (issue #13594)
- INT64 indices in `Gather` and `ScatterND` routinely cause silent CPU fallback
- Variable-length `Gather`, dynamic `Slice`, dynamic `Reshape` in autoregressive decoders are the structural reason the whole model historically had to be static

When in doubt about whether your graph compiles, the answer is to try and read the compile log. The error messages are reasonably informative; the failures are usually localizable to a specific op.

# Quantization: PTQ, Not QAT

Post-training quantization (PTQ) is the default path on Intel NPU. Quantization-aware training (QAT) is technically supported by NNCF but rarely necessary — the PTQ recipes Intel has tuned for the validated NPU model list are good enough for most use cases, and they don't require retraining.

The path looks like this: export your PyTorch model to OpenVINO IR via Optimum-Intel, pick a quantization recipe (INT8 weight-only, INT4 channel-wise, INT4 group-wise, NF4 on Lunar Lake+, FP8 on Panther Lake+), and let NNCF do the work. The recipe matters because Intel NPU has strict constraints on which combinations work.

**The NPU LLM quantization rule from Intel's GenAI-on-NPU guide is unambiguous**: maximize the 4-bit weight ratio (`--ratio 1.0`), use `--group-size 128` for models up to ~4–5 B parameters, use `--group-size -1` (channel-wise) for larger models, and always use symmetric quantization (`--sym`). Asymmetric quantization is documented to crash the NPU LLM compile path.

The precision matrix by NPU generation:

| Mode | NPU 3 (MTL) | NPU 4 (LNL) | NPU 5 (PTL) |
|---|---|---|---|
| INT8-sym weights | ✅ | ✅ | ✅ |
| INT4-sym, group-size 128 | ✅ | ✅ | ✅ |

| Mode | NPU 3 (MTL) | NPU 4 (LNL) | NPU 5 (PTL) |
|---|---|---|---|
| INT4-sym, channel-wise | ✅ | ✅ | ✅ |
| NF4 (channel-wise only) | ✅ | ✅ | ✅ |
| NF4 weights + FP16 KV | ✅ | ✅ (2025.3+) | ✅ |
| FP8 (E4M3/E5M2) | ❌ | ❌ | ✅ |

The NF4 Lunar Lake exclusivity comes verbatim from OpenVINO's GenAI-on-NPU docs: "*The NF4 data type is only supported on Intel Core Ultra Processors Series 2 NPUs (formerly codenamed Lunar Lake) and beyond.*" The FP8 Panther Lake gating is documented in Intel's `openvino-ai-plugins-gimp` 3.2 release notes: "*FP8 model installation is now gated to NPU5000 and newer architectures.*"

# Exporting M2M-100

Here are the two `optimum-cli` invocations you'll actually use:

```
# INT8 weights, stateful with KV cache (the safe default)
optimum-cli export openvino \
  --model facebook/m2m100_418M \
  --task text2text-generation-with-past \
  --weight-format int8 \
  m2m100_418M_ov_int8

# INT4 group-wise, NPU-targeted
optimum-cli export openvino \
  --model facebook/m2m100_418M \
  --task text2text-generation-with-past \
  --weight-format int4 --sym --ratio 1.0 --group-size 128 \
  m2m100_418M_ov_int4_npu
```

Two pitfalls worth calling out before you spend an afternoon debugging them. **`--task translation` does not exist** in Optimum-Intel; it lives in `optimum-neuron` for AWS Neuron, which is a different toolkit. The correct task name for M2M-100 is `text2text-generation-with-past`. And **the `--with-past` suffix is required** for a stateful, KV-cached decoder; without it the export produces a stateless decoder that re-encodes the full target prefix on every step, which destroys decode throughput.

The output is a directory containing `openvino_encoder_model.xml`, `openvino_decoder_model.xml`, `openvino_decoder_with_past_model.xml`, and the tokenizer files. Three separate models, each independently compileable to a different device — which is exactly the lever we need for the hybrid execution pattern.

# Why M2M-100 Is Architecturally Expensive

Three reasons M2M-100 is harder to deploy on Intel NPU than a comparably-sized decoder-only model:

**Full multi-head attention with no GQA or MQA.** Look at `modeling_m2m_100.py` in HuggingFace Transformers: `self.k_proj` and `self.v_proj` both project to full `embed_dim`, and `num_heads == num_kv_heads`. The HF config has no `num_key_value_heads` field at all. A 1.2B-parameter M2M-100 decoder has the same per-token KV bandwidth as a 3.8B-parameter Phi-3-mini, because Phi-3 uses GQA with one-quarter the KV heads. We'll do the math in Chapter 2.1. The implication for NPU deployment: M2M-100's decode is bandwidth-bound at smaller parameter counts than modern models. No retrofit; switching to GQA would require retraining from scratch.

**Autoregressive decoder with dynamic sequence length.** The decoder generates one token at a time, with the KV cache growing on every step. The 2025.3 chunked-prefill feature relaxes this for decoder-only LLMs via `LLMPipeline`, but **no equivalent pipeline exists for `OVModelForSeq2SeqLM`**. OpenVINO 2026.0's NPU GenAI guide lists Whisper, LLM, and VLM pipelines only. M2M-100's decoder is on its own.

**Encoder-decoder cross-attention.** The decoder reads its own self-attention KV state *and* the encoder output every step, doubling the per-layer attention overhead relative to a decoder-only model. M2M-100's cross-attention KV cache is the same size as its self-attention KV cache for any given encoder length. This is the price of being a translation model — you keep the source sentence accessible throughout decoding — and there's no way to optimize it away.

The honest deployment recommendation that follows: **encoder on NPU** (single static prefill pass, ideal NPU fit), **decoder on CPU or iGPU** (dynamic autoregressive, where the runtime handles variable shapes well). Optimum-Intel does not expose per-component `device_map`, so this requires either subclassing `OVModelForSeq2SeqLM` or driving the IR files directly via `core.compile_model(...)`. Chapter 3.1 shows the code.

# The Sizing Heuristic, Specific to Intel

For Intel NPU specifically, the rough sizing budget is: **a model whose post-quantization weight memory fits in roughly 4–8 GB will run comfortably on Lunar Lake NPU 4**. The 16 GB Copilot+ minimum spec gives you the LPDDR5X room; the static-shape constraint sets compile complexity; the LPDDR5X bandwidth ceiling sets decode throughput.

In M2M-100 sizes:

| Variant | Params | FP16 weights | INT8 weights | INT4 weights | Fit |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| 418M | 418M | ~840 MB | ~420 MB | ~210 MB | Comfortable on NPU 3+ |
| 1.2B | 1.2B | ~2.4 GB | ~1.2 GB | ~600 MB | Comfortable on NPU 4+ |
| 12B | 12B | ~24 GB | ~12 GB | ~6 GB | Infeasible on consumer NPU at FP16; tight at INT4 |

The 12B variant essentially doesn't fit on consumer Lunar Lake outside of pathological configurations. The 418M and 1.2B variants are the realistic deployment targets.

# What This Section Bought You

You should now understand:

- **Static shapes are mandatory** for non-LLM workloads on Intel NPU; chunked prefill softens this for LLMs since 2025.3 but not for seq2seq
- **The Intel NPU operator coverage is encoder-friendly and decoder-fragile** — `DetectionOutput`, `ScatterNDUpdate`, INT64 indices, and dynamic Slice/Gather are recurring landmines
- **PTQ is the default path**; the NPU LLM quantization rule is `--sym --ratio 1.0` with group-size 128 (small) or -1 (large)
- **The precision matrix gates by generation**: NF4 needs Lunar Lake, FP8 needs Panther Lake
- **M2M-100 export goes through Optimum-Intel** with task `text2text-generation-with-past`; common mistakes are `--task translation` and missing `--with-past`
- **M2M-100 is architecturally expensive** for three structural reasons — full MHA, dynamic decode, cross-attention — none of which is fixable in post-training
- **The hybrid pattern is encoder-on-NPU, decoder-on-CPU/iGPU**, and the rest of the book builds on it

The next section turns to performance: given a model that compiles cleanly, what does its latency profile actually look like on Intel hardware, and what does that imply for agent design patterns?

---

**Previous:** *1.1 Understanding NPU Architecture* **Next:** *1.3 Latency, Throughput, and Hardware-Aware Patterns*

# 1.3 Latency, Throughput, and Hardware-Aware Patterns

The architecture and constraints from Chapters 1.1 and 1.2 set the ceiling. This section is about measuring it: what does a real model's latency profile look like on Intel hardware, how does that latency break down, and what does that imply for the agent loop design patterns Chapter 2 will develop?

We use two published benchmarks as anchors: **Llama 2 7B on MLPerf Client v0.6**, measured by Intel on a Core Ultra Series 1 processor, and **DeepSeek-R1-Distill-Llama-8B INT4 on OpenVINO Model Hub**, both real data points that set the floor and ceiling for what you can expect.

# The Two Key Latency Metrics: TTFT and ITL

Model inference latency on accelerators is traditionally quoted as a single number (e.g., "inference takes 50 ms"). That's been obsolete for over a decade in LLM contexts because LLMs have two phases with radically different characteristics.

**Time-To-First-Token (TTFT)** is the latency of the prefill phase: the time from when you send the prompt to when the model emits the first output token. The prompt is static, potentially long (hundreds of tokens), and the entire computation is on the critical path — you can't generate a second token until the first one exists. TTFT is compute-bound.

**Inter-Token Latency (ITL)** is the latency of each subsequent token in the decode phase. The decoder sees only the new token slot plus the KV cache, and the computation is roughly constant per new token. ITL is memory-bandwidth-bound on NPU.

On Intel Core Ultra with Lunar Lake, the published benchmarks nail this split:

**Llama 2 7B on MLPerf Client v0.6** (Intel internal, Core Ultra Series 1 Meteor Lake):

- TTFT at 128 input tokens: **1.09 seconds**
- ITL (tokens 2+): **~54 ms/token**
- Implied throughput: **18.55 tok/s** sustained

**DeepSeek-R1-Distill-Llama-8B INT4 on OpenVINO Model Hub** (public benchmark, Intel NUC 14 Pro with Lunar Lake):

- Measured at **6.10 tok/s sustained**, which is **~163 ms/token ITL**
- TTFT is not published; extrapolate from the 8B size and INT4 quantization

The 2.8× gap between Llama 2 (18.55 tok/s) and DeepSeek-Distill-8B (6.10 tok/s) is real. A naive explanation is parameter count: 7B vs 8B is 14% more matmul. But the gap is closer to 3×, not 14%, which means something structural is different. The honest answer: these are measured on different hardware revisions (Series 1 Meteor Lake vs Series 2 Lunar Lake is a 4× MACs gain), different quantization targets (Llama 2 at FP16? INT8?), and different workload assumptions (batch size, prompt length). The benchmarks are not apples-to-apples; treat them as reference ranges.

# The Roofline: Hardware Limits

The sustainable throughput on Intel NPU is bounded by the LPDDR5X bandwidth ceiling from Chapter 1.1: **136.5 GB/s platform-wide shared among CPU, iGPU, and NPU**. No device gets the full 136.5 GB/s; the actual per-device quota depends on driver scheduling and competing loads.

For an 8B INT4 model:

- **Weight memory**: 4 GB (8B params × 4 bits/param / 8)
- **Sustained throughput**: 6.10 tok/s (from the published benchmark)
- **DRAM read rate**: 4 GB × 6.10 tok/s = **24.4 GB/s**

This is roughly **18% of platform peak bandwidth**. The NPU is not starving, but it's not saturating the bus either. The gap between 24.4 GB/s and 136.5 GB/s is scheduling overhead, driver latency, and contention from other agents on the SoC (CPU, iGPU). The roofline model says: if you could eliminate all contention and overhead, you'd hit bandwidth saturation at roughly **(136.5 GB/s) / (4 GB model weight) = 34 tok/s** — about 5.5× higher than what's measured. That gap is real and structural.

The practical implication: **you cannot expect sustained decode speeds above 15–20 tok/s on Lunar Lake NPU for reasonable 8B models**. Going faster requires either a smaller model, lower precision (NF4, FP8 on NPU 5), or moving decode to the iGPU.

# Comparing to iGPU

The same Core Ultra platform has an Xe2 iGPU (Lunar Lake) or Xe1 iGPU (Meteor Lake). The iGPU is not on the same 136.5 GB/s bandwidth constraint as the NPU — it has its own path to VRAM — and it's substantially faster for decode workloads.

On the same hardware (Core Ultra Series 2), Llama 2 7B typically reaches **~40 tok/s on iGPU** (measured by community benchmarks; Intel does not publish iGPU LLM numbers). That's a **2.1× speedup over NPU** for decode. For prefill (TTFT), the gap is wider: iGPU TTFT is typically **300–400 ms** for a 128-token prompt, vs **1.09 seconds on NPU** — a 3–4× gap.

The hybrid story emerges: if you can split the workload with prefill on NPU and decode on iGPU, you get 2.1× throughput for the large constant-cost phase (decode) and take the NPU's hit only on the one-time prefill. Chapter 3.1 builds the code for this pattern.

# What Phi Silica Tells Us

Microsoft's Phi Silica is the closest public reference architecture for an NPU-targeted LLM, deployed on Snapdragon X (Qualcomm NPU, not Intel). The published numbers are **TTFT 230 ms, 20 tok/s sustained** on a 2K context window. The architecture is: **CPU tokenizer + embedding + LM-head, NPU transformer blocks, CPU decode with N=64 KV sliding window**.

This is instructive not because Snapdragon X hardware maps cleanly to Intel NPU (it doesn't), but because it shows what real deployed decisions look like: **encoder on accelerator, decoder split between accelerator and CPU**, because the decode phase's structure (lots of memory, little compute per token) is where the accelerator's architecture breaks down.

Phi Silica also exposes the **sliding-window KV cache technique**: instead of keeping the full context KV in memory, keep only the most recent N tokens (here N=64). This trades recompute (re-running attention over discarded context) for memory bandwidth. For NPU where bandwidth is the constraint, this trade-off wins. The Llama 2 and DeepSeek-Distill benchmarks above use full KV caches. If they switched to sliding-window N=128, ITL would drop materially, but context awareness would degrade after 128 tokens. This is a tuning knob for the agent's working memory size.

# Architecture-Specific Wisdom

Three things deserve to be nailed down because they're easy to get wrong:

**Batching doesn't help on NPU for decode.** On GPU, you can batch multiple independent decode streams and keep the compute pipeline full — token 1 from user A, token 1 from user B, token 1 from user C, all in parallel. On NPU with a fixed-shape pipeline and 136.5 GB/s bandwidth ceiling, batching adds more weight reads without adding more available bandwidth. Batching increases *latency* (because you're now serving multiple users sequentially) without increasing *throughput* (because you hit the bandwidth ceiling with a single-user stream). The practical result: **always use batch size 1 for decode on Intel NPU**.

**LPDDR5X speed is shared, not divisible.** The 136.5 GB/s includes all traffic: CPU instruction fetches, iGPU reads, NPU reads, system memory traffic. If the CPU is running code and the iGPU is running a concurrent task, the NPU's available bandwidth drops. If you want predictable NPU performance, you need to account for potential contention. The Phi Silica sliding-window approach partly exists to reduce bandwidth hunger, reducing contention sensitivity.

**Compile-time overhead is real.** The first invocation of a compiled model on NPU takes 30–60 seconds (from Chapter 1.2 cold-start benchmarks). Subsequent invocations take <3 seconds (warm start, cached to disk via `CACHE_DIR`). This cost is amortized over the model's lifetime in production, but for development and short-running agents, it's a gotcha. Always set `CACHE_DIR` to a persistent location; otherwise you pay the cold-start penalty on every process restart.

# The Agent-Loop Latency Budget

A 5-step agent loop — where the agent reasons, takes an action, observes the result, and repeats — looks like this in latency terms:

- **Step 1 prefill**: 512-token accumulated prompt, ~4 seconds TTFT
- **Step 1 decode**: 64 output tokens (the agent's "Thought / Action / Observation"), ~10.4 seconds ITL
- **Steps 2–5**: same pattern, context grows each iteration
- **Total**: ~70–75 seconds for 5 steps at this prompt size

On iGPU: ~35 seconds.

This is the roofline for agent patterns on Intel NPU, and it's the why behind the Chapter 2 reasoning-architecture recommendations: ReAct (which is inherently loopy) doesn't fit the latency budget, but single-shot and cascade patterns do.

# What This Section Bought You

You should now understand:

- **TTFT and ITL are two distinct metrics** with different hardware bottlenecks — compute vs bandwidth
- **Published benchmarks**: Llama 2 7B at 18.55 tok/s (TTFT 1.09s), DeepSeek-Distill-8B at 6.10 tok/s
- **The roofline ceiling is 136.5 GB/s LPDDR5X** shared across CPU/iGPU/NPU, yielding ~34 tok/s theoretical max for 8B INT4
- **iGPU is 2.1× faster than NPU for decode**, making hybrid prefill-on-NPU / decode-on-iGPU the natural pattern
- **Phi Silica shows real deployment wisdom**: CPU encoder/decoder, NPU transformer, sliding-window KV for bandwidth
- **Batch size 1 for decode on NPU**; batching doesn't increase throughput, only latency
- **Compile-time overhead is 30–60s cold, <3s warm**; set `CACHE_DIR` always
- **A 5-step ReAct loop takes ~70–75 seconds on NPU**, which is the structural reason Chapter 2 recommends single-shot or cascade patterns

Chapter 2 now turns from hardware to software: given these latency budgets, how does model state (KV cache, attention memory) factor into design, and what reasoning architectures actually work within constraint?

---

**Previous:** *1.2 Computational Constraints & Model Optimization* **Next:** *Chapter 2: Agent State & Decision-Making*

# 1.4 The Accuracy Cost of Quantization

Chapter 1.2 laid out the quantization recipes Intel NPU supports: INT8-sym, INT4-sym group-128 or channel-wise, NF4 on Lunar Lake, FP8 on Panther Lake. The hardware story ended there. This section is the missing other half — what those recipes actually cost you in model quality, how to measure it, and when the cost stops being acceptable.

It matters because the gap between "this compiles and runs" and "this works for users" is wider than the OpenVINO docs suggest. Quantization is never free. You're trading bits of weight precision for bandwidth and memory headroom, and somewhere on the spectrum from FP16 down to INT4 the model starts being meaningfully worse at the thing you're paying it to do. The job is to find where that line is for your specific workload, not to assume Intel's validated recipes are quality-validated for every task.

# Why Quantization Isn't Free

A 4-bit weight has 16 distinct values. The full-precision FP16 weight it replaces has roughly 65,000. Group-wise quantization mitigates this by sharing a scale factor across 128 weights — so the effective dynamic range per group is closer to FP16's, but every weight in the group still has to round to one of 16 levels relative to that scale. Channel-wise quantization is more aggressive: one scale per output channel, hundreds or thousands of weights sharing it.

For weights with a broad distribution and a few outliers (which is what most transformer layers have), this rounding error compounds layer by layer. By the time a token has traversed 24 transformer blocks, the accumulated drift is large enough to change which next-token probability wins. Sometimes that's invisible — the model still says something reasonable. Sometimes it cascades into hallucinated facts, broken JSON, or a translation that means something subtly different from what the source said.

The intuition that matters: **decode is more sensitive to quantization than prefill**. Prefill processes the entire prompt in one parallel pass; small errors get averaged across many tokens. Decode generates one token at a time, with each new token's logits depending on every prior step's KV state. An error introduced at step 5 propagates through step 6, 7, 8 — and the model can't "self-correct" because greedy decoding (the only mode on NPU `LLMPipeline`) commits to whichever token won the contaminated argmax. This is why aggressive quantization tends to break agents specifically: agents do long decodes, and the drift accumulates.

# The Standard Intrinsic Metric: Perplexity

Perplexity measures how surprised a model is by held-out text. Lower is better. A model that quantizes well sees its perplexity rise by 1–3% relative to the FP16 baseline; a model that quantizes badly sees 10%+ rises and tangible output degradation.

Perplexity is computed on a fixed corpus (WikiText-2 is conventional) by running the model forward and averaging the negative log-likelihood of each token in context. The mechanics:

```
# Sketch — use lm-evaluation-harness in practice
import math
from transformers import AutoTokenizer
import openvino_genai as ov_genai

tokenizer = AutoTokenizer.from_pretrained("path/to/model")
pipe = ov_genai.LLMPipeline("path/to/openvino_ir", "NPU")

total_nll = 0.0
total_tokens = 0
for text in wikitext_test_corpus:
    tokens = tokenizer.encode(text)
    # Run the model in teacher-forced mode; sum -log P(token_i | tokens_<i)
    nll, n = score_sequence(pipe, tokens)  # implementation-specific
    total_nll += nll
    total_tokens += n

ppl = math.exp(total_nll / total_tokens)
```

In practice you use `lm-evaluation-harness` (`lm-eval --tasks wikitext`) against the OpenVINO model and against the FP16 baseline; the comparison is what tells you anything. A standalone perplexity of 7.2 means nothing without context. A perplexity that jumped from 6.8 (FP16) to 7.5 (INT4-sym group-128) means a 10% increase — borderline acceptable for chat, probably noticeable for translation.

Perplexity is a useful canary, not a verdict. A model can have stable perplexity but fail on the specific structured-output task your agent depends on. Always pair it with task-level evaluation.

# Downstream Task Evaluation

The benchmarks that matter for agents:

- **MMLU** (Massive Multitask Language Understanding) — multiple-choice across 57 academic subjects. Tests knowledge retention; sensitive to quantization in mid-range models (3B–8B). Expect 1–3 percentage points of accuracy loss at INT4-sym group-128.
- **IFEval** — instruction-following evaluation. Measures whether the model obeys constraints like "respond in JSON" or "answer in exactly three sentences." **This is the test most directly relevant to agent reliability.** Quantization-sensitive in a non-linear way: models often pass at INT8, scrape by at INT4 group-128, fail at INT4 channel-wise.
- **HumanEval** / **MBPP** — Python code generation. If your agent writes code, run these. Aggressive quantization tends to break syntax (extra parentheses, missing colons) before it breaks logic.
- **MT-Bench** — multi-turn conversation quality with LLM-as-judge scoring. Most expensive to run; most representative of chat use cases.
- **BLEU / chrF / COMET for translation** — if you're following the M2M-100 thread of this book, these are the metrics that matter. Quantization can drop BLEU by 1–2 points on common language pairs and 3–5 points on low-resource pairs.

Run every benchmark twice: once against the FP16 baseline you're shipping with, once against the quantized OpenVINO export. Report the delta, not the absolute number. The delta is what tells you whether the quantization recipe is safe.

# Empirical Findings, Roughly

The book can't promise specific numbers for every model — they vary too much — but the shape of the curve generalizes. For decoder-only LLMs in the 3B–8B range:

| Recipe | Perplexity delta vs FP16 | MMLU delta | IFEval delta | Typical verdict |
|---|---|---|---|---|
| **INT8 weight-only sym** | +0.5 to +1.5% | -0 to -1 pp | -0 to -1 pp | Safe; ship it |
| **INT4-sym group-128** | +2 to +5% | -1 to -3 pp | -1 to -4 pp | Usually acceptable; verify on your task |
| **INT4-sym channel-wise** | +5 to +10% | -2 to -5 pp | -3 to -8 pp | Acceptable only for large models (>10B) |
| **NF4 channel-wise** | +2 to +4% | -1 to -2 pp | -1 to -3 pp | Close to INT8 quality; Lunar Lake+ only |
| **FP8 (E4M3)** | +0.5 to +1.5% | -0 to -1 pp | -0 to -1 pp | Close to FP16 quality; Panther Lake+ only |

The values are illustrative ranges. Your model will land somewhere in them, and where it lands depends on training data, model family, and the specifics of NNCF's calibration. Don't quote this table to anyone as if it were measured on their model. It isn't.

For **encoder-decoder seq2seq models** like M2M-100, the numbers tend to be worse than decoder-only at the same recipe. Two reasons. First, the encoder and decoder accumulate errors independently, and the cross-attention exposes both. Second, translation is a "every token matters" task: a mistranslated noun isn't a soft degradation, it's a wrong answer. Where decoder-only LLMs can lose 2 MMLU points without anyone noticing, a seq2seq translator that loses 2 BLEU is detectably worse to a user. Test more aggressively; consider INT8 weights as your floor rather than INT4 if quality matters.

# The Specific Failure Modes for Agents

Agents fail at quantization in characteristic ways:

**Structured output breaks first.** The model that produced valid JSON at FP16 starts emitting trailing commas, unbalanced braces, or extra commentary outside the fence. This is because JSON is a low-probability tail of the model's output distribution; small logit perturbations are enough to push it into the higher-probability "natural language" basin. Mitigation: constrained decoding (Chapter 3.4), or step back to a less aggressive quantization.

**Long-decode coherence degrades.** The first 50 tokens of an output look fine; tokens 200–500 drift into repetition, off-topic content, or factual confusion. The drift is the KV-cache-accumulating-quantization-error effect. Mitigation: tighter context windows, shorter task decomposition, or larger model at less aggressive quantization.

**Instruction following weakens.** "Answer in exactly three sentences" becomes "answer in roughly three or four sentences." "Output only the translated text" becomes "Here is the translation: [translated text]." This shows up as IFEval drops and is one of the most common silent failures.

**Multi-step reasoning hallucinates more.** Chain-of-thought arguments stay grammatical but become factually wrong, or arrive at correct conclusions via incorrect intermediate steps. Particularly bad for plan-then-execute agents where the plan depends on accurate reasoning at step 1.

**Rare-token tasks break.** Anything involving uncommon vocabulary (medical terms, proper nouns, code identifiers) degrades faster than common-prose tasks. The 4-bit weight space simply doesn't have the precision to discriminate among rare-token logits as cleanly as FP16 does.

# Why Asymmetric Quantization Crashes the NPU LLM Path

Chapter 1.2 mentioned the rule — `--sym --ratio 1.0` for NPU LLMs, asymmetric quantization crashes the compile path — without explaining why. The reason is worth understanding because it

tells you something about what the NPU compiler is doing.

Symmetric quantization represents the weight range as `[-scale × 7, +scale × 7]` for 4-bit (or analogous ranges for other bit widths); the zero-point is fixed at 0 and not encoded per-weight. Asymmetric quantization adds a per-tensor or per-group zero-point shift: `[-scale × 7 + zp, +scale × 7 + zp]`, encoding both scale and zero-point.

The NPU compiler's matmul kernels assume the symmetric layout. The MAC arrays are wired to multiply against a fixed offset of zero. Asymmetric weights would require a per-input-channel adjustment that doesn't exist in the kernel templates Intel ships. The result isn't a quality degradation; it's a compile failure, often with an opaque error message about kernel selection.

This is changing — Intel has talked about asymmetric INT4 support in future releases — but as of OpenVINO 2026.1, sticking to symmetric is mandatory for LLM workloads on NPU.

# When Quantization Is Breaking Your Model

Specific signs to watch for:

**Validation perplexity rises >5%.** Stop and reconsider. Either the recipe is too aggressive, the calibration dataset doesn't match your domain, or there's a layer that quantizes badly and needs to be excluded.

**Greedy decoding produces obviously different output than the FP16 baseline on the same prompt.** Greedy is deterministic; if FP16 and INT4 give different answers, the rounding has shifted the argmax somewhere material. Run the same prompt 5–10 times; pattern-match the differences.

**Specific tokens become impossible.** Hashes, code identifiers, low-frequency vocabulary words may simply never be selected because their FP16 logit advantage gets rounded away. Easy to detect by checking the top-10 candidate tokens at known critical positions.

**Outputs get longer or shorter systematically.** The end-of-sequence token's logit shifts under quantization. If outputs are systematically truncated, the EOS token is being selected too early; if they ramble past the natural endpoint, it's being selected too late.

# NNCF's Accuracy-Aware Workflow

When you have a quality budget and a calibration dataset, NNCF offers a workflow that automates layer-by-layer precision selection. The mechanics, roughly:

1. Run the FP16 model on your calibration dataset; record per-layer activation statistics.
2. Try quantizing each layer in isolation; measure perplexity / task accuracy delta.
3. Identify which layers contribute disproportionately to quality loss.
4. Apply less aggressive precision (or skip quantization) on the problematic layers; keep the rest aggressive.
5. Iterate until quality is within budget.

In Optimum-Intel:

```
# Sketch — actual flags vary by version
optimum-cli export openvino \
  --model facebook/m2m100_1.2B \
  --task text2text-generation-with-past \
  --weight-format int4 --sym --ratio 0.8 --group-size 128 \
  --quantization-config '{"sensitivity_metric": "weight_quantization_error"}' \
  m2m100_1.2B_ov_int4_mixed
```

The `--ratio 0.8` says "quantize 80% of layers to INT4; keep the most sensitive 20% at INT8." NNCF picks the layers based on the sensitivity metric. This is a real lever for tuning quality-vs-size when uniform quantization is too aggressive. It also costs you some compile complexity and a longer export step.

# What This Section Bought You

You should now understand:

- **Quantization isn't free** — somewhere on the precision spectrum, model quality starts degrading materially
- **Decode is more quantization-sensitive than prefill** — agents (which do long decodes) feel quantization more than one-shot inference
- **Perplexity is a canary, not a verdict** — pair it with task-level benchmarks (MMLU, IFEval, HumanEval, MT-Bench, BLEU)
- **Empirical ranges**: INT8 ~1% degradation, INT4 group-128 ~3% on decoder-only LLMs; encoder-decoder is worse
- **Agent failure modes**: structured output breaks first, long-decode coherence drifts, instruction following weakens, rare tokens vanish
- **Asymmetric quantization crashes the NPU LLM compile path** because the MAC kernels assume symmetric layout
- **NNCF accuracy-aware workflow** lets you mix precisions per layer when uniform quantization hurts too much
- **Always compare to your FP16 baseline** — absolute quantized numbers mean nothing in isolation

The next section steps back to the bandwidth ceiling we established and asks whether there's any way to dodge it — specifically, whether speculative decoding (now available on NPU) can buy us back the throughput the LPDDR5X bus refuses to give.

---

**Previous:** *1.3 Latency, Throughput, and Hardware-Aware Patterns* **Next:** *1.5 Speculative Decoding*

# 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*