Real-World Case Studies & Best Practices

Building customer-facing NPU agents (chatbots, assistants). Batch vs. streaming inference strategies. Handling fallbacks and graceful degradation. Lessons learned and anti-patterns to avoid.

5.1 What's Actually Shipping on Intel NPUs

The most useful thing a book like this can do, in its closing chapter, is be honest about what is *really* deployed on NPU hardware today versus what is announced, planned, or aspirational. The gap matters. If you build your roadmap on press releases, you'll discover too late that the workload you assumed worked doesn't. This section surveys publicly documented NPU deployments, calls out what's measured versus marketed, and identifies the patterns that recur across them.

Microsoft Copilot+ PC: The Reference Deployment

The most public NPU case study is Microsoft's **Copilot+ PC** program. Certification requires ≥40 TOPS NPU, 16 GB RAM, 256 GB SSD, and Windows 11 24H2 or newer. On Intel silicon, **only Core Ultra 200V (Lunar Lake) qualifies** — Meteor Lake and Arrow Lake-S do not. This is the first hard truth about NPU agent deployment: marketing covers many SKUs, but feature certification covers very few.

Features confirmed to run on the NPU on Copilot+ PCs include Windows Recall, Live Captions with Translation (40+ languages), Studio Effects (Background Blur, Eye Contact, Auto-framing, Voice Focus, Portrait Light), Cocreator in Paint, Restyle in Photos, Phi Silica, and Click to Do (which is hybrid NPU+cloud). Note what's *not* on this list: most third-party agents.

Phi Silica: The Gold Standard

Phi Silica is the published reference for NPU-resident agentic LLM inference. Microsoft's December 2024 Windows Experience Blog post is one of the most technically detailed NPU agent writeups available. The summary:

- Based on a derivative of Phi-3.5-mini (3.3B parameters)
- 4-bit weight quantization
- Prompt processing fully on NPU at **650 tokens/sec prefill, ~1.5 W**
- Decode at ~27 tokens/sec on **CPU**, reusing the NPU's KV cache (hybrid execution)
- Long prompts decomposed into 64-token chunks
- Speculative decoding with a smaller draft model
- Tokenizer/embedding/LM head on CPU, transformer block on NPU
- "650 tokens/second prefill, ~1.5W power" — *Microsoft Windows Experience Blog*

Crucially, **all published Phi Silica numbers are from Snapdragon X Elite hardware**, not Intel. Microsoft has not published Intel-NPU-specific Phi Silica figures. Phi Silica reached Intel Copilot+ PCs through Windows Updates (KB5079266, KB5084176, KB5089866) during 2025, but the comparative performance data isn't in the public record.

The Phi Silica architecture is the template the rest of the chapter draws on. Three patterns from it generalize directly:

1. **Hybrid NPU+CPU execution** for transformer LLMs (prefill on NPU, decode on CPU)
2. **Tokenizer/embedding/LM head on CPU** while the transformer block runs on NPU
3. **Speculative decoding with a smaller draft model** to amplify NPU throughput

If you remember nothing else from this chapter, remember that this is what production NPU LLM deployment looks like in 2025–2026: not "all on NPU," but a careful partition with the NPU doing what it's best at.

Quote Worth Internalizing

Microsoft's Phi Silica post contains the most concise statement of why NPU agents matter:

> "NPUs can sustain AI workloads that exhibit emergent behavior (3 to 7B parameter SLMs) in a semi-continuous loop, allowing users to make limitless low-latency queries to the model… we now have the ability to run powerful reasoning agents as part of background operating system services."

This is the architectural shift the whole book has been pointing at. Not "AI in the cloud, called through a network." Not "AI on the GPU, blocking the user's foreground work." Something genuinely new: agents that live in the OS, available continuously, at a power budget the user doesn't notice. The NPU is what makes that economic.

Adobe: Documented, Limited

Adobe Premiere Pro's Audio Category Tagger is the only Adobe feature jointly confirmed by Adobe and Microsoft to run on Intel NPU (via DirectML, announced November 2024). Other Adobe AI features run differently:

- **Enhance Speech, Scene Edit Detection**: GPU via DirectML, not NPU
- **Photoshop Generative Fill**: cloud
- **Lightroom AI Denoise on Apple Silicon NPU**: enabled, then suspended due to artifacts

That last one is the cautionary tale. A shipped NPU feature was *withdrawn* because users noticed visual artifacts that didn't appear in the GPU/CPU path. This is exactly the failure mode Chapter 4's pre-flight validation is meant to catch.

Audacity and OBS Studio: Open Source NPU Agents

The cleanest open-source NPU case study is `intel/openvino-plugins-ai-audacity`. It's a plugin suite for Audacity exposing:

- DeepFilterNet noise suppression
- Demucs music source separation
- MusicGen audio continuation
- Whisper transcription
- Audio super-resolution

It includes a runtime device selector for CPU / GPU / NPU. The plugin docs explicitly warn users: "**10 to 30 seconds the first time** you run this effect, then on-disk caching kicks in." This is the right user-facing communication pattern — it sets expectations and shows you trust the user to understand cold-start.

OBS Studio has a similar plugin set (`intel/openvino-plugins-for-obs-studio`) for smart-framing and face-mesh effects on NPU.

These are good case studies precisely because they're open source. You can read the device-selection code, the fallback paths, and the user-facing communication patterns. They are also useful negative examples: notice how much code is required just to expose NPU as an option in a desktop app.

Gaps in the Public Record

Several Intel partners have been announced but have *no public Intel-NPU latency or quality numbers*:

- **DaVinci Resolve**: forum threads as of 2025 confirm Resolve does not engage the NPU even on supported hardware
- **Topaz Photo AI / Video AI**: no NPU support
- **CyberLink PowerDirector, Skylum Luminar Neo, BUFFERZONE, McAfee Deepfake Detector, Rewind, Deep Render**: announced Intel partners, no public numbers
- **Zoom, Webex, Teams**: use Windows Studio Effects (NPU) when present, but no published quality comparisons

Dell and Intel co-marketing claims **38% more battery life on a Zoom call** with NPU engaged (Dell KB 000223944). This is one of the few quantified third-party numbers in circulation, but it's a power claim, not a quality claim.

Intel-Published Benchmarks Worth Citing

For NPU LLM throughput, Intel's OpenVINO Model Hub gives the most reliable numbers:

- **DeepSeek-R1-Distill-Llama-8B INT4 on Core Ultra 7 NPU**: **6.10 tokens/sec, 163.10 ms/token**
- Same model on iGPU: 19.80 tokens/sec
- Same model on Arc B-Series dGPU: 75.75 tokens/sec

That single data point captures the entire economic argument for NPU agents: a 7-watt NPU delivers a third the throughput of a 15-watt iGPU. Use the NPU for sustained low-power workloads; use the iGPU when latency matters and you have the power budget.

For Lunar Lake specifically, Intel disclosed Llama 3.2 3B numbers: TTFT 28.5 ms for 32 input tokens, 31.4 ms for 1024 input tokens, throughput 32–35 tokens/sec. Intel did not disclose whether this is NPU, iGPU, or hybrid — which itself is a tell about what they're willing to commit to.

No Intel-published numbers exist for M2M-100, NLLB, MarianMT, or any other encoder-decoder NMT model on the NPU. If you're using the book's M2M-100 example, you will be measuring on your own hardware — there is no public baseline to defer to.

What This Section Bought You

You now have an honest map of the NPU agent landscape:

- **Copilot+ PCs and Phi Silica** are the reference deployment — only Lunar Lake on Intel side
- **Hybrid NPU+CPU execution** with tokenizer/embedding/LM head on CPU is the production pattern
- **Adobe, Audacity, OBS Studio** are the documented third-party deployments
- **Many announced partners haven't shipped measurable NPU features** — be skeptical of roadmaps
- **Intel's OpenVINO Model Hub** is the best public benchmark source — though biased toward LLMs
- **No public seq2seq translation benchmarks exist** on Intel NPU — you'll measure your own

The next section is where the rubber meets the road: building an end-to-end agentic translation assistant using the lessons of the book. We'll see how the abstract patterns turn into actual code, actual numbers (where they exist), and actual trade-offs.

Next: *5.2 A Worked Agentic Translation Assistant*

5.2 A Worked Agentic Translation Assistant

This section ties the book together by walking through an end-to-end agentic translation assistant. The goal isn't a polished product — it's to show how the patterns from Chapters 1–4 combine in real code, what the latency budget looks like in practice, and where the genuine uncertainties remain. We'll build a translation agent that detects language, translates with M2M-100 on Intel NPU, and optionally explains key vocabulary with a small reasoning LLM.

Architecture

The architecture mirrors what Phi Silica taught us in 5.1: don't put everything on the NPU. Put the right things on each engine.

```
┌──────────────────────────────────────────────────────────────────┐
│  Agent orchestrator (CPU, Python asyncio)                        │
│   ├─ Intent / language detection (XLM-R-base on NPU)             │
│   ├─ Tool dispatch                                               │
│   │    ├─ TranslationTool(M2M-100 418M INT8)                     │
│   │    │     ├─ Encoder → NPU                                    │
│   │    │     └─ Decoder + with-past → CPU                        │
│   │    ├─ ExplanationTool(Phi-3.5-mini INT4) → NPU prefill, CPU decode│
│   │    └─ FormatterTool(regex/string ops) → CPU                  │
│   └─ Response composition                                        │
└──────────────────────────────────────────────────────────────────┘
```

The flow for a typical request: user types something, the orchestrator runs language detection on NPU (cheap), picks the translation tool, runs M2M-100 encoder on NPU and decoder on CPU, optionally calls Phi-3.5-mini for an explanation, and assembles the response. Each step touches the engine best suited to it.

Latency Budget (Illustrative)

Below is an end-to-end latency budget for translating a single short sentence and optionally explaining it. **These numbers are illustrative.** Intel has not published M2M-100 or Phi-3.5-mini NPU benchmarks, so these are reasoned estimates extrapolated from related public results

(DeepSeek-Distill-Llama-8B at 6.10 tok/s on Core Ultra 7 NPU; Phi Silica's 650 tok/s prefill on Snapdragon X; the order-of-magnitude expectation for small encoder forwards on Lunar Lake NPU).

Stage	Device	Estimated latency
Language detection (XLM-R, 256 tok)	NPU	~5–20 ms
Agent planning hop	CPU	5–50 ms
M2M-100 encode (128-tok input)	NPU	tens of ms
M2M-100 decode (~25 output tokens, greedy)	CPU/iGPU	50–200 ms
Optional Phi-3.5 explanation (200 tokens)	NPU prefill, CPU decode	TTFT ~50 ms, decode @ ~25 tok/s
Total user-perceived (translate-only)	—	**sub-second**
Total user-perceived (translate + explain)	—	**1–2 seconds**

Two takeaways from this budget. First, sub-second translation is achievable on consumer NPU hardware — fast enough for interactive use, slow enough that async I/O matters. Second, the moment you add a second model (explanation), you've doubled the latency, and you need to decide whether the user actually wants that second step or whether the agent should default to translation-only with an "explain" affordance.

The Code

Here's a skeleton agent that ties the patterns together:

```
import asyncio
import openvino as ov
from optimum.intel import OVModelForSeq2SeqLM
import openvino_genai as ov_genai
from transformers import AutoTokenizer

class TranslationAgent:
    def __init__(self):
        core = ov.Core()
        core.set_property({"CACHE_DIR": "./.ov_cache"})

        # Language detection: small, static, runs on NPU
        self.lang_detect = LangDetectTool(device="NPU")
```

```
        # M2M-100 translation: encoder on NPU, decoder on CPU
        self.translate = TranslationTool(
            model_dir="ov_m2m100_418M_int8",
            encoder_device="NPU",
            decoder_device="CPU",
        )

        # Phi-3.5-mini for explanations: hybrid via LLMPipeline
        self.explain = ov_genai.LLMPipeline(
            "ov_phi35_mini_int4",
            device="NPU",
        )

    async def handle(self, user_input, target_lang="fr",
                     explain=False):
        # 1. Detect source language (NPU)
        src = await asyncio.to_thread(self.lang_detect, user_input)

        # Bail out if already in target language
        if src == target_lang:
            return {"translation": user_input,
                    "note": "already in target language"}

        # 2. Translate (NPU encoder + CPU decoder)
        translation = await asyncio.to_thread(
            self.translate, user_input, src, target_lang)

        result = {"src_lang": src,
                  "translation": translation}

        # 3. Optional: explain key vocabulary
        if explain:
            explanation = await asyncio.to_thread(
                self.explain.generate,
                f"Briefly explain key vocabulary in: {translation}",
                max_new_tokens=150,
            )
            result["explanation"] = explanation

        return result
```

Compare this to a cloud-native equivalent: a single API call to Google Translate plus an optional GPT-4 call. The cloud version is shorter, has higher quality on common language pairs, and requires zero local resources. The NPU version runs offline, respects user privacy, has predictable per-call cost (essentially zero), and gives you a reusable platform for other on-device AI features. Neither is universally right — the choice depends on the deployment context, exactly as Chapter 3.2 argued.

Where the Uncertainties Live

Several things in this example are genuinely uncertain and worth flagging for any reader who builds on it:

M2M-100 quality after quantization on NPU. The arXiv work on NMT quantization (2509.23990) studied NLLB-200 on GPU, not M2M-100 on NPU. The closest signal: "even aggressive quantization (INT4) preserved high levels of accuracy and fluency, with trade-offs more pronounced in low-resource settings." Whether that holds for M2M-100 on Intel NPU is unknown — measure on FLORES devtest before claiming production quality.

Whether M2M-100 actually compiles cleanly on Intel NPU. Intel's validated NPU LLM list is exclusively decoder-only or VLM models (Llama, Mistral, Qwen, Phi, Gemma, MiniCPM, Qwen-VL). M2M-100 and NLLB are not on it. The encoder may compile fine (it's a standard transformer encoder); the decoder is more fragile and likely to need CPU fallback. The hybrid pattern in the code above hedges against this.

Phi-3.5-mini availability for NPU. OpenVINO publishes Phi-3.5-mini in INT4 variants (`OpenVINO/Phi-3.5-mini-instruct-int4-cw-ov`), and `LLMPipeline(device="NPU")` is documented to work for Phi-class models. This is the more reliable side of the example.

The "explanation" use case is contrived. Why would a translation tool also explain vocabulary? Because it's a plausible agentic composition that exercises two NPU tools simultaneously and forces the memory-budget conversation. Real agents may do something else with the second model — summarization, clarification, sentiment annotation, formatting. The pattern matters; the specific task is illustrative.

What This Architecture Earns You

Three things this design buys that a simpler approach doesn't:

Privacy. Translation text never leaves the device. For users translating personal correspondence, business documents, or medical notes, this is the entire reason to build NPU-local in the first place.

Predictable latency. No network jitter, no rate limits, no vendor outages. The same machine produces the same latency every time, within hardware noise.

A platform. Once you've built the orchestrator, language detector, translation tool, and explanation tool, adding a fifth tool (transcription, summarization, code translation, anything else NPU-capable) is incremental work. The expensive part — the orchestration, the device partitioning, the cache and lifecycle management — is already done.

The fourth thing it buys you is operational complexity that the cloud equivalent didn't have. You own the model, the quantization, the driver compatibility matrix, the cache invalidation, and the fallback paths. That cost is real. Build NPU agents because the privacy, latency, or cost wins are worth that operational tax — not because NPUs are exciting.

What This Section Bought You

You now have an end-to-end picture of an NPU agent in practice:

- **Phi Silica's architecture** generalizes — hybrid NPU+CPU execution, with tokenizer/embedding/LM head on CPU
- **The latency budget** is sub-second for translation, 1–2 seconds with an explanation step
- **The code is modest** — a few hundred lines if you build cleanly — once you have the right primitives
- **Some uncertainties remain** about specific models (M2M-100 quantization, decoder compilation on NPU) that real deployment will resolve through measurement
- **The architecture earns you privacy, predictability, and a platform** at the cost of operational complexity

The next section, closing the book, catalogues the anti-patterns to watch for and the lessons distilled from the case studies — including INT4 vs INT8 trade-offs specific to encoder-decoder seq2seq models like M2M-100.

Previous: *5.1 What's Actually Shipping on Intel NPUs* **Next:** *5.3 Anti-Patterns and Lessons*

5.3 Anti-Patterns and Lessons

We've covered foundations, state, tools, deployment, and case studies. This final section pulls together the failure modes that recur in real NPU deployments — the things that look like they should work but don't — and the durable lessons distilled from the public record. It also tackles the question this book has flirted with throughout: when to quantize aggressively (INT4) versus conservatively (INT8) for encoder-decoder seq2seq models like M2M-100.

Anti-Patterns

Things that look like they should work on Intel NPU but don't

`OVModelForCausalLM(device="NPU")` — produces dynamic-shape graphs that crash. Use `openvino_genai.LLMPipeline` instead, which manages static-shape compile internally.

Beam search on NPU — not supported. Greedy or multinomial only. If your translation quality depended on beam-4 or beam-8, you'll be running the decoder on CPU or iGPU.

`log_probs` **from NPU LLMs** — not returned. Most evaluation harnesses (lm-eval, HELM) probe model behavior via token-level probabilities and will fail against the NPU build. Run evals against the CPU build of the same model.

MoE models (Mixtral, DeepSeek-V2/V3) — not in Intel's validated NPU list; gating layers fall back to CPU/GPU silently.

Embedding models for RAG (e.g., `bge-large`) — fail to compile on NPU per openvino_notebooks issue #2364. Run them on iGPU.

WSL2 NPU passthrough — Intel's `linux-npu-driver` issue #56 (filed November 2024) is **still open**. Not supported as of May 2026. Workaround: Windows-host NPU proxy with WSL2 clients over localhost.

NPU on Windows 10 — deprecated in driver 32.0.100.4621 (Feb/Mar 2026). All NPU development should target Windows 11.

Diffusion models larger than SD 1.5 (SDXL UNet, FLUX) — exceed NPU SRAM on Meteor Lake; spill to iGPU or are infeasible.

Treating TOPS as headroom. NPU LLM inference is memory-bandwidth-bound (~100 GB/s ceiling on LPDDR5X), not compute-bound; a 48-TOPS Lunar Lake delivers ~20 tok/s on an 8B model. Intel's own NPU acceleration library docs are explicit: decode is "DRAM Bandwidth" bound. The TOPS number tells you about *prefill*, not *decode*.

Custom `topK` in YOLO post-processing, `ScatterND/Gather` with INT64 indices, `as_convolution` on 0-channel grouped conv — all documented to break NPU compile (issues #29297, #34617, #34450).

Anti-patterns in agent design that compound on NPU

Calling the model for things a regex would do. The agent doesn't need an LLM to extract a phone number from text. NPU is sequential and expensive — every avoidable call wastes the entire engine's budget.

Long system prompts. Every system prompt token is prefilled on the NPU on every cold start (or every request without prefix caching). A 1500-token system prompt that could be 500 tokens costs you ~1 second of TTFT.

Synchronous orchestration. Blocking the asyncio event loop while waiting for a 200ms NPU inference means the rest of the agent stops too. Always `await asyncio.to_thread`.

Pre-baking NPU blobs into your container image. Driver and OpenVINO updates invalidate them; users get cryptic crashes. Compile on first run, cache on disk.

Putting everything on the NPU because you can. The NPU is the smallest engine. Things that don't fit go on iGPU or CPU. Phi Silica puts only the transformer block on NPU — and Phi Silica is the reference.

INT4 vs INT8 for Encoder-Decoder Seq2Seq

This is the question many readers will face directly with M2M-100. The honest summary is that **this is mostly an open question in public literature.** Decoder-only LLMs dominate published INT4 work; OpenVINO's HF org publishes `*-int4-ov` and `*-int4-cw-ov` collections for decoder-only models. Intel's validated NPU LLM list is exclusively decoder-only or VLM. M2M-100 and NLLB are not on it. Whisper (encoder-decoder speech) is the one seq2seq family with public NPU evidence, via OpenVINO GenAI's `WhisperPipeline` and the Audacity plugin.

The closest published study is arXiv:2509.23990 ("The Hidden Costs of Translation Accuracy"), which benchmarks NLLB-200 across FP32/FP16/INT8/INT4 on an A100 GPU — not NPU. Its key

finding: "even aggressive quantization (INT4) preserved high levels of accuracy and fluency… trade-offs are more pronounced in low-resource settings." Distillation (3.3B → 600M) is a far bigger BLEU lever than INT4-vs-INT8.

Practical recommendation for M2M-100 on Intel NPU:

Component	Recommendation
Encoder	**INT8 weight-only** — short, static, FP16 activations on NPU; the safe default
Decoder	**INT4 SYM, group_size=128** if you must compress; expect 0.5–2.0 BLEU drop on high-resource pairs based on the analogous NLLB GPU study; benchmark on FLORES devtest yourself
1.2B+ models	Channel-wise (`-1`) quantization combined with AWQ + Scale Estimation to recover accuracy
Lunar Lake exclusive	**NF4** tends to beat INT4-CW on 7B-class accuracy when available

AWQ + dynamic activation quantization is dangerous. OpenVINO docs explicitly warn that AWQ may hurt accuracy when combined with dynamic activation quantization. Pick one.

The deeper lesson: **measure on your real task distribution**, not on perplexity. A 0.5-point BLEU drop on FLORES devtest does not predict a 0.5-point quality drop on, say, legal-document translation. Quantization-induced degradation is workload-specific.

Distilled Lessons

Across the book and across the public case studies, a small set of lessons recur. These are the things experienced NPU practitioners would tell you over coffee:

Performance per watt is the value prop, not performance. The iGPU is faster. The NPU is more efficient. Build for that. Microsoft's Phi Silica "56% power-consumption improvement vs CPU" is the type of claim that motivates NPU use — not "X% faster than GPU."

Static shapes win. Pick a sequence length, pad to it, compile once. Recompiling on Intel NPU is expensive (seconds to tens of seconds). Dynamic shapes fall back to CPU or fail outright.

Memory is the wall, not compute. Decode is DRAM-bandwidth-bound. INT4 is the entry ticket because halving weights halves decode latency, not because INT4 is intrinsically magical.

Hybrid execution is the production pattern. Phi Silica's CPU-tokenizer + NPU-prefill + CPU-decode is the template. Don't aim for "all on NPU."

The driver is a third-party dependency. Drivers update without your consent, can change output behavior, and aren't always uniform across SKUs. Pre-flight validation is your safety net.

Cold-start is a UX problem, not a perf problem. 10–30 seconds is fine if you tell the user. It's catastrophic if you don't. Audacity gets this right by showing the user explicit copy about it.

Skip the NPU if you can. If your workload is dynamic-shape, beam-search-dependent, MoE, embedding-based, or large-diffusion — run it on the iGPU. NPU is not a strictly better target; it's a *different* target with a specific shape.

Instrument heavily. OS-level telemetry is uneven. Application-level metrics — latency by stage, cache hits, compile time, fallback rate, tool distribution — are where you'll actually catch regressions.

Honest deployment costs are real. You own the model, the quantization, the driver matrix, the cache invalidation, and the fallback paths. Build NPU agents because the privacy, latency, or cost wins are worth that operational tax — not because NPUs are exciting.

Where the Field Is Going

A short forward-looking note, with the caveat that hardware roadmaps and software stacks both move fast:

- **Panther Lake (Core Ultra 300, 2026) brings NPU 5** with native FP8 support, smaller per-tile MAC array but improved efficiency. Expected to extend Copilot+ certification to more SKUs.
- **OpenVINO 2026.x** continues to expand the NPU LLM path; the API namespace consolidated in 2026.0.
- **Hybrid execution will get easier** as both Intel's Compiler-In-Plugin and OpenVINO's automatic device-partitioning mature.
- **The non-LLM seq2seq story remains thin** — translation, ASR, OCR on NPU is mostly DIY. If you want this book's M2M-100 path to be smoother in 2027, it'll need community effort that doesn't exist yet.
- **Power telemetry may or may not improve.** HWiNFO maintainer reports Intel has no commitment. Plan around its absence.

Closing the Book

You came in knowing transformer architecture. You leave with a working model of how an agent actually operates inside a tightly constrained accelerator's limits, where the abstract trade-offs become real engineering decisions, what shipped products look like, and what doesn't quite work yet.

The pattern of this book has been to be honest about what we know and what we don't. NPU agents are a young deployment paradigm with a lot of public confusion about what's measured, what's marketed, and what's possible. The numbers in this book that aren't cited are labeled illustrative. The recommendations that work in 2026 may need revision in 2027. The Intel NPU stack that's still maturing today will look different in a year.

What won't change is the architectural shift the NPU enables: **agents that live inside the operating system, available continuously, at a power budget the user doesn't notice.** That's the prize. The work in this book is the cost of admission.

If you're building one of these systems, the most useful thing you can do is measure on your hardware, document what you learn, and share it. The public record of NPU deployment is thin precisely because most builders haven't published. The book closes with the same request that opened it: pick a target NPU, pick a candidate model, and actually measure TTFT and ITL on it. Everything else flows from there.

Previous: *5.2 A Worked Agentic Translation Assistant*

— End of book —