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*

Revision #2
Created 2026-05-12 17:29:23 UTC by Admin
Updated 2026-05-12 19:00:04 UTC by Admin