3.1 Designing Tools for NPU-Bound Agents

Chapter 2 ended with a claim: tool selection is a decision problem, not a search. This chapter goes further. The *tools themselves* — what they do, where they run, how they're shaped — are part of agent architecture, not separate from it. Get the tool design right and an NPU-bound orchestrator becomes capable. Get it wrong and even a fast model spends its time waiting on slow plumbing.

To keep this concrete, we'll use **Intel Core NPU** (the integrated accelerator in Core Ultra processors) and **M2M-100** (Meta's 100-language translation model) as a running example throughout the chapter. Translation is an unusually clean agentic tool: stateless, finite input space, deterministic with greedy decoding, useful in many higher-level workflows. It's also a model that *actually fits* on the hardware, which lets us talk about real numbers rather than hypotheticals.

What Makes a Good NPU-Bound Tool

A tool the orchestrator can call efficiently has four properties:

Stateless or near-stateless. The orchestrator shouldn't need to reason about hidden state inside the tool. Each call is fully determined by its inputs. Translation passes trivially: `translate(text, src, tgt)` has no memory of previous calls. A retrieval tool against a vector DB also qualifies if the index is treated as read-only. A "remember this fact" tool does not — and should be modeled as a long-term memory store, not a tool call.

Finite, validatable input space. The agent's planner can cheaply check input validity before calling the tool, which saves a wasted NPU compile or a slow runtime error. M2M-100 supports 100 source languages × 100 target languages = 9,900 directions, all enumerable. Compare with "search the web" — infinite input space, no pre-validation possible, every call is a leap of faith.

Predictable resource footprint. The orchestrator needs to know that calling tool X costs roughly Y memory and Z milliseconds. NPU-bound tools have an additional twist: their footprint is set *at compile time*, not call time. A translation tool compiled for sequence length 128 cannot accept sequence length 256 without recompiling — which, on Intel NPU, takes seconds to tens of seconds. Tools should be sized once and reused, not recompiled on demand.

Deterministic where possible. Reproducibility makes the agent debuggable. Greedy decoding on M2M-100 produces the same output for the same input, every time. As soon as you add beam search or sampling, two calls with identical arguments produce different outputs and your bug reports become unreproducible.

The M2M-100 Translation Tool

Concretely, here's the shape of a translation tool wrapping M2M-100 on Intel NPU. Note the patterns: load-once, validate-cheap, compile-static, call-stateless.

```
from optimum.intel import OVModelForSeq2SeqLM
from transformers import AutoTokenizer

class TranslationTool:
    """Stateless translate(text, src, tgt) backed by M2M-100 on Intel NPU."""

    schema = {
        "name": "translate",
        "description": "Translate text between any of 100 languages on-device.",
        "parameters": {
            "type": "object",
            "properties": {
                "text":     {"type": "string", "maxLength": 2000},
                "src_lang": {"type": "string", "enum": SUPPORTED_LANGS},  # 100
                "tgt_lang": {"type": "string", "enum": SUPPORTED_LANGS},
                "max_new_tokens": {"type": "integer", "default": 128},
            },
            "required": ["text", "src_lang", "tgt_lang"],
        },
    }

    def __init__(self, model_dir="ov_m2m100_418M_int8",
                 encoder_device="NPU", decoder_device="CPU"):
        self.tok = AutoTokenizer.from_pretrained(model_dir)
        self.model = OVModelForSeq2SeqLM.from_pretrained(
            model_dir,
            encoder_device=encoder_device,
            decoder_device=decoder_device,
            decoder_with_past_device=decoder_device,
            ov_config={"CACHE_DIR": "./.ov_cache",
                       "PERFORMANCE_HINT": "LATENCY"},
        )
        self.model.reshape(batch_size=1, sequence_length=128)
        self.model.compile()  # one-time cost: seconds to tens of seconds
```

```
    def __call__(self, text, src_lang, tgt_lang, max_new_tokens=128):
        if src_lang not in self.tok.lang_code_to_id:
            raise ValueError(f"src_lang {src_lang!r} not supported")
        self.tok.src_lang = src_lang
        enc = self.tok(text, return_tensors="pt",
                       truncation=True, max_length=128, padding="max_length")
        out = self.model.generate(
            **enc,
            max_new_tokens=max_new_tokens,
            forced_bos_token_id=self.tok.get_lang_id(tgt_lang),
            num_beams=1,  # NPU does not support beam search
        )
        return self.tok.batch_decode(out, skip_special_tokens=True)[0]
```

Three details in this code carry most of the lessons of the chapter:

The `reshape(...)` call forces a static shape. Intel NPU compilers require fully static shapes for non-LLM models, and dynamic-shape graphs either fall back to CPU (slow) or fail to compile (broken). Padding every input to length 128 wastes some compute on short strings, but the alternative — recompiling for each new length — is much worse.

The `encoder_device="NPU", decoder_device="CPU"` split is not an oversight. Encoder-decoder seq2seq models split cleanly: the encoder runs once over a fixed-length input (NPU-friendly), while the decoder runs autoregressively with a growing KV cache (NPU-hostile, because the cache is dynamic). The same pattern shows up in Microsoft's published Phi Silica architecture, which places the tokenizer, embedding, and LM head on CPU while only the transformer block runs on NPU.

The `forced_bos_token_id` is the single most important detail of M2M-100 inference. The decoder needs a token telling it which of 100 languages to generate; omit it, and the model produces fluent text in some random language. This is not an NPU thing — it's an M2M-100 thing — but it bites every team integrating the model for the first time.

Tool Schema Design Beyond JSON Validation

The schema above is what the agent sees. A good schema does three things beyond declaring types:

It states real constraints. `maxLength: 2000` isn't arbitrary; it's tied to the NPU compile-time sequence budget. If the orchestrator sends 5000 characters, the tool truncates or splits — and the

schema documents that contract. Tools that silently lose data when over their limit are landmines.

It enumerates discrete options instead of accepting free-form strings. `src_lang: "fr"` is validatable; `src_lang: "French"` requires fuzzy matching and accumulates edge cases. M2M-100's tokenizer uses BCP-47-ish codes (`en`, `fr`, `zh`, `pt`...); the schema enforces them.

It includes a length cap on outputs. `max_new_tokens` puts a hard bound on how long a tool call can take. An agent that says "translate this 50-page document" without a cap can wedge the NPU for minutes.

Long Inputs: The Orchestrator's Job, Not the Tool's

When the user gives the agent more text than the tool's static shape can handle, the temptation is to recompile the tool for a larger input. Resist it. Recompiling a model on Intel NPU takes several seconds in the best case and tens of seconds in pathological ones — that's a user-perceptible hang.

Instead, push the chunking responsibility up to the orchestrator. The agent splits the document at sentence boundaries, calls the tool repeatedly on chunks of the right size, and reassembles the outputs. This:

- Keeps the NPU compile graph fixed and warm
- Lets the orchestrator parallelize chunks across async invocations
- Surfaces progress to the user ("translating page 3 of 50...") instead of opaque waiting

For M2M-100 specifically, sentence-level chunking actually *improves* quality, because the model was trained on sentence-pair data and degrades on very long inputs.

What This Section Bought You

You now have a model for what an NPU-bound tool looks like in practice:

- **Four properties** define a good tool: statelessness, finite input space, predictable footprint, determinism
- **Compile-time shape decisions** dominate runtime flexibility on Intel NPU — pick a sequence length and stick with it
- **Encoder-decoder splits** map naturally onto NPU+CPU partitioning
- **Schemas encode real constraints**, not just types — length limits, valid enums, output caps
- **Long-input handling lives in the orchestrator**, not in the tool

The next section turns to the question that follows directly: now that the tool exists, *should* the agent call it locally, or punt to a cloud API? The answer is more nuanced than "always local" or "always cloud," and the trade-offs are different on NPU than on either CPU or cloud GPU.

Next: *3.2 Local-NPU vs Cloud Tools: A Real Trade-Off Table*

Revision #2
Created 2026-05-12 17:24:33 UTC by Admin
Updated 2026-05-12 18:59:05 UTC by Admin