Tool Use & Integration Patterns

Designing lightweight tools for NPU-based agents. Async I/O and non-blocking integrations. Local vs. remote tool execution trade-offs. Building tool abstractions that respect hardware constraints.

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*

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

If the tool runs locally on the NPU, the orchestrator pays a one-time compile cost and then has predictable, private, offline-capable inference. If the tool runs in the cloud, the orchestrator pays per-call network latency and per-token API fees but gets larger models and easier operations. Choosing between them is one of the most consequential decisions in agent design — and one of the most often made on vibes.

This section gives you a defensible framework. We'll use M2M-100 on Intel Core NPU vs cloud translation APIs (Google Translate, DeepL) as the worked example, but the framework generalizes.

The Honest Comparison

Most "local vs cloud" comparisons cheat by leaving out the items that don't favor their conclusion. Here is the comparison with nothing hidden:

Dimension	NPU-local M2M-100 418M	Cloud Translation API
Cold-start to first token	10–30s first compile, ~1s with `CACHE_DIR`	50–100 ms TLS + DNS
Steady-state latency (25-token sentence)	Order of 50–200 ms on Lunar Lake*	100–400 ms round-trip from a mid-latency region
Throughput, single stream	~5–20 sentences/sec*	Rate-limited (Google Translate: 600 req/min default)
Quality on common pairs (en→fr, en→de)	M2M-100 BLEU competitive but lower than commercial systems	Best in class
Quality on direct non-English pairs	M2M-100 paper: +10.2 BLEU over English-pivot models	Often pivots through English internally
Privacy	On-device, never leaves	Sent to vendor; data residency concerns
Cost	One-time NPU compute, ~5–7W power	$20 per million chars (Google), $25 per million (DeepL)
Offline	✓	✗
Latency variance	Predictable, no network jitter	Highly variable on mobile/edge networks
Maintenance	You own model updates, drift, eval	Vendor handles it

Dimension	NPU-local M2M-100 418M	Cloud Translation API
Failure modes	NPU compile errors, driver bugs	Network outage, rate limits, vendor pricing changes

* No public Intel benchmarks exist for M2M-100 on Intel NPU specifically. These numbers are extrapolations from related published results (DeepSeek-Distill-Llama-8B at 6.10 tokens/sec on Core Ultra 7 NPU, Phi Silica's 650 tokens/sec prefill on Snapdragon X NPU) and should be treated as illustrative until you measure on your target hardware.

What This Table Doesn't Tell You

A naive read of the table suggests "use cloud when online, NPU when offline." That's a bad heuristic. The real decisions are more interesting:

Privacy is binary. Medical, legal, or enterprise data often *cannot* be sent to a cloud API regardless of latency. If your agent translates patient notes or contract clauses, the cloud option is not actually an option, and the local NPU's quality ceiling becomes the entire problem to solve.

Cost compounds. A cloud API at $20 per million characters seems cheap until your translation agent processes 10 million characters per user per month. On-device translation, even on a laptop NPU drawing 7 watts, is essentially free at the per-call level after the device is purchased.

Cold-start kills first impressions. The 10–30 second first-compile cost is *invisible* if your app preloads the model at startup. It's *catastrophic* if the first time a user clicks "translate" they wait 30 seconds. Audacity's OpenVINO plugin documents this explicitly to its users: "the first time you run this effect, it will take 10 to 30 seconds." Don't hide cold-start from users — schedule around it.

Network failure is a real failure mode. Cloud-only agents fail catastrophically on planes, in tunnels, on flaky hotel WiFi. NPU-local agents don't. For some users (field workers, travelers, journalists) this is the entire reason to choose local.

The Hybrid Default

In practice, the best NPU-based agents are *not* purely local. They're hybrid: local-by-default, cloud-as-fallback or cloud-as-upgrade.

```
class HybridTranslationTool:
    def __init__(self, local_tool, cloud_client, prefer_local=True):
        self.local = local_tool
        self.cloud = cloud_client
        self.prefer_local = prefer_local
```

```
async def __call__(self, text, src, tgt, quality="default"):
    # Quality-driven routing: send hard pairs to cloud
    if quality == "premium" and self.cloud.available():
        try:
            return await self.cloud.translate(text, src, tgt)
        except CloudError:
            pass  # fall through to local

    # Privacy-driven routing: PII stays local
    if contains_pii(text):
        return self.local(text, src, tgt)

    # Default: local
    if self.prefer_local:
        return self.local(text, src, tgt)

    # Online fast-path with local fallback
    try:
        return await asyncio.wait_for(
            self.cloud.translate(text, src, tgt), timeout=0.5)
    except (CloudError, asyncio.TimeoutError):
        return self.local(text, src, tgt)
```

The router itself is cheap (a few microseconds of Python) and gives you four useful behaviors: premium quality on demand, automatic privacy preservation, fallback on cloud failure, and offline operation. The agent's planner doesn't need to know which path took the request — that's an implementation detail of the tool.

Async I/O Is Not Optional

A tool call that takes 100 ms is short enough to *seem* synchronous and long enough to *feel* sluggish if the agent blocks the event loop. NPU inference is squarely in this band. Every NPU-backed tool should be async-callable.

OpenVINO offers two patterns:

The simple wrapper. Run the synchronous inference in a thread pool:

```
import asyncio
async def translate_async(tool, text, src, tgt):
    return await asyncio.to_thread(tool, text, src, tgt)

# The orchestrator can now plan in parallel with translation
results = await asyncio.gather(
    *[translate_async(tool, t, "en", "fr") for t in docs[:4]])
```

The native AsyncInferQueue. For higher throughput with a single OpenVINO model:

```
queue = ov.AsyncInferQueue(compiled_model, jobs=4)
queue.set_callback(lambda req, userdata: handle(req.get_output_tensor()))
for inp in inputs:
    queue.start_async(inp)
queue.wait_all()
```

`AsyncInferQueue` keeps the NPU fed: while inference N runs, inputs N+1..N+jobs are queued, and outputs are dispatched via callbacks. For single-request paths the `to_thread` wrapper is enough.

One Intel-specific caveat: if you need strict in-order semantics (e.g., a streaming translation where output order matters), set `NPU_RUN_INFERENCES_SEQUENTIALLY=YES`. The NPU plugin will serialize requests internally, removing concurrency but guaranteeing FIFO completion.

When the Cloud Option Doesn't Exist

We've talked as if cloud is always available. It often isn't:

- **Air-gapped deployments** (defense, classified networks, secure government) prohibit any external API
- **Cost-sensitive scale** (millions of requests per day per device) makes per-call cloud pricing untenable
- **Regulatory residency** (GDPR, HIPAA, finance) sometimes makes any cross-border data transit illegal
- **Embedded products** (kiosks, vehicles, industrial equipment) often don't have reliable network at all

In these contexts the "trade-off" collapses: NPU-local is the only option. The trade-off becomes *which* local model — and at *which* quality tier — fits the hardware budget. For the M2M-100 family, the 418M variant is the obvious starting point (slot easily into 2GB RAM with INT8 quantization), the 1.2B variant trades capability for memory, and the 12B variant is a non-starter on consumer NPUs.

What This Section Bought You

You now have a defensible framework for the local-vs-cloud decision:

- **The honest comparison** has many more dimensions than latency — privacy, cost, offline, variance, maintenance, failure modes
- **Hybrid is usually the right default** for consumer products with optional connectivity
- **Routing logic belongs in the tool**, not the orchestrator — the agent shouldn't have to know which path took its call
- **Async I/O is mandatory** for tools in the 50–500 ms latency band
- **For some deployments the cloud option doesn't exist** — and that's when NPU-local stops being a feature and becomes the entire product

The next section pulls back to the system level: now that you have tools, how do you orchestrate multiple of them across the CPU, NPU, and integrated GPU on a single Intel Core SoC?

Previous: *3.1 Designing Tools for NPU-Bound Agents* **Next:** *3.3 Multi-Device Orchestration on a Single SoC*

3.3 Multi-Device Orchestration on a Single SoC

A Core Ultra SoC isn't one engine — it's three. CPU cores for general-purpose work, an integrated GPU for parallel compute and graphics, and the NPU for low-power neural inference. An agent that uses only one of them is leaving capacity on the table. An agent that uses all three carelessly is fighting itself for memory, thermals, and power. This section is about partitioning intelligently.

The Three Engines

Each device on the Intel Core SoC has a distinct sweet spot. Designing around them starts with knowing what each is actually good for:

CPU (P-cores + E-cores). Best for: orchestration logic, tokenization, branchy control flow, async I/O, tool dispatch, fallback when other devices are saturated. Worst for: sustained matrix multiplication at low power. Memory: shared with the rest of the SoC, with the largest cache hierarchy.

Integrated GPU (Xe-LPG on Meteor Lake, Xe2 on Lunar Lake). Best for: dynamic-shape models, large transformer decoding, diffusion, anything that won't fit in NPU memory. ~67 TOPS INT8 on Lunar Lake. Worst for: low-power background workloads — the GPU is the hottest device on the SoC under load. Memory: shared LPDDR5X with CPU and NPU; no dedicated VRAM.

NPU (3rd-gen on Meteor Lake/Arrow Lake, NPU 4 on Lunar Lake). Best for: sustained, low-power, static-shape inference; short-prompt LLM prefill; vision encoders; audio enhancement. Up to 48 TOPS INT8 on Lunar Lake. Worst for: dynamic shapes, beam search, anything that doesn't fit in its memory model. Memory: shares LPDDR5X via system fabric; no dedicated NPU DRAM.

Here's the inconvenient truth most NPU marketing skips: on the same Stable Diffusion 1.5 workload, the Meteor Lake iGPU is **261% faster than the NPU at INT8** (Chips and Cheese measurement) — at roughly 3x the power. The NPU is not the fastest engine on the SoC. It's the most *efficient* one. Microsoft quantifies this for Phi Silica: putting the SLM on the NPU achieved "56% power-consumption improvement vs CPU." The NPU's value is performance-per-watt, not performance.

Mapping Agent Components to Devices

For our translation-agent example, the partitioning that works in production looks like this:

Component	Device	Rationale
Agent orchestrator (asyncio event loop)	**CPU**	Branchy logic, async I/O
Tokenizer (M2M-100 SentencePiece, vocab=128k)	**CPU**	Lookup-bound, no math
Language detection (XLM-R-base or similar)	**NPU**	Static shape, short input, sustained background
M2M-100 **encoder**	**NPU**	Fixed-length input after padding, encoder-only graph
M2M-100 **decoder + decoder-with-past**	**CPU or iGPU**	Autoregressive, dynamic KV cache
Optional reasoning LLM (Phi-3.5-mini INT4)	**NPU**	Sustained, low-power, fits NPU memory budget
Diffusion / image gen, if any	**iGPU**	Large dynamic graphs, NPU memory insufficient

This isn't theoretical. It mirrors Microsoft's published Phi Silica architecture, which puts the tokenizer, embedding, and LM head on CPU while running only the transformer block on NPU. The pattern recurs because it follows the engines' actual capabilities, not their marketing.

OpenVINO's Multi-Device Primitives

OpenVINO exposes three virtual devices for cross-engine orchestration. Each solves a different problem.

AUTO picks the best device for the model and falls back automatically. The killer feature: while the NPU compiles (which can take seconds), AUTO runs the model on CPU so the user isn't blocked. When the NPU is ready, AUTO transparently switches over. **Important gotcha: NPU is not in AUTO's default priority list** — you must name it explicitly:

```
compiled = core.compile_model(model, "AUTO:NPU,GPU,CPU",
                              {"PERFORMANCE_HINT": "LATENCY"})
```

MULTI sends each request to one device, splitting a request stream across engines for throughput. Useful when you have many independent inference requests and want to parallelize across the NPU and GPU simultaneously:

```
compiled = core.compile_model(model, "MULTI:NPU,GPU")
```

HETERO splits a *single* model across devices at operator granularity, letting the NPU handle what it supports and the CPU handle what it doesn't. The OpenVINO documentation is explicit that "HETERO with NPU as primary is partially supported, for certain models" — meaning you should

treat it as empirical. Test with your specific model before committing:

```
compiled = core.compile_model(model, "HETERO:NPU,CPU")
```

In practice, most production NPU agents use AUTO for safety (with NPU listed first) and avoid HETERO for anything mission-critical, because operator-level fallback can silently introduce CPU-NPU transfers that destroy performance.

Concurrency and the Single-NPU Problem

A single Intel Core SoC has one NPU. The NPU's scheduler (Intel's LeonRT) supports multiple concurrent hardware contexts per the datasheet, but in practice, requests serialize end-to-end for LLM workloads. **OpenVINO Model Server on NPU is explicitly documented as sequential-only**: "OpenVINO Model Server with NPU acceleration processes the requests sequentially… benchmarking should be performed in `max_concurrency=1`."

This has architectural consequences. If your agent has two NPU-bound tools (say, translation + reasoning LLM), they cannot both be active simultaneously. You have three reasonable patterns:

Cold-swap. Only one model loaded at a time. Switch on demand. Simple and memory-efficient, but you pay the compile-or-load cost (seconds) on every switch.

Warm-coexist. Both models compiled and loaded, sharing the NPU memory budget. Switching between them is fast, but each model gets less memory. This is the pattern OVMS uses for its `model_repository`.

Time-share with checkpointing. Serialize one model's state to host RAM when switching. Less common; useful when neither model fits alone but you need both.

For most agent designs, warm-coexist on a Lunar Lake (16 GB RAM, 48-TOPS NPU 4) is workable for two small models (M2M-100 418M INT8 + Phi-3.5-mini INT4 together fit comfortably). Three medium models won't fit. Plan accordingly.

Common Integration Mistakes

The expensive mistakes practitioners make, drawn from GitHub issues and developer blog posts:

Re-loading the model on every call. First compile on NPU is 10–90 seconds; without `CACHE_DIR` you pay it every process launch. Always set `core.set_property({"CACHE_DIR": "./.ov_cache"})`. With cache, subsequent loads drop from tens of seconds to a fraction of a second.

Using `OVModelForCausalLM` instead of `openvino_genai.LLMPipeline` on NPU. The former produces dynamic-shape graphs that crash on NPU. The latter manages static-shape compile internally and exposes the right knobs (`MAX_PROMPT_LEN`, `MIN_RESPONSE_LEN`).

Tokenizer initialization in the hot path. SentencePiece load is 100–300 ms. Do it at app startup, never per-request. M2M-100's vocabulary is 128,112 tokens — that's a non-trivial parse.

Batch size > 1 on NPU. OpenVINO 2025.4+ internally reshapes to batch size 1 anyway. Use multiple async requests for concurrency, not larger batches.

INT8 weight-only LLM IR on NPU. Documented to crash with uncatchable 0xC0000005 (issue #35641). Intel's NPU LLM path requires INT4 symmetric quantization. The error is silent at compile time, fatal at first `generate()` call.

Forgetting `forced_bos_token_id` on M2M-100. Produces silent garbage in random target languages. Not an NPU issue, but the most common M2M-100 integration bug.

Wrapping Up Chapter 3

Tools and orchestration are where the abstract constraints from Chapters 1 and 2 become concrete decisions:

- **Good tools are stateless, finite, predictable, and deterministic** — translation is the textbook example because it satisfies all four
- **Static shapes drive everything on Intel NPU** — pick a sequence length, compile once, pad inputs to fit
- **Encoder-decoder models split naturally** across NPU (encoder) and CPU/iGPU (decoder with dynamic cache)
- **Local-vs-cloud is multi-dimensional**, not just latency — hybrid is usually right
- **The SoC has three engines**, each with a sweet spot — partition by capability, not by enthusiasm
- **OpenVINO's AUTO, MULTI, HETERO** are the orchestration primitives; AUTO with explicit NPU listing is the safest default
- **Six common mistakes** are responsible for most NPU integration failures — `CACHE_DIR`, `LLMPipeline`, lazy tokenizer init, batch size, INT4-symmetric quantization, and `forced_bos_token_id`

Chapter 4 turns to what happens *after* you ship: the deployment stack, observability, A/B testing, hotswaps, and the operational surprises that only show up in production.

Previous: *3.2 Local-NPU vs Cloud Tools: A Real Trade-Off Table* **Next:** *Chapter 4: Production Deployment & Observability*

3.4 Structured Outputs and Constrained Decoding

An agent is only as reliable as the parser that reads its output. Chapter 3.1 covered designing the tools; Chapter 3.2 weighed local against cloud; Chapter 3.3 routed work across devices on the SoC. This section closes the loop on the agent-tool contract: how do you get the LLM to actually return something a tool can ingest, deterministically, every time, when the LLM in question is greedy-only and quantized to within an inch of its life?

The standard cloud answer is "set `response_format=json_object` and add `temperature=0.1`." That doesn't translate to NPU. On Intel NPU's `LLMPipeline`, sampling temperature is irrelevant — the only mode is greedy — and `response_format` isn't an OpenVINO concept. You get whatever the model emits, and if INT4 quantization has shifted the JSON-mode logit just enough for a stray "Here is the result:" prefix to win the argmax at position zero, your parser fails. This section is about preventing that.

The Three Tiers of Structured Output

There are three escalating tactics for getting the LLM to produce parseable output. Each costs more to implement and is more reliable than the last. Use the cheapest one that works for your use case.

Tier 1: Prompt engineering. Tell the model what format you want, give a few-shot example, and hope. Works surprisingly often at FP16 on a competent model. Degrades faster than you'd expect under quantization. Free; no infrastructure required.

Tier 2: Output filtering and retry. Parse the model's output; if it fails, retry with an error message included in the next prompt. Costs you extra inference round-trips when the model misbehaves. Reasonable for low-frequency failures (~5% error rate); ruinous for high-frequency failures because each retry burns ~5–10 seconds on NPU.

Tier 3: Constrained decoding. At each decode step, mask out tokens that would make the output invalid according to a formal grammar or schema. The model can only ever produce parseable output because the alternative tokens are zeroed in the logits before argmax. Implementation is real work — you need a grammar engine that hooks into the sampling loop — but the reliability is total. Once correctly wired up, parser failures simply cannot happen.

For agents that go to production, Tier 3 is usually the right answer. Tier 1 alone fails too often in practice; Tier 2 burns too much budget on retries when greedy decoding means the model's failure

mode is deterministic — if it failed once on a given prompt, it'll fail every time.

Why This Is Harder on NPU

Three things that make structured output specifically harder on NPU than on a cloud-GPU LLM API:

Greedy-only sampling. On GPU, you can sample with `temperature=0.7` and re-roll until you get valid JSON. The randomness gives you a free retry budget. On NPU, the same prompt always produces the same output. If it's broken, it's broken across every invocation.

Quantization shifts the logit landscape. Section 1.4 covered how INT4 quantization shifts argmax decisions on edge cases. Structured output lives at exactly the kind of edge: the difference between emitting `{"answer":` and `{ "answer":` (extra space) is two adjacent tokens with nearly identical FP16 logits. Quantization can flip which wins. Suddenly your parser, which assumed no leading space, breaks.

No native JSON-mode primitive. OpenAI's API has `response_format={"type": "json_object"}`, which routes to a specialized backend path. OpenVINO GenAI's `LLMPipeline` does not. You have to implement the constrained-decoding integration yourself, or use a third-party library.

The good news: the situation is improving. OpenVINO 2026.x has been adding structured-output APIs incrementally, and the most popular third-party constrained-decoding libraries (Outlines, lm-format-enforcer, Guidance) can be wired into OpenVINO with moderate effort. As of May 2026 the integration is real work, not a one-line config.

How Constrained Decoding Works

The mechanism is conceptually simple. At each decode step, the model produces a vector of logits over the full vocabulary (~32K–128K entries, depending on tokenizer). Ordinary greedy decoding takes the argmax over that vector. Constrained decoding adds a step before the argmax:

1. Track the current state in some grammar or schema (e.g., "we're inside a JSON object after a key, expecting a colon").
2. Determine the set of tokens that would be *valid* in that state (e.g., tokens that start with `:` or `: `).
3. Set the logits of all *invalid* tokens to negative infinity.
4. Take the argmax. The result is guaranteed to be valid.
5. Update the grammar state based on the chosen token.

For JSON, the grammar is fixed and well-known. The state machine tracks brace depth, whether the next token must be a key or a value, what types are still allowed, and so on. For JSON-schema-constrained output, the constraints are tighter — only specific keys are valid, only specific value types, only specific enum values — and the state machine encodes the schema as a tree.

The cost is mostly in step 2: efficiently computing the allowed-token mask. Naïve implementations check every vocabulary token at every step, which would be too slow. Production implementations precompute caches keyed on grammar state — given state S, here are the valid token IDs — and lookup is O(1). The libraries below all do this.

The Libraries

Outlines (`dottxt-ai/outlines`) is the most popular constrained-generation library in the Python ecosystem. Supports regex constraints, JSON schema, and arbitrary Lark grammars. Has an OpenVINO backend integration. Pattern:

```
from outlines import models, generate
import outlines


model = outlines.models.openvino("models/llama-3.1-8b-int4_npu")
generator = generate.json(model, schema=MyPydanticModel)
result = generator("List two countries in Europe.")
# result is guaranteed to be a valid MyPydanticModel instance
```

lm-format-enforcer (`noamgat/lm-format-enforcer`) is more lightweight, focused specifically on JSON and regex. Hooks into Hugging Face Transformers' `LogitsProcessor` interface, so anywhere that interface is supported (Optimum-Intel's `OVModelForCausalLM`, for instance), it plugs in. Often a better fit when you're already using `optimum-intel` directly rather than `openvino_genai.LLMPipeline`.

Guidance (`microsoft/guidance`) is more general, supporting multi-turn templates that interleave model-generated and developer-fixed content. Heavier than Outlines or lm-format-enforcer; more powerful if your use case actually needs the interleaved-template feature.

For most NPU agent use cases, **Outlines + JSON schema** is the right starting point. It's the most actively maintained, has direct OpenVINO support, and handles 95% of the realistic structured-output needs (tool calls, classification, extraction).

A Worked Pattern: Tool-Calling Schema

Take a concrete example. Your agent has three tools: `search_web`, `read_file`, `send_email`. You want the model to output a tool call like:

```
{
  "tool": "search_web",
  "arguments": {
    "query": "quarterly revenue Q3 2025"
```

```
  }
}
```

A Pydantic schema enforces this:

```
from pydantic import BaseModel, Field
from typing import Literal, Union

class SearchWeb(BaseModel):
    tool: Literal["search_web"]
    arguments: dict = Field(..., description="Must contain 'query'")

class ReadFile(BaseModel):
    tool: Literal["read_file"]
    arguments: dict = Field(..., description="Must contain 'path'")

class SendEmail(BaseModel):
    tool: Literal["send_email"]
    arguments: dict = Field(..., description="Must contain 'to', 'subject', 'body'")

ToolCall = Union[SearchWeb, ReadFile, SendEmail]
```

Wire it into Outlines:

```
from outlines import models, generate

model = models.openvino("models/llama-3.1-8b-int4_npu")
generator = generate.json(model, schema=ToolCall)

response = generator(
    "User asked: 'What were our Q3 numbers?'. Choose a tool to answer this."
)
# response is guaranteed to validate as ToolCall
# response.tool is one of the three Literal values
# response.arguments is a dict (the schema doesn't constrain its keys here,
#   but you can add tighter sub-schemas if needed)
```

The model can only emit tokens that lead to a valid `ToolCall`. It can't say "I don't know" or "Let me think about this" first — those tokens are masked. It must commit to a tool. Once committed, it must produce a valid arguments structure. Parser failures become impossible.

For tighter argument constraints, define explicit per-tool argument schemas:

```
class SearchWebArgs(BaseModel):
    query: str


class SearchWeb(BaseModel):
    tool: Literal["search_web"]
    arguments: SearchWebArgs
```

Now the model can only emit `{"tool": "search_web", "arguments": {"query": "..."}}` exactly. Adding extra keys is impossible.

The Costs You Pay

Constrained decoding isn't free. Three costs to budget for:

Decode latency increases 5–20%. Computing the allowed-token mask per step adds overhead. On NPU where decode is already bandwidth-bound, this overhead is real but not catastrophic — it adds CPU work between NPU forward passes, not bandwidth load. Outlines is particularly fast; lm-format-enforcer is comparable; Guidance can be slower.

The model's "free reasoning" is constrained too. If the model is forced into JSON from token zero, it can't first reason out loud and then commit to a tool. This is sometimes a real loss of capability — chain-of-thought reasoning is valuable for hard tool selections. The mitigation is a two-step pattern: first do an unconstrained reasoning step that picks a tool, then do a constrained step that emits the JSON. Two prefills, two decodes, but the reasoning is preserved.

Schemas that are too rigid produce nonsense outputs. If your schema says "the answer must be one of [A, B, C]" but the right answer is D, the model will pick whichever of A/B/C has the highest logit. The output validates; the answer is wrong. Constrained decoding turns "parser failures" into "schema-violation-of-reality failures" — same failure, different shape. Design schemas to include escape hatches: an optional `"unknown"` value, a `confidence` field that can be low.

Failure Modes

Specific things that go wrong when wiring constrained decoding into an NPU agent:

Tokenizer mismatch between Outlines and OpenVINO. Outlines builds its allowed-token cache against a specific tokenizer; if the OpenVINO export used a slightly different tokenizer config (e.g., `add_special_tokens=False`), the cached masks misalign and you get gibberish or compile errors. Verify tokenizer parity explicitly.

Schema is permissive in ways you didn't intend. A field typed as `dict` allows any keys, any values, infinite nesting. Constrained decoding will produce valid-but-useless JSON: `{"tool": "search_web", "arguments": {}}` is a valid `ToolCall` if `arguments` is just `dict`. Tighten with explicit sub-schemas.

The grammar engine doesn't know about a special token. EOS, BOS, padding tokens may need explicit handling in the grammar. Outlines handles this; some custom integrations don't. Symptom: the model "never stops" because the EOS token is masked out.

Mask computation becomes a hot path. For complex schemas (large enums, recursive structures), per-step mask computation can dominate CPU time. Profile. If it's bad, consider caching aggressively or simplifying the schema.

The compiled `LLMPipeline` doesn't expose a `LogitsProcessor` hook. Native `LLMPipeline.generate()` doesn't currently let you intercept logits between the model and the sampler. Two workarounds: (1) drop down to `OVModelForCausalLM` from Optimum-Intel, which uses standard Transformers `LogitsProcessor`, at the cost of losing some `LLMPipeline` NPU optimizations; (2) wait for OpenVINO to add the hook, which is on the public roadmap.

When Not to Use Constrained Decoding

Three cases where the cost outweighs the benefit:

The output is naturally well-formed. If your model already produces valid JSON 99% of the time at FP16 on a well-structured prompt, the 5–20% decode-latency cost of constrained decoding may not justify the small reliability gain. Run the unconstrained version first; measure.

You need the model's natural-language reasoning. Constrained decoding forbids the model from "thinking out loud." For complex reasoning steps, run unconstrained; only constrain the final answer.

Your schema would be enormous. Constrained decoding has overhead proportional to schema complexity. If your "structured output" is really "a free-form string field with some bounded annotations," constrained decoding may not help — the constrained portion is so small relative to the free portion that you're paying overhead for very little structure.

What This Section Bought You

You should now understand:

- **Three tiers of structured output**: prompt-only, output filtering with retry, constrained decoding — pick the cheapest that works
- **NPU's greedy-only sampling makes "failures are deterministic"** — if the model failed once, it fails every time, so retries don't help
- **Quantization shifts logit landscapes** in exactly the places structured output is fragile
- **Constrained decoding masks invalid tokens at every step**, guaranteeing schema-valid output
- **Outlines + JSON schema** is the standard recipe; lm-format-enforcer and Guidance are alternatives
- **Pydantic schemas** with `Literal` discriminators give you tool-calling reliability for free
- **Costs**: 5–20% decode latency overhead, reasoning constrained, schema-rigidity risk
- **Failure modes** include tokenizer mismatches, permissive sub-schemas, missing special-token handling
- **Skip it when**: the output is already well-formed, free-form reasoning is essential, the schema is mostly free-text

Chapter 4 turns from how the agent talks to its tools to how the agent is deployed and operated in production — serving, telemetry, rollouts, and the security model that on-device deployment actually implies.

Previous: *3.3 Multi-Device Orchestration on a Single SoC* **Next:** *Chapter 4: Production Deployment & Observability*