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*

Revision #2
Created 2026-05-12 17:25:57 UTC by Admin
Updated 2026-05-12 18:59:20 UTC by Admin