4.1 Serving NPU Models with OVMS

A development-time `compile_model(...)` call is not a production deployment. Once your agent is real, it needs to survive process restarts, model updates, multiple concurrent clients, health checks, and the operations team. This section is about how to actually serve NPU-resident models — what the tooling looks like, what its limits are, and the gap between "running on my laptop" and "running for users."

OpenVINO Model Server, Honestly

The canonical serving stack for Intel NPU models is **OpenVINO Model Server (OVMS)** — Intel's C++ inference server with gRPC and REST endpoints, TF-Serving compatibility, and OpenAI-compatible `/v3/chat/completions`, `/v3/embeddings`, and `/v3/images/generations` paths. The Docker image `openvino/model_server:latest-gpu` bundles both GPU and NPU runtimes; there is no separate `-npu` image.

A reference Linux invocation for an NPU-served LLM looks like this:

```
docker run -d --rm \
  --device /dev/accel \
  --group-add=$(stat -c "%g" /dev/dri/render* | head -n 1) \
  -u $(id -u):$(id -g) -p 8000:8000 \
  -v $(pwd)/models:/models:rw \
  openvino/model_server:latest-gpu \
  --rest_port 8000 \
  --source_model OpenVINO/Qwen3-8B-int4-cw-ov \
  --model_repository_path /models \
  --target_device NPU --task text_generation \
  --cache_dir /models/.ov_cache \
  --enable_prefix_caching true --max_prompt_len 2000
```

Three things here are non-obvious. `/dev/accel` is the NPU character device on Linux. `--group-add` of the render group is required because the NPU shares Linux permission groups with the GPU. `--cache_dir` is mandatory in production — without it, the server pays the full NPU compile cost on every restart.

For an M2M-100 translation tool, you'd swap `--task text_generation` for `--task embeddings` or use OVMS's generic ML serving mode, since seq2seq translation isn't in OVMS's hardcoded task list as of mid-2026. Most teams running M2M-100 in production wrap it in a small FastAPI server rather than fighting OVMS into a non-LLM seq2seq shape.

Honest Limitations

OVMS on NPU has real limits you need to know before architecting around it. From Intel's own documentation:

Sequential execution only. "OpenVINO Model Server with NPU acceleration processes the requests sequentially… benchmarking should be performed in `max_concurrency=1`." There is no continuous batching on NPU. This is the single biggest production caveat — if you imagined the NPU server would handle dozens of concurrent users, it won't. It handles one at a time, very efficiently.

NPU LLMs must be exported with `--sym --ratio 1.0` and either channel-wise (`-1`) or `group-size 128`. Asymmetric quantization is not accepted. Most generic HuggingFace INT4 exports won't work.

Beam search is not supported on NPU. Greedy and multinomial only. If your translation tool depended on beam search for quality (and many M2M-100 deployments do, by default), you'll be running the decoder on CPU or iGPU, not NPU.

`log_probs` are not returned from NPU LLMs. This breaks most evaluation harnesses (lm-eval, HELM) that probe model behavior via token-level probabilities. You can run evals against the CPU build of the same model, but not against the NPU build directly.

FLUX image generation was unsupported on NPU as of OpenVINO 2025.2 and only partially enabled in later builds. Stable Diffusion 1.5 and SDXL UNets work, but require a static `--resolution` flag — no dynamic-resolution image gen on NPU.

Generation-request cancellation for NPU was added in OpenVINO 2025.3. Before that release, you could not cancel an in-flight NPU generation; you waited for it to finish. If you're running an older OpenVINO, factor that into your timeout strategy.

Health Probes and Metrics

OVMS exposes KServe v2 health endpoints at `/v2/health/live` and `/v2/health/ready`, plus Prometheus metrics at `/metrics`. The metrics worth alerting on:

- `ovms_inference_time_us` — actual model inference, in microseconds
- `ovms_wait_for_infer_req_time_us` — how long requests waited in the queue

- `ovms_request_time_us` — end-to-end time including serialization
- `ovms_current_requests` — in-flight request count
- `ovms_infer_req_queue_size` — backlog depth

On NPU specifically, `ovms_wait_for_infer_req_time_us` is the most telling metric. Because NPU execution is sequential, queue depth translates directly into latency for every request behind it in the queue. Watch the p95 of wait time as your operational SLO; a sustained climb means the NPU is saturated and you need to either accept higher latency, scale horizontally (more devices), or fall back to GPU/CPU for spill traffic.

When to Skip OVMS Entirely

OVMS is well-engineered, but it's not the right tool for every NPU deployment. Skip it when:

- **You have one process, one device, one model.** A locally embedded translation tool inside a desktop application doesn't need a separate inference server. Just call `compile_model` from your application and use the model directly.
- **You need non-LLM seq2seq serving.** OVMS's task abstraction is optimized for LLMs, embeddings, and image gen. Translation, ASR, OCR, and other seq2seq workloads fit awkwardly. A thin FastAPI wrapper around `OVModelForSeq2SeqLM` is often less work.
- **You need cancellation, streaming, or progress reporting** beyond what OVMS exposes. Direct OpenVINO is more flexible.
- **You're targeting a single user, not a service.** Phi Silica, Audacity, and OBS Studio all use direct OpenVINO, not OVMS — because their model lives inside one application, not behind a server.

OVMS earns its place when you have multiple client applications hitting one model, or when you need TF-Serving / KServe / OpenAI API compatibility for ecosystem reasons. For embedded NPU agents, direct OpenVINO is the simpler choice.

Lifecycle: Cold Start, Warm Path, Hot Reload

NPU model serving has three time horizons and you should design for each separately.

Cold start. First model load from disk. On Intel NPU this is dominated by compile time: 10–30 seconds for a 1B-parameter model, 30–90 seconds for a 7B. With `CACHE_DIR` set and the cache populated, subsequent cold starts drop to a few seconds. Production deployments should warm the cache as part of the container image build, not at runtime.

Warm path. Steady-state inference. This is what your benchmarks measure. For M2M-100 418M on Lunar Lake NPU 4 with INT8 weights, expect tens of milliseconds for encoder forward, and tens

to hundreds of milliseconds for decoder generation depending on output length. Real numbers vary; measure on your hardware.

Hot reload. Pushing a new model version without downtime. OVMS supports this natively (next page), but for direct OpenVINO deployments you can do it yourself: load the new `CompiledModel` in the background, atomically swap a Python reference, let the old model GC when in-flight requests finish. This is the second hidden value of `CACHE_DIR` — the new model's compile happens off the hot path.

A Sanity Checklist Before Going Live

Before shipping an NPU-served agent:

- `CACHE_DIR` is set, and the cache is populated at build time, not first user request
- The model is exported with the NPU-compatible quantization recipe (`--sym --ratio 1.0` for INT4 LLMs)
- Static shapes are fixed at compile time, not inferred from request data
- Prometheus metrics are scraped and dashboards exist for wait-time p95 and queue depth
- Health probes are wired to your orchestrator (Kubernetes, systemd, whatever)
- You've tested process restart and confirmed cache hits cut cold-start to a tolerable budget
- You have a fallback path (CPU or iGPU) for when the NPU is unavailable or saturated
- You've measured actual latency on actual hardware — not extrapolated from spec sheets

The next section turns to the harder problem: observing what's actually happening on the NPU when things go wrong.

Next: *4.2 Telemetry: What Works, What Doesn't, and What's Missing*

Revision #2
Created 2026-05-12 17:26:37 UTC by Admin
Updated 2026-05-12 18:59:32 UTC by Admin