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

You can't operate what you can't observe. NPU agents have a harder observability story than CPU- or GPU-bound workloads — partly because the hardware is newer, partly because vendor tooling lags, partly because some of the telemetry you'd expect simply isn't exposed. This section catalogues what's actually available, where the gaps are, and how to work around them.

Per-Layer Profiling: The Hammer

OpenVINO has built-in per-layer profiling that works on the NPU. Turn it on with `PERF_COUNT: True`:

```
core.set_property("NPU", {"PERF_COUNT": True})
compiled = core.compile_model("model.xml", "NPU")
req = compiled.create_infer_request()
req.infer(input_tensor)
for info in req.get_profiling_info():
    print(info.node_name, info.status,
          info.real_time.total_seconds() * 1e6, "us")
```

This is the closest you'll get to a flamegraph for NPU inference. It tells you, per operator, how much time was spent and whether the op actually executed or was optimized away during graph compile.

The caveat that bites people: for *fused* LLM graphs, `real_time` often returns 0 with `Status=NOT_RUN`, because the entire transformer block was compiled into a single super-kernel and the per-op counters don't trace inside it (this is documented in issue #24885). The OpenVINO docs themselves note that perf counters don't reflect queue time. Use these counters for **relative** comparisons between models or between optimization passes — not for absolute latency attribution. If you need a single end-to-end number, use `benchmark_app`:

```
benchmark_app -m model.xml -d NPU -hint latency -niter 1000 -pc
```

`-pc` prints per-counter stats; `-niter 1000` runs enough iterations to smooth out noise.

VTune for NPU

Intel's VTune Profiler added an `npu` analysis type in version 2024.1. It's the most powerful NPU profiler available, surfacing per-core SHAVE DSP utilization, MAC array occupancy, memory bandwidth, and queue wait times. The catch: it has a steep setup, requires specific driver versions, and Linux support is genuinely rough — the Chips and Cheese deep-dive on Meteor Lake NPU complained "I only got the NPU profiling mode to work exactly once."

Use VTune when you're seriously optimizing an NPU model — picking between quantization schemes, validating that fusion happened, debugging unexpected slowness. Don't reach for it for routine application monitoring.

OS-Level Utilization

This is where the story gets uneven.

On Windows, the Task Manager NPU graph (labeled "Intel AI Boost") has been present since Windows 11 23H2. **Windows 11 Insider build 26300.8142** (2025) added per-process NPU columns and "NPU Engine" tracking — the first build with per-process NPU utilization visible. But there is **no public API** for third-party apps to query per-process NPU%; Microsoft engineers have confirmed that the formula behind Task Manager's NPU graph is not released. You can *detect* the NPU through DXCore (the relevant flag is `DXCORE_ADAPTER_ATTRIBUTE_D3D12_CORE_COMPUTE` set with `D3D12_GRAPHICS` not set), but you cannot read its utilization programmatically.

On Linux, there's no official `intel_npu_top` and Intel has not committed to building one. The community has filled the gap with several tools:

- **`nputop`** (Rust, parses sysfs) — the most popular community option
- **`intel-npu-top`** (Python wrapper) — simpler alternative
- **Resources 1.10.2** (GNOME app, March 2026) — added NPU core-frequency monitoring; pre-installed in Ubuntu 26.04

For production fleet monitoring, the practical answer is: parse `/sys/class/drm/renderD*/device/npu/` yourself, the same way these community tools do.

The Power Telemetry Gap

You'd think a chip purpose-built for "performance per watt" would expose its watts. It doesn't. **The NPU is not a separate RAPL domain on Intel Core SoCs.** It sits inside the System Agent power envelope. The HWiNFO maintainer wrote on his forum: "According to Intel, NPU power monitoring is (currently) not possible. I have raised this question several times to them but no commitment whether it will be possible."

This means:

- You cannot directly attribute power to NPU vs CPU vs iGPU
- Marketing claims of "1.5 W on NPU" (like Microsoft's Phi Silica numbers) are vendor-measured, not user-verifiable
- Chips and Cheese's ~7 W typical NPU figure on Meteor Lake was *inferred* from System Agent RAPL deltas, not read from a direct counter

For agents that need to make routing decisions based on power state (e.g., "on battery, prefer NPU; on AC, prefer iGPU"), you'll have to use proxy signals: AC vs battery, current package temperature, total RAPL energy. None of them tells you specifically what the NPU is drawing.

Intel Power Gadget reached end-of-life in 2023; the recommended alternatives are PresentMon and Intel's oneAPI Application Performance Snapshot, both of which give you indirect views.

Application-Level Telemetry

Since the OS layer is gappy, most production NPU agents instrument heavily at the application level. The metrics that consistently pay off:

End-to-end request latency, broken down by stage. For a translation tool, that means: tokenize_time, npu_encode_time, decoder_time, detokenize_time, total_time. Add these as histograms in Prometheus or OpenTelemetry. The breakdown tells you whether slow user-perceived translation is the NPU, the tokenizer, or the orchestrator.

Cache hit rate for prefix caching and model caching. A `CACHE_DIR` hit ratio under 80% means your prefix isn't actually fixed (Chapter 2.2) or your model files are changing between deployments.

NPU compile time as a separate metric from inference time. Spikes in compile time signal driver or runtime changes — the kind of thing you want to catch in canary builds, not production.

Fallback rate. If your agent falls back from NPU to CPU/iGPU on errors, the fallback rate is your single most important reliability signal. Healthy is near zero. Climbing is a driver issue, a model issue, or a hardware issue you need to investigate before it cascades.

Tool selection distribution. Logging which tool the agent chose for each request tells you whether the agent is over- or under-using each tool — and which tools are actually earning their NPU residency.

A Diagnostic Tree

When NPU performance regresses in production, work through this list before panicking:

1. **Did the OpenVINO version change?** Release notes between minor versions document NPU-specific regressions; always check.
2. **Did the NPU driver change?** Windows drivers update through Windows Update; track the version in your telemetry.
3. **Is `CACHE_DIR` populated?** A cleared cache turns a 500ms warm load into a 30s cold compile.
4. **Did the model file change without the cache being invalidated?** Pre-baked blobs are not guaranteed across driver versions.
5. **Is the wait-time p95 climbing?** Queue saturation on a sequential NPU. Add a fallback path or scale.
6. **Are fallback rates elevated?** Something is failing on NPU and silently degrading. Inspect logs.

What You Don't Get

It's worth being explicit about what's *not* available, so you don't waste time looking:

- **No public API for per-process NPU utilization** on Windows
- **No official Linux NPU monitoring tool** from Intel
- **No NPU-specific power counter** anywhere
- **No per-op profiling inside fused LLM graphs** on NPU
- **No standard streaming-token latency metric** from OVMS (you build it yourself)
- **No `log_probs` from NPU LLMs**, breaking many eval harnesses

You're going to instrument more, more carefully, than you would for CPU- or GPU-based workloads. Plan for that operational cost from the start.

What This Section Bought You

NPU observability is real, but uneven. The summary:

- **Per-layer profiling** via OpenVINO `PERF_COUNT` is the workhorse — best for relative comparisons
- **VTune NPU analysis** is the heavy artillery — use sparingly
- **OS-level utilization** is partial on Windows and DIY on Linux
- **Power telemetry doesn't exist** at NPU granularity; you'll use proxy signals
- **Application-level telemetry** is where you get reliable signal — instrument latency by stage, cache hits, compile time, fallback rate, tool distribution
- **A diagnostic tree** beats panic when production regresses

The next section closes Chapter 4 by looking at the harder questions of A/B testing, canaries, and hotswaps — once you can see what's happening, what do you do with that visibility?

Previous: *4.1 Serving NPU Models with OVMS* **Next:** *4.3 A/B Testing, Canaries, and Hotswaps*

Revision #2
Created 2026-05-12 17:27:19 UTC by Admin
Updated 2026-05-12 18:59:39 UTC by Admin