4.3 A/B Testing, Canaries, and Hotswaps

Models drift. Drivers update. Quantization schemes change. The NPU you tested against in February is not the NPU your users have in November. Shipping an NPU-resident agent is not a one-time event — it's a continuous negotiation between your release process and a hardware stack that's still evolving rapidly. This section is about how to make that negotiation safe.

Why A/B and Canary Matter More on NPU

For cloud APIs, A/B testing is a luxury that pays for itself in measurable quality lifts. For NPU agents, **it's borderline mandatory**, because the failure modes are different from cloud:

- **Driver updates can change model output.** Intel's OpenVINO release notes literally document this: 2025.3 noted "miniCPM3-4B model is inaccurate with NPU driver 32.0.100.4239." 2026.1 noted "Distilled SDXL Unet result fixed to be same with CPU and GPU." These are vendor-acknowledged behavioral changes triggered by versions you don't control.
- **Quantization changes are not value-preserving.** Re-exporting M2M-100 from INT8 to INT4 changes BLEU. The arXiv paper on translation accuracy and quantization found INT4 NLLB preserves "high levels of accuracy and fluency" on average, but with measurable drops on low-resource pairs. You need eval coverage that catches these.
- **Fallback paths silently change quality.** If your agent falls back from NPU to CPU due to a driver bug, output stays *functionally* correct but may differ in subtle ways the user notices.

The combined effect: an NPU agent that "works in dev" can subtly regress in production from upstream changes you didn't make. A/B testing is your safety net against changes you don't fully control.

Patterns That Work On-Device

There is **no built-in traffic-splitting framework** for NPU agents. OVMS supports model versioning (numeric subdirectories like `models/translate/{1,2,3}/`) with a `model_version_policy` controlling how many versions stay loaded, and clients can pin via `/v2/models/translate/versions/3/infer`. But routing traffic between versions is your problem, not

OVMS's.

Three patterns recur in production NPU deployments:

Feature flags wrapping `device_name`. The simplest A/B: a runtime flag chooses NPU or CPU. Useful for testing the *device choice* itself, less useful for testing model variants.

```
device = "NPU" if user.in_canary_cohort else "CPU"
compiled = core.compile_model(model, device)
```

Shadow inference. Run the new model in a fire-and-forget thread alongside the production model, diff outputs, log discrepancies. This is **non-optional** for NPU LLMs because the release notes literally document quantization-induced output drift. Shadow inference catches drift before you cut traffic over.

```
async def serve(request):
    result = await prod_model(request)
    asyncio.create_task(shadow_compare(request, result))  # fire and forget
    return result

async def shadow_compare(request, prod_result):
    canary_result = await canary_model(request)
    if not outputs_equivalent(prod_result, canary_result):
        log_divergence(request, prod_result, canary_result)
```

Per-request routing by user-id hash with a kill switch. Hash the user ID, route X% to canary, keep a flag that can be flipped to 0% instantly. The standard cloud canary pattern, ported to local NPU deployments.

```
def select_model(user_id, canary_fraction=0.05):
    if canary_kill_switch.is_set():
        return prod_model
    h = hash(user_id) % 10000
    return canary_model if h < canary_fraction * 10000 else prod_model
```

A/B in Memory-Constrained Environments

Here's where NPU diverges from cloud A/B sharply: **each enabled model version compiles a separate `ov::CompiledModel`** and consumes its own slice of NPU memory. On a 16 GB Lunar Lake

laptop, having two versions of M2M-100 1.2B loaded simultaneously is feasible (about 2.5 GB combined at INT4). Having two versions of Phi-3.5-mini loaded simultaneously is a tight fit. Having three versions of anything substantial is not happening.

Practical implications:

- **Canary cohort sizes can't be tiny.** If you're loading the canary model on every device, every user is paying memory cost — there's no "5% of fleet" the way there is in cloud.
- **Cold-swap A/B is often more memory-efficient than warm-coexist.** Load whichever model the user's cohort needs at session start, unload the other.
- **The cost of A/B is paid by users**, not by your inference server. Memory pressure shows up as slower app startup or other apps OOM-killing. Tread carefully.

For server-side deployments where multiple devices serve a fleet, the cloud canary pattern works fine: route 5% of requests to a separate physical machine running the canary build. This sidesteps the memory contention entirely.

Hotswap Without Downtime

Pushing a new model version without restarting the process is straightforward on OpenVINO if you respect the cache and the lifecycle:

1. **Download the new IR** (`model.xml` + `model.bin`) to a staging directory. Verify checksums.
2. **Compile in the background** with `CACHE_DIR` set: `new_compiled = core.compile_model(new_path, "NPU", {"CACHE_DIR": cache})`.
3. **Atomically swap** a Python reference (or atomic-replace a service pointer): `self.model = new_compiled`.
4. **Let the old `CompiledModel` GC** when in-flight requests finish.
5. **Keep N-1 versions in memory** for instant rollback.

OVMS automates step 3 — it auto-detects new model versions in the repository every `--file_system_poll_wait_seconds` (default 1 second), and `POST /v1/config/reload` triggers a full reload without restarting. Honor the **Windows gotcha**: OpenVINO mmaps IR files by default, so on Windows the old `.xml` can't be deleted until unloaded. Disable mmap with `--plugin_config '{"ENABLE_MMAP":"NO"}'` if you're hotswapping on Windows.

Cache Compatibility Across Updates

Here's a subtle production trap: **OpenVINO blob compatibility is not guaranteed across versions**. The compiled NPU binary you cached with OpenVINO 2025.3 may not load correctly under 2026.1, and the driver may have changed too. Three rules:

- **Never ship pre-baked NPU blobs** to production. Always compile on-device, on first run.
- **Invalidate** `CACHE_DIR` when OpenVINO or NPU driver versions change. Detect this at startup and clear the cache deliberately.
- **Cache invalidation costs cold-start time**, so plan to do it during a maintenance window or app update, not during peak hours.

The Driver Update Problem

NPU drivers update through Windows Update on Windows and through your package manager on Linux. You don't fully control when they arrive. Two defensive patterns:

Pin the driver version in CI. Document the minimum NPU driver version your agent has been tested against. Refuse to start (with a clear error) if the runtime driver is older. The Intel NPU plugin returns version info through OpenVINO's device properties; check it at startup.

Pre-flight model validation. When the agent starts, run a small canonical test through the NPU and verify output matches a known-good result. Fail loudly if output is wrong — that catches the "driver update silently changed output" case before users see it.

```
def validate_npu_pipeline(model, test_input, expected_output, tolerance=1e-3):
    actual = model.infer(test_input)
    if not within_tolerance(actual, expected_output, tolerance):
        raise RuntimeError(
            f"NPU output divergence detected. "
            f"Driver version: {get_npu_driver_version()}, "
            f"OpenVINO: {ov.__version__}")
```

This isn't paranoia. Intel's own release notes call out behavioral changes triggered by driver versions. Treat the NPU like a third-party dependency that updates without warning.

Wrapping Up Chapter 4

Production NPU deployment is harder than the marketing implies — but it's also tractable if you respect the realities:

- **OVMS is the canonical server** but has real limits: sequential execution, INT4-symmetric only, no beam search, no `log_probs`
- **Telemetry is uneven** — instrument heavily at the application layer because the OS layer is gappy
- **A/B testing isn't optional** when drivers and quantization can subtly change output
- **Hotswap is straightforward** if you respect `CACHE_DIR`, version atomicity, and the Windows mmap gotcha

- **The driver is a third-party dependency** that updates without your consent — pre-flight validation is your safety net

Chapter 5, the closer, takes us to the field: case studies of NPU agents that actually shipped, what they did right, what they got wrong, and what generalizes from their experience to yours.

Previous: *4.2 Telemetry: What Works, What Doesn't, and What's Missing* **Next:** *Chapter 5: Real-World Case Studies & Best Practices*

Revision #2
Created 2026-05-12 17:28:01 UTC by Admin
Updated 2026-05-12 18:59:46 UTC by Admin