2.2 KV Cache Engineering: Reuse, Eviction, and Prefix Sharing

The distinction between KV cache (what you keep in memory) and KV cache bandwidth (what you stream per token) is subtle and worth being precise about, because it sets the operational window for what an agent can do in real time. This section descends into the implementation details: what does KV cache engineering look like in practice, and where do the OpenVINO APIs and caching layers fit?

Stateful KV Caching: In-Memory and On-Disk

OpenVINO's `LLMPipeline` (for decoder-only models) and the older OpenVINO 2025.3 GenAI interface expose KV caching through **stateful models** that hold KV state across multiple `infer()` calls.

A stateless forward pass recomputes the full context on every token:

```
outputs = model(prompt_tokens + [new_token])  # Expensive at each step
```

A stateful forward pass reuses KV from the previous step:

```
# First call (prefill): starts the chat session, returns KV state internally
outputs = model.start_chat(prompt_tokens)

# Subsequent calls (decode): feed only the new token, read cached KV
for step in range(num_steps):
    outputs = model.generate_next(new_token)
    # The model's internal KV state grows: [1, 1, step+1, head_dim] for self-attention
    # Each step is O(1) in context length, not O(seq_len)
```

This is exposed in OpenVINO via `LLMPipeline.start_chat()` and `LLMPipeline.finish_chat()`, or via the lower-level stateful pipeline API that manages the KV variable allocation.

On-disk KV caching is a feature of OpenVINO 2025.4+: the prefix cache (Chapter 2.2's cached KV across different prompts with shared prefixes) can be memory-mapped to disk, reducing hot DRAM footprint. This is not the same as KV cache spilling; it's a deliberate optimization for scenarios with many similar prompts (e.g., RAG where the retrieval context is shared).

The Three Layers of Caching

OpenVINO has three distinct caching mechanisms that developers often confuse:

1. Model caching (CACHE_DIR). The compiled blob (the IR XML + weights compiled to NPU bytecode) is written to disk on first compilation, then loaded from disk on subsequent runs. This is handled by setting `CACHE_DIR` environment variable or via `core.set_property("CACHE_DIR", path)`. Runtime: saves 30–60 seconds on cold start, costs ~1–3 seconds on warm start (load from disk, validate, run). Scope: global per model, not per-session.

2. KV cache (stateful model state). The key-value cache for attention is held in memory as model variables. Managed via `model.start_chat()` and `model.finish_chat()` for `LLMPipeline`, or directly via `InferRequest` variable state for lower-level APIs. Runtime: O(seq_len × head_size) memory per layer, amortized O(1) per token decode. Scope: per-session (one chat session = one KV state buffer).

3. Prefix caching (NPUW_LLM_ENABLE_PREFIX_CACHING). A newer feature (2025.4+) that caches the KV of common prompt prefixes across different requests. If you make multiple requests that share a long context prefix (e.g., system prompt + retrieved documents), the KV for the prefix is computed once and reused. Mechanism differs per device: on CPU/GPU it uses copy-on-write; on NPU it's a different path through the compiler. Runtime: saves recompute on shared prefixes, costs extra memory for the cache table. Scope: global per model (shared across all sessions).

These are orthogonal. You can have model caching (bytecode on disk) + KV caching (current session's attention memory) + prefix caching (shared prompt prefixes across sessions), all at once. The confusion arises because they all have "cache" in the name and all improve performance, but at different scopes.

KV Cache Precision and Quantization

The KV cache is almost always kept in **FP16 or higher precision** on NPU, even if weights are INT4 or INT8. Why? Because the attention mechanism (the softmax in particular) is sensitive to numerical precision; quantizing the KV to INT8 often causes noticeable degradation in output quality, particularly on longer contexts where accumulated rounding error matters.

The exception is **NF4 weights + FP16 KV** (Lunar Lake NPU 4 only, 2025.3+), where the weights are NF4 and the KV is held at FP16. This is a documented combination; going further (e.g., INT4 KV) is not validated and likely to cause accuracy loss.

For M2M-100 1.2B at 128 tokens:

- Weights at INT4: 600 MB
- KV cache at FP16: 25 MB
- Total hot memory: ~625 MB (fits comfortably)

For an 8B model at 2K context:

- Weights at INT4: 4 GB
- KV cache at FP16: ~400 MB (rough estimate for 8B with GQA)
- Total: ~4.4 GB (fits within Lunar Lake's 16 GB, but now memory bandwidth contention becomes real)

OVMS (OpenVINO Model Server) and Sequential Execution

A caveat from the documentation: **OpenVINO Model Server (OVMS) with NPU Stateful models has a "process requests sequentially" policy**. Some readers interpret this as "the NPU hardware can only process one request at a time." That's misleading.

What it actually means: the OVMS scheduler for NPU Stateful servables is currently single-threaded, so requests are queued and handled one at a time. The NPU hardware itself supports multiple concurrent inference requests (via async `InferRequest` in the native API), tile-level parallelism, and frequency scaling. The sequential policy is a **scheduler choice in OVMS**, not a hardware limitation.

If you're using the native OpenVINO Runtime API directly (not OVMS), you can use async requests and parallelize inference. OVMS is the higher-level serving layer; if you're building an agent system in-process (which is typical for edge/on-device agents), you're likely using the Runtime API and don't hit this constraint.

KV Cache Memory Lifecycle

For a long-running agent that cycles through multiple requests (interact with user, call a tool, observe, reason, repeat), KV cache management matters:

```
# Pseudocode for agent loop
model = ov.LLMPipeline(...)
for i in range(num_steps):
    # Prefill: prompt grows with accumulated observations
    outputs = model.start_chat(accumulated_prompt)  # Allocates KV state
```

```
    for j in range(decode_tokens):
        # Decode: uses cached KV
        outputs = model.generate_next()

    # Finish: release KV state
    model.finish_chat()  # Clears the KV buffer

    # Between steps: observations are appended to accumulated_prompt
    # accumulated_prompt grows; KV cache is discarded and recreated on next prefill
```

At each `start_chat()`, a fresh KV allocation is made. If your accumulated prompt has grown to 2K tokens, the KV allocation is 2K-sized and you're committed to that footprint until `finish_chat()`. If the next step's prompt is 3K tokens, a new 3K allocation is made.

For long-running agents, this means you can't accumulate unbounded history within a single KV buffer; you have to either:

- Truncate the context window (recent-only history, myopic agent)
- Use external long-term memory (vector store) and retrieve into fresh prefill (stateless from KV perspective, but stateful in application logic)
- Use sliding-window KV (drop oldest tokens, recompute if needed)

Implications for M2M-100 Deployment

M2M-100 is an encoder-decoder, so the KV lifecycle is:

1. **Encoder prefill**: source text is encoded once, encoder KV is computed and held for the entire decode phase
2. **Decoder decode**: new target tokens are generated; decoder self-attention KV grows, cross-attention KV is reused from encoder

The encoder KV doesn't get reused across multiple different source sentences; it's specific to that encode-decode pair. If you have a batch of translation requests, each one brings its own encoder KV. This is why batching M2M-100 (or any seq2seq) is awkward on NPU — you can't trivially share encoder KV across different inputs.

What This Section Bought You

You should now understand:

- **Stateful KV caching** via `start_chat()` / `finish_chat()` amortizes prefill cost across decode steps
- **Three orthogonal caching layers**: model cache (bytecode), KV cache (session state), prefix cache (shared prefix KV)
- **KV cache is kept at FP16+**, even when weights are INT4, for numerical stability
- **OVMS sequential execution** is a scheduler policy, not a hardware limit; native Runtime API supports async
- **KV cache allocation commits to context length** at `start_chat()` time; unbounded history requires external memory
- **M2M-100's encoder KV is per-request**, not shared across requests — this is why seq2seq batching is complex
- **Long-term agent memory lives outside the model** — KV cache is working memory only

The next section applies all of this to the agent's reasoning loop: given bounded context and bounded KV cache, what reasoning architectures actually work?

Previous: *2.1 Context Windows and the Memory Wall* **Next:** *2.3 Reasoning Loops Under Constraint*

Revision #4
Created 2026-05-12 15:59:10 UTC by Admin
Updated 2026-05-12 19:17:29 UTC by Admin