Agent State & Decision-Making on Constrained Hardware

Managing agent context and memory within NPU limits. Efficient reasoning loops for low-latency inference. Token budget strategies and context windowing. Caching and KV optimization for repeated queries.

2.1 Context Windows and the Memory Wall

The agent's state — what it remembers from past steps and what it uses to make the next decision — is the bridge between hardware constraints and agent behavior. This section is about the memory wall: why it exists, what it means in numbers, and how to budget for it in the agent loop.

The two key state mechanisms are **KV cache** (the prefill and decode phases' attention memory) and **context window** (the prompt that feeds the next prefill). They're distinct costs with different scaling properties, and conflating them is a common design mistake.

The KV Cache and Its Footprint

The KV (key-value) cache is the core optimization of autoregressive LLM inference: instead of recomputing the attention keys and values for every token position on every decoding step, you compute them once and keep them in memory. On the second token, you use the KV from token 1 plus the new KV for token 2. On the third token, you use KVs from tokens 1–2 plus the new one. This is why decode is so much faster than prefill — you're amortizing the work.

The KV cache lives in DRAM and is dimensioned by **[batch_size, num_heads, seq_len, head_dim]**. For a typical transformer:

- `batch_size`: 1 on NPU (Chapter 1.3)
- `num_heads`: 16 (common)
- `seq_len`: grows from 1 to context_length as you decode
- `head_dim`: 64 (common)

Per-token KV cache footprint = batch × num_heads × head_dim × 2 (K + V) × dtype_bytes.

For M2M-100 1.2B (16 heads, 64 head_dim) at FP16 (2 bytes):

- Per token per layer: 1 × 16 × 64 × 2 × 2 = **4,096 bytes = 4 KB per token per layer**
- M2M-100 1.2B has 24 encoder + 24 decoder layers; the decoder keeps **48 × 4 KB = 192 KB per token**
- At 128-token context: 128 tokens × 192 KB = **24.6 MB per inference batch**

But M2M-100 is encoder-decoder, so there's a second KV cache: the encoder output, which the decoder's cross-attention reads at every step. The encoder KV is computed once (during prefill) and reused throughout decode, so it doesn't grow with seq_len, but it's identical in size to the self-

attention KV of the decoder at any given encoder context length.

Full M2M-100 decoder KV footprint at T=128 token context and encoder context L=128:

- Self-attention KV: 24 layers × 128 tokens × 4 KB = **12.3 MB**
- Cross-attention KV (encoder output): 24 layers × 128 source tokens × 4 KB = **12.3 MB**
- **Total: ~25 MB per sequence** (FP16)

Now compare to **Phi-3-mini-3.8B**, which uses GQA (grouped-query attention) with 8 KV heads instead of 16:

- Per token per layer: 1 × 8 × 64 × 2 × 2 = **2,048 bytes = 2 KB per token per layer**
- 32 layers × 2 KB = **64 KB per token**
- At 128-token context: 128 × 64 KB = **8.2 MB** (before any encoder overhead)

So Phi-3-mini saves 3× on KV footprint per token, because it halves the KV head count. M2M-100 has full MHA and pays the bandwidth price.

The Attention Wall

The attention wall is simple to state: **at some context length, the KV cache's bandwidth demand exceeds what the NPU can sustain**. On Lunar Lake with 136.5 GB/s platform bandwidth, and given the 18% utilization we saw in Chapter 1.3, the per-NPU effective bandwidth is roughly 136.5 × 0.18 ≈ **~25 GB/s available**.

For M2M-100 decoder at FP16:

- 192 KB per token (self + cross attention, 48 decoder+encoder layers)
- At **6.10 tok/s**: 192 KB × 6.10 = **~1.17 MB/s** of KV cache bandwidth

This is well below the 25 GB/s ceiling, so the M2M-100 KV cache isn't the bottleneck yet. The wall appears at much larger context lengths or larger models.

The working hypothesis from Chapter 2.1 is that the KV cache wall appears somewhere between 2K and 8K tokens for typical 8B models on Lunar Lake, depending on model architecture. Intel's validated 8K context "preview" on Lunar Lake is right at that edge. The wall doesn't mean you *can't* have 8K; it means you're committing to recompute, sliding windows, or multi-GPU distribution to stay above a latency floor.

Context Window vs. KV Cache

A critical distinction: **context window is what the model can attend to; KV cache is what you must keep in memory**.

For a decoder-only model like Llama 3 70B:

- Context window: 8K tokens
- KV cache for full context: 70B parameters × 16 heads × 64 head_dim × 2 (K+V) × 2 bytes × 8K tokens ÷ (70B total params) = roughly 70–80 GB for a single sequence at full context.

That doesn't fit on a single Lunar Lake. The roofline says: if you want 8K context with 70B, you compress the model (quantize), shard it (multi-GPU), or use a sliding window (throw away old context).

For M2M-100 1.2B at 128 tokens, KV cache is 25 MB, which fits easily. At 2K tokens, it's about 400 MB (2K ÷ 128 × 25 MB). At 8K, it's 1.6 GB — still under the 4–8 GB weight budget, but now you're committing real DRAM.

The practical implication: **the agent's working-memory window (what it can see in a single prompt) is bounded by KV cache size, not by model capability**. An 8B model trained on 8K context can't actually use that context on NPU if the KV cache doesn't fit.

Implications for Agent Design

Three consequences flow from this:

1. Bounded context is a feature, not a limitation. If your agent loops (agent thinks → acts → observes), and the context window is fixed at, say, 1K tokens, then the agent's working memory is fixed. Every observation older than 1K tokens falls off the window. This forces a design choice: either the agent uses only recent observations (myopic), or long-term memory lives outside the model in a vector store or database (Chapter 2.3).

2. KV cache reuse is precious. In the M2M-100 pattern (encoder-decoder), the encoder is computed once; the KV cache is reused throughout decode. In a chatbot where the user query is short but the response is long, this is efficient. In a long-conversation scenario where both sides grow, every new user message requires a re-encode. This is why copy-on-write KV cache techniques (keeping separate buffers for user messages that don't change) matter.

3. The sliding-window technique (Phi Silica's N=64 approach from Chapter 1.3) is a deliberate trade: throw away the oldest tokens' KVs to free DRAM, then recompute them if you need to backtrack. On NPU where compute is cheaper than bandwidth (relatively speaking), this is a valid trade. On GPU where compute is expensive relative to DRAM, it usually isn't.

How Intel's "8K Validated Preview" Works

Intel's announcement that Lunar Lake supports "8K context" (Chapter 1.2's static-shape discussion) is narrowly true: the compiler can emit a static-shape graph for 8K, and it runs without crashing. What's not guaranteed is latency.

The 8K window likely uses chunked prefill (process 1K chunks at a time) and either sliding-window KV for decode or hybrid compute-cache layering (let the CPU assist with KV management). The "preview" designation means it's not validated for production; the team is still characterizing it.

For agent design, treat 8K as the ceiling, not the target. A 1K–2K working memory is reliable; 4K–8K requires careful modeling and testing; beyond 8K requires either multi-GPU or architectural workarounds.

What This Section Bought You

You should now understand:

- **KV cache footprint scales with [seq_len, num_heads, head_dim, layers, dtype]** — M2M-100 1.2B at 128 tokens is ~25 MB
- **Full MHA (M2M-100) vs. GQA (Phi-3-mini) creates a 3× KV bandwidth difference** — attention architecture is destiny
- **The attention wall appears at 2K–8K tokens** on Lunar Lake depending on model size
- **KV cache growth is the per-token latency problem**; context window is the per-prompt problem
- **Encoder KV reuse** (encoder-decoder models) is a structural advantage
- **Sliding-window KV** trades compute for bandwidth — a valid move on NPU
- **8K context on Lunar Lake is validated-preview, not production**; design for 1K–2K working memory
- **Long-term memory for the agent lives outside the model** — in SQLite, vector stores, or filesystems

The next section turns to the agent's reasoning loop: given bounded context and bounded KV cache, what patterns actually work for multi-step agents?

Previous: *Chapter 1: Foundations* **Next:** *2.2 KV Cache Engineering*

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*

2.3 Reasoning Loops Under Constraint

Chapter 2 closes here. We have a model that fits, weights we can stream, KV state we can manage, and decode at roughly 6–20 tok/s. The question this section answers: given that decode budget, what reasoning architectures actually work? The naive answer — bolt a ReAct loop on top and let the agent think — collides with the latency ceiling in a way worth being specific about.

The Three Reasoning Architectures

Three patterns dominate agent design, and they sort cleanly by NPU compatibility:

Single-shot. One prompt, one response. No loop. The agent reads the input, produces the output, done. Translation is the canonical single-shot task: source sentence in, target sentence out. The cost is one prefill plus one decode. Phi Silica's Click to Do affordances are single-shot. **This is the NPU-native pattern.**

Plan-then-execute. The model produces a plan once, then executes the plan deterministically (often without further model calls, or with a small number of pre-determined model calls). For a translation assistant: "rewrite this paragraph for a teenage audience and translate to French" decomposes to (1) rewrite via Phi-3.5-mini, (2) translate via M2M-100. The plan is one LLM call; the execution is a fixed pipeline. Two model calls total, predictable latency.

ReAct (Reason + Act). The model alternates between thinking and tool-calling in a loop, with each iteration informed by the last. The hallmark is that the *number of iterations is not known in advance*. This is the dominant pattern for cloud agents and the one developers reach for by default. **It's also the pattern that NPU latency budgets cannot afford.**

The ReAct Latency Budget

Let's price out a 5-step ReAct loop on Intel Core Ultra NPU, anchored on Chapter 1.3's two published benchmarks.

Assumptions: 512-token context per step (prompt grows as the loop accumulates), 64-token decode per step (the agent's "Thought / Action / Observation" turn). Using Llama 2 7B at MLPerf's TTFT-1.09s/128-tok-prompt and DeepSeek-Distill-Llama-8B's 163 ms/token decode as the conservative anchors:

Component	Value	Source
TTFT, ~128-token prompt	1.09 s	MLPerf Client v0.6
TTFT extrapolated to 512-token prompt	~4 s	linear-ish
ITL per decode token (8B INT4)	163 ms	OpenVINO Model Hub
Decode 64 tokens	10.4 s	computed
One ReAct iteration	**~14–15 s**	extrapolated
5 iterations	**~70–75 s**	extrapolated

On the same SoC's iGPU (12.8 tok/s, ~78 ms/token): one iteration ≈ 7 s, five iterations ≈ 35 s.

A 5-step ReAct agent at this context size on Intel NPU sits in the 60–90 second range — usable for offline summarization, marginal for chat, infeasible for interactive autocomplete. Stretching the loop to 10 steps doubles it. ReAct's behavior of growing the context monotonically with each step makes it worse over time, not better, because every iteration's prefill takes longer than the last.

These numbers are extrapolations from published single-call benchmarks, not measurements of ReAct loops. We flagged in Chapter 1.3 that Intel and Microsoft have published almost nothing about multi-step agents on NPU. Treat the table as the right order of magnitude, not as a precise SLA.

Why Single-Shot Wins on NPU

The structural reasons single-shot translates to NPU and ReAct doesn't:

Each ReAct step pays full TTFT. The prefill is the compute-bound, MAC-array-heavy phase; on NPU it's relatively fast per-prompt, but you do it `N` times per loop instead of once. A 5-step ReAct burns 5× the TTFT of an equivalent single-shot.

Context grows monotonically. Step 1's prefill is short. Step 5's prefill includes everything that came before. The TTFT cost rises through the loop. Chunked prefill on NPU helps, but doesn't fix the issue: each chunk costs constant time, and step 5 has more chunks.

Cold-cache pressure increases. The KV state from step 1 has to be valid at step 5 — which works fine within `LLMPipeline.start_chat()` but means the state-variable allocation must accommodate the full final context. You commit to the worst-case footprint up front.

Greedy-only hurts most here. On NPU's static pipeline, no beam search. ReAct's "Thought" outputs are exactly the kind of free-form text that benefits from beam-4 sampling diversity. Greedy ReAct tends to fall into repetitive loops.

The cumulative effect: ReAct on Intel NPU magnifies the very constraints that NPUs are worst at. It's the wrong architecture for the hardware.

What to Do Instead

Prefer single-shot. If your task can be reduced to one prompt and one response, do that. Translation is single-shot. Summarization is single-shot. Tone-rewrite is single-shot. "Explain this code" is single-shot. The cloud-agent culture's enthusiasm for ReAct has obscured how many useful tasks don't actually need a loop.

Use plan-then-execute when you need composition. A planning call decides the structure; deterministic code runs the plan. The planning model needs to produce structured output (JSON, XML), which works fine in single-shot. The execution is fixed-cost, and any individual sub-call can hit its own device — the plan can route one sub-task to NPU, another to iGPU.

Use the cascade pattern for triage. A tiny model on NPU decides whether the request needs the heavy model. The cheap path is sub-second; the expensive path is the budget you'd already pay for a single-shot. Worst-case latency is the heavy-model latency, not the heavy-model latency *times the number of ReAct iterations*.

When you genuinely need ReAct, run it on iGPU. The 2.1× speedup from Chapter 1.3 turns 75-second NPU ReAct into 35-second iGPU ReAct. Still slow by cloud standards; in budget for offline workflows like document analysis. The NPU's role becomes drafting and triage; the iGPU does the reasoning loop.

Tighten context aggressively. Every byte you can prune from the running prompt is bandwidth you don't pay for at every step. The Phi Silica architecture's N=64 sliding window over context is an aggressive version of this — most of the time you don't need everything in scope.

Working vs Long-Term Memory

The reasoning loop's *state* — what the agent remembers across steps — splits into two regimes.

Working memory is what's in the prompt this turn. On NPU it's bounded by `MAX_PROMPT_LEN`. Generous on chunked-prefill-capable models (up to 8K validated on Lunar Lake); tighter on encoder-decoder seq2seq like M2M-100. Working memory is fast (it's in the model's attention window) and ephemeral (it doesn't persist across sessions).

Long-term memory lives outside the model — in a SQLite database, a vector store, a key-value cache, a local filesystem. It's persistent and unbounded in size, but accessing it costs an explicit retrieval step before the next prompt. For NPU agents, **long-term memory needs to be local**, which means it's a few milliseconds away and orders of magnitude cheaper than another NPU forward pass.

The pattern that works well on NPU: aggressive working-memory pruning (small context, small TTFT), with retrieval into a local vector store between turns. The vector store is on CPU; the embedding model can be on NPU (which is exactly the kind of single-shot, batch-friendly workload NPU is great at — see Chapter 3.3 for the OpenVINO 2026.1 `TextEmbeddingPipeline` NPU support). The reasoning model gets short, dense context; the agent stays responsive.

Where Intel and Microsoft Have Been Quiet

Honest gaps to flag, because this is the section most likely to invite extrapolation:

No Intel-published guidance on multi-step LLM agents on NPU. The Hugging Face × Intel Qwen3-8B Agent blog is the closest analog, and it explicitly runs on iGPU, not NPU.

Phi Silica is documented as single-turn. Microsoft routes it through Click to Do prompt templates with no learned router and no documented multi-step loop. The Windows Developer Blog extends the Phi Silica stack to DeepSeek-R1-Distill (1.5B at ~40 tok/s, 14B at ~8 tok/s on Snapdragon X NPU) — a reasoning model on NPU — but does not describe an agent architecture around it.

No published ReAct-loop measurements on Intel NPU exist. The 60–90 second budget in the table above is extrapolation from single-call benchmarks. If you build a real ReAct agent on NPU, the data points you collect will be original contributions to the public record.

The chapter's recommendation — prefer single-shot, fall back to plan-then-execute, treat ReAct as the iGPU pattern — reflects the absence of evidence for ReAct working well on NPU as much as it reflects the math. When more data appears the calculus might shift. As of May 2026 it hasn't.

What This Section Bought You

You should now understand:

- **Three reasoning architectures**: single-shot (NPU-native), plan-then-execute (decomposable), ReAct (iGPU pattern, not NPU)
- **A 5-step ReAct loop costs ~70–75 seconds on NPU** vs ~35 seconds on iGPU for an 8B INT4 model — extrapolated, not measured
- **ReAct magnifies the constraints NPUs are worst at**: repeated TTFT, growing context, greedy-only sampling, accumulating KV state
- **Single-shot tasks are more common than the cloud-agent literature suggests** — translation, summarization, tone-rewrite, code explanation all fit
- **Cascade triage is the NPU-native multi-step pattern** — tiny model decides whether the heavy model needs to run

- **Working memory (prompt) is bounded by `MAX_PROMPT_LEN`; long-term memory lives in local stores** with embedding-model retrieval between turns
- **Intel and Microsoft have published almost nothing on multi-step NPU agents** — be honest about the gap when designing for production

Chapter 2 ends here. The reader now has a working mental model of the constraints, the state, and the decision-making patterns. Chapter 3 turns to tools: how does an NPU-bound agent reach the world, what tool designs survive the latency budget, and where does the cloud fit?

Previous: *2.2 KV Cache Engineering* **Next:** *Chapter 3: Tool Use & Integration Patterns*