3.4 Structured Outputs and Constrained Decoding

An agent is only as reliable as the parser that reads its output. Chapter 3.1 covered designing the tools; Chapter 3.2 weighed local against cloud; Chapter 3.3 routed work across devices on the SoC. This section closes the loop on the agent-tool contract: how do you get the LLM to actually return something a tool can ingest, deterministically, every time, when the LLM in question is greedy-only and quantized to within an inch of its life?

The standard cloud answer is "set `response_format=json_object` and add `temperature=0.1`." That doesn't translate to NPU. On Intel NPU's `LLMPipeline`, sampling temperature is irrelevant — the only mode is greedy — and `response_format` isn't an OpenVINO concept. You get whatever the model emits, and if INT4 quantization has shifted the JSON-mode logit just enough for a stray "Here is the result:" prefix to win the argmax at position zero, your parser fails. This section is about preventing that.

The Three Tiers of Structured Output

There are three escalating tactics for getting the LLM to produce parseable output. Each costs more to implement and is more reliable than the last. Use the cheapest one that works for your use case.

Tier 1: Prompt engineering. Tell the model what format you want, give a few-shot example, and hope. Works surprisingly often at FP16 on a competent model. Degrades faster than you'd expect under quantization. Free; no infrastructure required.

Tier 2: Output filtering and retry. Parse the model's output; if it fails, retry with an error message included in the next prompt. Costs you extra inference round-trips when the model misbehaves. Reasonable for low-frequency failures (~5% error rate); ruinous for high-frequency failures because each retry burns ~5–10 seconds on NPU.

Tier 3: Constrained decoding. At each decode step, mask out tokens that would make the output invalid according to a formal grammar or schema. The model can only ever produce parseable output because the alternative tokens are zeroed in the logits before argmax. Implementation is real work — you need a grammar engine that hooks into the sampling loop — but the reliability is total. Once correctly wired up, parser failures simply cannot happen.

For agents that go to production, Tier 3 is usually the right answer. Tier 1 alone fails too often in practice; Tier 2 burns too much budget on retries when greedy decoding means the model's failure

mode is deterministic — if it failed once on a given prompt, it'll fail every time.

Why This Is Harder on NPU

Three things that make structured output specifically harder on NPU than on a cloud-GPU LLM API:

Greedy-only sampling. On GPU, you can sample with `temperature=0.7` and re-roll until you get valid JSON. The randomness gives you a free retry budget. On NPU, the same prompt always produces the same output. If it's broken, it's broken across every invocation.

Quantization shifts the logit landscape. Section 1.4 covered how INT4 quantization shifts argmax decisions on edge cases. Structured output lives at exactly the kind of edge: the difference between emitting `{"answer":` and `{ "answer":` (extra space) is two adjacent tokens with nearly identical FP16 logits. Quantization can flip which wins. Suddenly your parser, which assumed no leading space, breaks.

No native JSON-mode primitive. OpenAI's API has `response_format={"type": "json_object"}`, which routes to a specialized backend path. OpenVINO GenAI's `LLMPipeline` does not. You have to implement the constrained-decoding integration yourself, or use a third-party library.

The good news: the situation is improving. OpenVINO 2026.x has been adding structured-output APIs incrementally, and the most popular third-party constrained-decoding libraries (Outlines, lm-format-enforcer, Guidance) can be wired into OpenVINO with moderate effort. As of May 2026 the integration is real work, not a one-line config.

How Constrained Decoding Works

The mechanism is conceptually simple. At each decode step, the model produces a vector of logits over the full vocabulary (~32K–128K entries, depending on tokenizer). Ordinary greedy decoding takes the argmax over that vector. Constrained decoding adds a step before the argmax:

1. Track the current state in some grammar or schema (e.g., "we're inside a JSON object after a key, expecting a colon").
2. Determine the set of tokens that would be *valid* in that state (e.g., tokens that start with `:` or `: `).
3. Set the logits of all *invalid* tokens to negative infinity.
4. Take the argmax. The result is guaranteed to be valid.
5. Update the grammar state based on the chosen token.

For JSON, the grammar is fixed and well-known. The state machine tracks brace depth, whether the next token must be a key or a value, what types are still allowed, and so on. For JSON-schema-constrained output, the constraints are tighter — only specific keys are valid, only specific value types, only specific enum values — and the state machine encodes the schema as a tree.

The cost is mostly in step 2: efficiently computing the allowed-token mask. Naïve implementations check every vocabulary token at every step, which would be too slow. Production implementations precompute caches keyed on grammar state — given state S, here are the valid token IDs — and lookup is O(1). The libraries below all do this.

The Libraries

Outlines (`dottxt-ai/outlines`) is the most popular constrained-generation library in the Python ecosystem. Supports regex constraints, JSON schema, and arbitrary Lark grammars. Has an OpenVINO backend integration. Pattern:

```
from outlines import models, generate
import outlines


model = outlines.models.openvino("models/llama-3.1-8b-int4_npu")
generator = generate.json(model, schema=MyPydanticModel)
result = generator("List two countries in Europe.")
# result is guaranteed to be a valid MyPydanticModel instance
```

lm-format-enforcer (`noamgat/lm-format-enforcer`) is more lightweight, focused specifically on JSON and regex. Hooks into Hugging Face Transformers' `LogitsProcessor` interface, so anywhere that interface is supported (Optimum-Intel's `OVModelForCausalLM`, for instance), it plugs in. Often a better fit when you're already using `optimum-intel` directly rather than `openvino_genai.LLMPipeline`.

Guidance (`microsoft/guidance`) is more general, supporting multi-turn templates that interleave model-generated and developer-fixed content. Heavier than Outlines or lm-format-enforcer; more powerful if your use case actually needs the interleaved-template feature.

For most NPU agent use cases, **Outlines + JSON schema** is the right starting point. It's the most actively maintained, has direct OpenVINO support, and handles 95% of the realistic structured-output needs (tool calls, classification, extraction).

A Worked Pattern: Tool-Calling Schema

Take a concrete example. Your agent has three tools: `search_web`, `read_file`, `send_email`. You want the model to output a tool call like:

```
{
  "tool": "search_web",
  "arguments": {
    "query": "quarterly revenue Q3 2025"
```

```
  }
}
```

A Pydantic schema enforces this:

```
from pydantic import BaseModel, Field
from typing import Literal, Union

class SearchWeb(BaseModel):
    tool: Literal["search_web"]
    arguments: dict = Field(..., description="Must contain 'query'")

class ReadFile(BaseModel):
    tool: Literal["read_file"]
    arguments: dict = Field(..., description="Must contain 'path'")

class SendEmail(BaseModel):
    tool: Literal["send_email"]
    arguments: dict = Field(..., description="Must contain 'to', 'subject', 'body'")

ToolCall = Union[SearchWeb, ReadFile, SendEmail]
```

Wire it into Outlines:

```
from outlines import models, generate

model = models.openvino("models/llama-3.1-8b-int4_npu")
generator = generate.json(model, schema=ToolCall)

response = generator(
    "User asked: 'What were our Q3 numbers?'. Choose a tool to answer this."
)
# response is guaranteed to validate as ToolCall
# response.tool is one of the three Literal values
# response.arguments is a dict (the schema doesn't constrain its keys here,
#   but you can add tighter sub-schemas if needed)
```

The model can only emit tokens that lead to a valid `ToolCall`. It can't say "I don't know" or "Let me think about this" first — those tokens are masked. It must commit to a tool. Once committed, it must produce a valid arguments structure. Parser failures become impossible.

For tighter argument constraints, define explicit per-tool argument schemas:

```
class SearchWebArgs(BaseModel):
    query: str


class SearchWeb(BaseModel):
    tool: Literal["search_web"]
    arguments: SearchWebArgs
```

Now the model can only emit `{"tool": "search_web", "arguments": {"query": "..."}}` exactly. Adding extra keys is impossible.

The Costs You Pay

Constrained decoding isn't free. Three costs to budget for:

Decode latency increases 5–20%. Computing the allowed-token mask per step adds overhead. On NPU where decode is already bandwidth-bound, this overhead is real but not catastrophic — it adds CPU work between NPU forward passes, not bandwidth load. Outlines is particularly fast; lm-format-enforcer is comparable; Guidance can be slower.

The model's "free reasoning" is constrained too. If the model is forced into JSON from token zero, it can't first reason out loud and then commit to a tool. This is sometimes a real loss of capability — chain-of-thought reasoning is valuable for hard tool selections. The mitigation is a two-step pattern: first do an unconstrained reasoning step that picks a tool, then do a constrained step that emits the JSON. Two prefills, two decodes, but the reasoning is preserved.

Schemas that are too rigid produce nonsense outputs. If your schema says "the answer must be one of [A, B, C]" but the right answer is D, the model will pick whichever of A/B/C has the highest logit. The output validates; the answer is wrong. Constrained decoding turns "parser failures" into "schema-violation-of-reality failures" — same failure, different shape. Design schemas to include escape hatches: an optional `"unknown"` value, a `confidence` field that can be low.

Failure Modes

Specific things that go wrong when wiring constrained decoding into an NPU agent:

Tokenizer mismatch between Outlines and OpenVINO. Outlines builds its allowed-token cache against a specific tokenizer; if the OpenVINO export used a slightly different tokenizer config (e.g., `add_special_tokens=False`), the cached masks misalign and you get gibberish or compile errors. Verify tokenizer parity explicitly.

Schema is permissive in ways you didn't intend. A field typed as `dict` allows any keys, any values, infinite nesting. Constrained decoding will produce valid-but-useless JSON: `{"tool": "search_web", "arguments": {}}` is a valid `ToolCall` if `arguments` is just `dict`. Tighten with explicit sub-schemas.

The grammar engine doesn't know about a special token. EOS, BOS, padding tokens may need explicit handling in the grammar. Outlines handles this; some custom integrations don't. Symptom: the model "never stops" because the EOS token is masked out.

Mask computation becomes a hot path. For complex schemas (large enums, recursive structures), per-step mask computation can dominate CPU time. Profile. If it's bad, consider caching aggressively or simplifying the schema.

The compiled `LLMPipeline` doesn't expose a `LogitsProcessor` hook. Native `LLMPipeline.generate()` doesn't currently let you intercept logits between the model and the sampler. Two workarounds: (1) drop down to `OVModelForCausalLM` from Optimum-Intel, which uses standard Transformers `LogitsProcessor`, at the cost of losing some `LLMPipeline` NPU optimizations; (2) wait for OpenVINO to add the hook, which is on the public roadmap.

When Not to Use Constrained Decoding

Three cases where the cost outweighs the benefit:

The output is naturally well-formed. If your model already produces valid JSON 99% of the time at FP16 on a well-structured prompt, the 5–20% decode-latency cost of constrained decoding may not justify the small reliability gain. Run the unconstrained version first; measure.

You need the model's natural-language reasoning. Constrained decoding forbids the model from "thinking out loud." For complex reasoning steps, run unconstrained; only constrain the final answer.

Your schema would be enormous. Constrained decoding has overhead proportional to schema complexity. If your "structured output" is really "a free-form string field with some bounded annotations," constrained decoding may not help — the constrained portion is so small relative to the free portion that you're paying overhead for very little structure.

What This Section Bought You

You should now understand:

- **Three tiers of structured output**: prompt-only, output filtering with retry, constrained decoding — pick the cheapest that works
- **NPU's greedy-only sampling makes "failures are deterministic"** — if the model failed once, it fails every time, so retries don't help
- **Quantization shifts logit landscapes** in exactly the places structured output is fragile
- **Constrained decoding masks invalid tokens at every step**, guaranteeing schema-valid output
- **Outlines + JSON schema** is the standard recipe; lm-format-enforcer and Guidance are alternatives
- **Pydantic schemas** with `Literal` discriminators give you tool-calling reliability for free
- **Costs**: 5–20% decode latency overhead, reasoning constrained, schema-rigidity risk
- **Failure modes** include tokenizer mismatches, permissive sub-schemas, missing special-token handling
- **Skip it when**: the output is already well-formed, free-form reasoning is essential, the schema is mostly free-text

Chapter 4 turns from how the agent talks to its tools to how the agent is deployed and operated in production — serving, telemetry, rollouts, and the security model that on-device deployment actually implies.

Previous: *3.3 Multi-Device Orchestration on a Single SoC* **Next:** *Chapter 4: Production Deployment & Observability*

Revision #1
Created 2026-05-12 19:37:31 UTC by Admin
Updated 2026-05-12 19:37:31 UTC by Admin