Structured Output from Local LLMs: JSON, YAML, and Schemas

February 10, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** The most reliable way to get structured JSON from a local LLM is Ollama's format parameter with a JSON schema — it converts your schema to a grammar constraint and guarantees valid output. For llama.cpp users, the –json flag does the same thing. Both work by filtering tokens during generation so the model physically cannot produce invalid syntax. For Python pipelines, Instructor with Ollama gives you Pydantic-validated objects with automatic retries. Outlines gives you constrained generation with Hugging Face and vLLM backends. If you just need quick JSON, even prompting works with Qwen 2.5 14B+ — but always validate the output. For anything in a pipeline, use constraints. Prompting alone breaks on small models and under load.

📚 **Related:** llama.cpp vs Ollama vs vLLM · Best LLMs for Coding · Best LLMs for Data Analysis · Qwen Models Guide

You need your LLM to return `{"category": "urgent", "confidence": 0.92}` — not “Sure! Here’s the JSON you requested:” followed by a code block with a trailing comma and a missing bracket.

Structured output is what separates “chatting with an AI” from “building something with an AI.” Pipelines, agents, automation, data extraction — all of it breaks the moment your model returns text instead of parseable data. And LLMs return text. That’s literally what they do.

The good news: local tools have solved this. Ollama and llama.cpp can now guarantee valid JSON output at the token level — the model physically cannot produce invalid syntax. Here’s every method, ranked by reliability, with working code you can copy.

Why Structured Output Is Hard

LLMs generate tokens one at a time. Each token is chosen based on probability, not syntax rules. The model doesn’t “know” it’s in the middle of a JSON object — it’s predicting the next most likely token given everything before it.

This causes predictable failures:

- **Preamble text.** “Here is the JSON:” before the actual data. Your parser chokes on the English.
- **Missing brackets.** The model generates a long object and forgets to close it, or hits the token limit mid-output.
- **Wrong types.** `"True"` instead of `true`. `"null"` as a string instead of null. Numbers wrapped in quotes.
- **Trailing commas.** `{"a": 1, "b": 2,}` — valid in JavaScript, invalid in JSON.
- **Hallucinated keys.** You asked for `name` and `age`, the model adds `favorite_color` and `zodiac_sign`.
- **Markdown wrappers.** ` ```json ` code blocks around the output because the model learned JSON is usually displayed that way.

Bigger models are more reliable. Qwen 2.5 14B+ handles JSON prompting well. But “usually works” isn’t good enough for production. You need guarantees.

Methods Ranked by Reliability

1. Grammar-Constrained Generation (Best — 100% Reliable)

This is the correct answer for anything in a pipeline. Grammar constraints work at the token level — during generation, the model can only select tokens that keep the output syntactically valid. It cannot produce invalid JSON because invalid tokens have zero probability.

Ollama: format Parameter with JSON Schema

Since Ollama v0.5, you can pass a JSON schema directly to the `format` parameter. Ollama converts it to a GBNF grammar internally and passes it to llama.cpp.

```
curl http://localhost:11434/api/chat \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5:14b",
    "messages": [{"role": "user", "content": "Extract info: John Smith is 34, works at Acme Cor
    "stream": false,
    "format": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
```

```
        "age": {"type": "integer"},
        "company": {"type": "string"},
        "role": {"type": "string"}
      },
      "required": ["name", "age", "company", "role"]
    }
  }'
```

Response (guaranteed valid):

```
{"name": "John Smith", "age": 34, "company": "Acme Corp", "role": "software engineer"}
```

No preamble. No markdown. No trailing commas. The schema is enforced at the token level.

Python with the Ollama library:

```
import ollama
from pydantic import BaseModel

class Person(BaseModel):
    name: str
    age: int
    company: str
    role: str

response = ollama.chat(
    model="qwen2.5:14b",
    messages=[{"role": "user", "content": "Extract info: John Smith is 34, works at Acme Corp."
    format=Person.model_json_schema()
)

# Parse the guaranteed-valid JSON
person = Person.model_validate_json(response.message.content)
```

Tip: Include your schema description in the prompt too. The `format` parameter constrains syntax, but the model still needs to understand what you want in each field. The model doesn't see the schema — it only feels the constraints.

llama.cpp: –json Flag

llama.cpp supports JSON schema constraints directly:

```
# Pass a JSON schema file
./llama-cli -m model.gguf \
  --json schema.json \
  -p "Generate a person with name, age, and occupation"
```

Where `schema.json` contains:

```
{
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "age": {"type": "integer"},
    "occupation": {"type": "string"}
  },
  "required": ["name", "age", "occupation"]
}
```

For custom formats beyond JSON, llama.cpp uses GBNF (GGML BNF) grammars:

```
# Use a grammar file
./llama-cli -m model.gguf \
  --grammar-file grammars/json.gbnf \
  -p "Generate a person"
```

A GBNF grammar for constrained JSON looks like this:

```
root   ::= "{" ws "\"name\"" ws ":" ws string "," ws "\"age\"" ws ":" ws number "," ws "\"role\""
string ::= "\"" [^"\\]* "\""
number ::= [0-9]+
ws     ::= [ \t\n]*
```

This forces exactly three fields in exactly that order. The model has no choice but to comply.

LM Studio: Structured Output API

LM Studio supports structured output through the OpenAI-compatible API:

```
curl http://localhost:1234/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5-14b",
    "messages": [{"role": "user", "content": "Extract: Jane Doe, 28, engineer"}],
    "response_format": {
      "type": "json_schema",
      "json_schema": {
        "name": "person",
        "schema": {
          "type": "object",
          "properties": {
            "name": {"type": "string"},
            "age": {"type": "integer"},
            "role": {"type": "string"}
          },
          "required": ["name", "age", "role"]
        }
      }
    }
  }'
```

LM Studio uses Outlines internally for constraint enforcement.

Performance Cost

Grammar constraints slow generation by 30-80% depending on grammar complexity and GPU offloading. For a typical JSON schema, expect roughly half the normal token speed. Complex grammars with many optional fields hit harder.

This tradeoff is almost always worth it. A 50% speed reduction that guarantees valid output beats full-speed generation that fails 10% of the time and needs retry logic.

---

2. Python Libraries: Outlines and Instructor

For Python pipelines, two libraries dominate.

Outlines — Constrained Token Sampling

Outlines constrains generation at the token level, like grammar constraints but through Python. It supports Hugging Face Transformers, vLLM, llama.cpp, and MLX backends. It does not support Ollama directly.

```
from pydantic import BaseModel
from enum import Enum
import outlines

class Priority(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"

class Ticket(BaseModel):
    priority: Priority
    category: str
    summary: str
    action_items: list[str]

# Load model (Hugging Face backend)
model = outlines.models.transformers("Qwen/Qwen2.5-7B-Instruct")

# Generate with schema constraint
generator = outlines.generate.json(model, Ticket)
ticket = generator("Classify this support email: Our server has been down for 3 hours...")
# Returns a validated Ticket instance
```

Outlines also supports regex constraints, choice constraints (pick from a list), and custom grammars. Install with:

```
pip install outlines
# Plus your backend: transformers, llama-cpp-python, vllm, or mlx
```

Instructor — Retries and Validation

Instructor takes a different approach: it wraps LLM API calls with Pydantic validation and automatic retries. If the model returns invalid output, Instructor sends the validation error back to the model and asks it to fix it.

Works with Ollama through the OpenAI compatibility layer:

```
import instructor
from pydantic import BaseModel
from openai import OpenAI
```

```
class Entity(BaseModel):
    name: str
    type: str
    confidence: float

class Extraction(BaseModel):
    entities: list[Entity]

client = instructor.from_openai(
    OpenAI(base_url="http://localhost:11434/v1", api_key="ollama"),
    mode=instructor.Mode.JSON
)

result = client.chat.completions.create(
    model="qwen2.5:14b",
    messages=[{"role": "user", "content": "Extract entities: Apple CEO Tim Cook announced the ne
    response_model=Extraction,
    max_retries=3
)

for entity in result.entities:
    print(f"{entity.name} ({entity.type}): {entity.confidence}")
```

Install with:

```
pip install instructor openai
```

When to Use Which

	Outlines	**Instructor**
Method	Constrains tokens during generation	Validates after generation, retries on failure
Guarantee	100% valid output (token-level)	High reliability with retries
Backends	Transformers, vLLM, llama.cpp, MLX	Any OpenAI-compatible API (Ollama, LM Studio, cloud)
Ollama support	No	Yes
Speed	Single pass, constrained	May need multiple passes on retry
Best for	Local inference pipelines	Multi-provider setups, Ollama users

3. JSON Mode (Good, Not Perfect)

Ollama's basic `format: "json"` forces the model to output valid JSON, but without a schema constraint. The output will parse, but the structure isn't guaranteed.

```
curl http://localhost:11434/api/generate \
  -d '{
    "model": "qwen2.5:14b",
    "prompt": "List 3 fruits with their colors. Return JSON.",
    "format": "json",
    "stream": false
  }'
```

You'll get valid JSON, but the keys and structure depend on what the model decides. It might return `{"fruits": [...]}` or `[{"name": "apple", "color": "red"}, ...]` or something else entirely.

Use this when you need valid JSON but the exact structure is flexible. For predictable structure, pass a full schema to `format` instead.

4. Prompting (Least Reliable)

Adding "Return valid JSON only. No other text." to your prompt works surprisingly well with capable models. It fails surprisingly often with smaller ones.

```
Extract entities from this text. Return a JSON array of objects with "name" and "type" fields. Re

Text: "Tim Cook announced the new iPhone at Apple Park in Cupertino."
```

This works most of the time with Qwen 2.5 14B+. It fails regularly with 7B models, especially on complex schemas. Common failures:

- Model adds "Here's the extracted data:" before the JSON
- Model wraps output in markdown code blocks
- Model adds an explanation after the JSON
- Model invents extra fields
- Small models produce invalid syntax under complex prompts

If you use prompting, always validate the output:

```
import json

def parse_llm_json(text: str):
    # Try direct parse
    try:
        return json.loads(text)
    except json.JSONDecodeError:
        pass

    # Try extracting from markdown code block
    if "```" in text:
        code = text.split("```")[1]
        if code.startswith("json"):
            code = code[4:]
        try:
            return json.loads(code.strip())
        except json.JSONDecodeError:
            pass

    # Try finding JSON in the text
    for start in range(len(text)):
        if text[start] in "{[":
            for end in range(len(text), start, -1):
                try:
                    return json.loads(text[start:end])
                except json.JSONDecodeError:
                    continue

    raise ValueError(f"No valid JSON found in: {text[:200]}")
```

Or use the `json_repair` library:

```
pip install json_repair
```

```
import json_repair
result = json_repair.loads(broken_json_string)
```

Which Models Handle Structured Output Best

Not all models are equal at generating JSON, even without constraints. If you're using prompting or basic JSON mode (no schema), the model matters.

Model	JSON Reliability (Prompting)	Notes
Qwen 2.5 (14B+)	Excellent	Specifically trained for structured output. Best native JSON among open models.
Qwen 2.5 (7B)	Good	Reliable for simple schemas, occasional failures on nested structures.
Llama 3.3 (70B)	Good	Strong tool/function calling support. Reliable with clear prompting.
Llama 3.1 (8B)	Moderate	Works for simple JSON. Struggles with complex nested schemas.
Mistral Nemo (12B)	Good	Solid JSON output, good instruction following.
DeepSeek R1	Good	Strong reasoning helps with complex schemas.
Phi-3 (3.8B)	Moderate	Decent for its size, but use constraints for reliability.
Any model <7B	Poor	Always use grammar constraints. Prompting alone isn't reliable.

Qwen 2.5 was specifically trained with improved structured output generation — it's the default recommendation for JSON-heavy workflows. If you're already running Qwen 2.5 for other tasks, you're in good shape.

With grammar constraints (Ollama `format` with schema, or llama.cpp `--json`), model quality matters less. The constraints guarantee valid output regardless. But a better model produces more accurate content within the valid structure.

Common Patterns

Entity Extraction

Pull structured data from unstructured text:

```
import ollama
from pydantic import BaseModel

class Entity(BaseModel):
    name: str
    type: str  # person, organization, location, date

class Entities(BaseModel):
    entities: list[Entity]

response = ollama.chat(
    model="qwen2.5:14b",
    messages=[{
        "role": "user",
        "content": """Extract all entities from this text:
        "On March 15, Google CEO Sundar Pichai spoke at Stanford University about Gemini."

        Entity types: person, organization, location, date"""
    }],
    format=Entities.model_json_schema()
)
```

Classification

Route content into categories:

```
class Classification(BaseModel):
    category: str  # bug, feature, question, docs
    priority: str  # low, medium, high, critical
    summary: str

response = ollama.chat(
    model="qwen2.5:14b",
    messages=[{
        "role": "user",
        "content": """Classify this GitHub issue:
        "App crashes when clicking save with an empty title field. Stack trace attached.
        This is blocking our release."

        Categories: bug, feature, question, docs
        Priorities: low, medium, high, critical"""
    }],
    format=Classification.model_json_schema()
)
```

Data Transformation

Convert messy input into clean schema-conforming output:

```
class Product(BaseModel):
    name: str
    price_usd: float
    in_stock: bool
    category: str

class ProductList(BaseModel):
    products: list[Product]

messy_input = """
we got:
- nvidia 3090 used, about $750, have 3 in stock
- 4070 ti super NEW $600 (out of stock rn)
- amd 7900 xtx ~$850 yes available
"""

response = ollama.chat(
    model="qwen2.5:14b",
    messages=[{
        "role": "user",
        "content": f"Parse this inventory into structured data:\n{messy_input}"
    }],
    format=ProductList.model_json_schema()
)
```

Multi-Step Pipeline

Chain structured outputs for agent-like workflows:

```
# Step 1: Analyze the query
class QueryAnalysis(BaseModel):
    intent: str
    entities: list[str]
    needs_search: bool
```

```
# Step 2: Generate search query (if needed)
class SearchQuery(BaseModel):
    query: str
    filters: dict

# Step 3: Format the response
class Response(BaseModel):
    answer: str
    sources: list[str]
    confidence: float

# Each step uses format= to guarantee parseable output
# Chain them together in your pipeline logic
```

YAML: Use JSON Instead

YAML constrained generation doesn't exist in Ollama or llama.cpp as of early 2026. You could write a custom GBNF grammar for YAML, but YAML's indentation-sensitive syntax makes this painful and error-prone.

The practical solution: generate JSON with constraints, then convert:

```
import json
import yaml

# Get guaranteed-valid JSON from Ollama
json_data = json.loads(response.message.content)

# Convert to YAML if that's what your pipeline needs
yaml_output = yaml.dump(json_data, default_flow_style=False)
```

Models are also better at JSON than YAML because there's more JSON in training data and JSON has simpler syntax rules (brackets vs. indentation). Stick with JSON for LLM output.

Practical Tips

Set temperature to 0 for structured output. Higher temperatures increase randomness, which means more creative interpretations of your schema. For data extraction and classification, you want deterministic output.

```
curl http://localhost:11434/api/chat \
  -d '{
    "model": "qwen2.5:14b",
    "messages": [{"role": "user", "content": "..."}],
    "format": {"type": "object", "properties": {...}},
    "options": {"temperature": 0}
  }'
```

Include an example in your prompt for complex schemas. The model doesn't see the JSON schema from the `format` parameter — it only constrains the output tokens. Put an example of what you want in the prompt:

```
Extract product info. Example output:
{"name": "RTX 3090", "price": 750, "condition": "used", "in_stock": true}

Now extract from: "We have a brand new 4090 for $1599"
```

Use Pydantic's `.model_json_schema()` everywhere. Define your schema once as a Pydantic model, then use it for Ollama's `format`, for Instructor's `response_model`, and for your own validation. One source of truth.

Always validate, even with constraints. Grammar constraints guarantee valid syntax but not valid content. The JSON will parse, but the values might be hallucinated. Validate that extracted data makes sense for your use case.

For small models, always use constraints. Models under 7B are unreliable at producing valid JSON through prompting alone. With Ollama's `format` parameter, even a 3B model produces perfectly valid JSON every time.

The Bottom Line

Use Ollama's `format` parameter with a JSON schema for most structured output needs. It's the easiest method, works with any model Ollama supports, and guarantees valid output. For Python pipelines that need Pydantic integration, add Instructor for Ollama or Outlines for Hugging Face/vLLM backends.

Don't rely on prompting alone unless you're using a strong model (Qwen 2.5 14B+) and have validation and retry logic. For anything in an automated pipeline, grammar constraints eliminate an entire category of bugs.

The models keep getting better at native JSON — Qwen 2.5 specifically targeted this — but constraints are cheap insurance. A 30-50% speed hit is nothing compared to a pipeline failure at 3 AM because the model decided to add a friendly explanation before the JSON.

Related Guides

- llama.cpp vs Ollama vs vLLM
- Best Local LLMs for Coding
- Best Local LLMs for Data Analysis
- Qwen Models Guide
- Ollama Troubleshooting
- Run Your First Local LLM
- Local AI Privacy Guide

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/structured-output-local-llms/

Free guides for running AI locally