Engineering Deterministic LLM Systems

Engineering Deterministic LLM Systems

LLMs are probabilistic components. A reliable application should not depend on an LLM producing byte-for-byte identical text on every run.

The practical goal is to make the **system behavior reproducible, constrained, testable, and safe**, even when the model has small variations.

> The LLM handles ambiguity. Deterministic software enforces the contract.

Contents

1. What Determinism Means

There are several different targets. They should not be confused.

1.1 Token-level determinism

The same input produces exactly the same token sequence.

This is the strongest target and is difficult to guarantee across:

- model revisions;
- providers;
- hardware and GPU kernels;
- batching strategies;

• library versions;
• distributed inference configurations.

It is useful for debugging, but it is rarely the best production contract.

1.2 Structural determinism

Every accepted response follows the same data contract:

• the same required fields;
• the same data types;
• controlled enum values;
• no unexpected properties;
• valid references between objects.

The values may vary, but the structure is predictable.

1.3 Semantic determinism

Different textual outputs produce the same business meaning.

For example, these answers are textually different but functionally equivalent:

```
"The invoice should be rejected because the total is negative."
"Reject: invoice total must be greater than or equal to zero."
```

Semantic determinism requires domain-specific evaluators, not string equality.

1.4 Operational determinism

The same validated decision causes the same controlled action:

• the same tool is authorized;
• the same arguments are validated;
• the same idempotency key is used;
• the same deterministic code path executes;
• retries do not duplicate side effects.

For production systems, structural, semantic, and operational determinism are usually more important than identical prose.

2. Why Determinism Matters

Deterministic boundaries improve:

• **Reliability:** malformed or unsafe output cannot silently propagate.
• **Debugging:** a failed request can be replayed with the same artifacts.
• **Testing:** behavior can be compared against versioned expectations.

• **Compliance:** decisions, inputs, and validation results are auditable.
• **Security:** authorization remains outside the model.
• **User experience:** equivalent requests behave consistently.
• **Cost control:** bounded loops prevent uncontrolled retries.
• **Safe automation:** side effects are validated and idempotent.

The stricter the downstream action, the stricter the boundary should be. A blog draft can tolerate variation. A payment, permission change, or production code deployment cannot.

3. Sources of Nondeterminism

3.1 Sampling

Temperature, nucleus sampling, top-k sampling, and random seeds influence token selection. Sampling is the most visible source, but not the only one.

3.2 Numerical execution

Parallel floating-point operations can be evaluated in different orders. Tiny logit differences can change the highest-ranked token when two candidates are close. Once one token changes, autoregressive generation may follow a different path.

3.3 Model and provider changes

Unversioned model aliases may point to newer snapshots. Providers can also change serving infrastructure, kernels, safety layers, or default parameters.

3.4 Prompt and context changes

Whitespace, message ordering, system instructions, tool descriptions, examples, and context truncation can change model behavior.

3.5 Retrieval

A RAG system can vary because of:

• changing documents;
• asynchronous indexing;
• approximate nearest-neighbor search;
• ties in similarity scores;
• metadata changes;

• reranker updates;

• nondeterministic ordering of equal-scored chunks.

3.6 Tools and external state

APIs, databases, clocks, user permissions, network responses, and concurrent updates are inherently dynamic.

3.7 Workflow concurrency

Parallel tools may finish in different orders. If result order affects the next prompt, the model receives different context.

4. Generation Parameters

Parameter	Effect	Reliability guidance
`temperature`	Scales the token distribution	Use `0` or a low value for extraction, routing, and planning
`top_p`	Keeps the smallest token set covering probability mass `p`	Usually leave at `1` when using low temperature
`top_k`	Keeps only the top `k` tokens	Disable or keep fixed for reproducible self-hosted inference
`min_p`	Removes tokens far below the most likely token	Useful for sampling, usually unnecessary for deterministic tasks
`do_sample`	Enables stochastic sampling in many local libraries	Set to `false` for greedy decoding
`num_beams`	Searches multiple candidate sequences	Keep at `1` unless beam search is explicitly required
`max_new_tokens`	Limits output length	Always set a task-appropriate bound
`stop`	Stops on specified sequences	Version and test stop sequences; avoid accidental JSON truncation
`seed`	Initializes supported random generators	Useful for best-effort replay, not a universal guarantee
repetition penalties	Discourage repeated text	Avoid unless the task needs them; they change token probabilities

For a managed API, a conservative structured-task configuration is:

```
{
  "temperature": 0,
  "top_p": 1,
  "max_output_tokens": 800,
  "seed": 7
}
```

For a self-hosted Transformers-style runtime:

```
generation_config = {
    "do_sample": False,
    "num_beams": 1,
    "max_new_tokens": 800,
}
```

Some libraries reject `temperature=0` when sampling is enabled. Greedy decoding with `do_sample=False` is the clearer local configuration.

Do not tune `temperature` and `top_p` at the same time unless an evaluation shows that the combination improves the task. Otherwise, it becomes difficult to know which parameter caused a behavior change.

5. Structured Output and Constrained Decoding

Prompting a model to "return JSON" is not the same as enforcing a schema.

5.1 Prompt-only JSON

The model is asked to produce JSON but can still generate:

- markdown fences;
- missing fields;
- invalid enums;
- extra explanatory text;
- malformed JSON;
- incorrect types.

This is acceptable only for low-risk prototypes.

5.2 JSON mode

JSON mode generally constrains output to syntactically valid JSON. It does not necessarily guarantee that the JSON follows the application's field-level schema.

5.3 Strict structured output

Strict structured output uses a JSON Schema, grammar, regex, or typed tool definition to constrain generation.

A typical constrained decoder works as follows:

1. Compile the schema into a grammar or parser state machine.
2. Track the parser state for the generated prefix.
3. Calculate which tokenizer tokens can legally continue that prefix.
4. Mask every invalid token by setting its logit to negative infinity.
5. Renormalize probabilities across the remaining valid tokens.
6. Select or sample the next valid token.
7. Update the parser state and repeat.

Conceptually:

```
logits = model(prefix)
allowed_tokens = grammar.allowed_tokens(prefix)

for token_id in vocabulary:
```

```
        if token_id not in allowed_tokens:
            logits[token_id] = float("-inf")

    next_token = argmax(logits)
```

Invalid tokens are usually made impossible, not merely less probable.

Constrained decoding guarantees syntax and supported schema rules. It does **not** guarantee semantic correctness. A field may contain a valid enum value that is wrong for the user's situation.

6. JSON Schema Example

This schema extracts a support ticket into a controlled structure:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "SupportTicket",
  "type": "object",
  "additionalProperties": false,
  "required": ["category", "priority", "summary", "requires_human"],
  "properties": {
    "category": {
      "type": "string",
      "enum": ["billing", "technical", "account", "other"]
    },
    "priority": {
      "type": "string",
      "enum": ["low", "medium", "high", "critical"]
    },
    "summary": {
      "type": "string",
      "minLength": 1,
      "maxLength": 500
    },
    "requires_human": {
      "type": "boolean"
    },
    "customer_id": {
      "type": ["string", "null"]
    }
  }
}
```

Important design choices:

• Use `additionalProperties: false` to block invented fields.
• Use enums for controlled decisions.
• Require every field the application depends on.
• Allow `null` explicitly when data may be absent.
• Add length and numeric bounds where supported.
• Version the schema independently from the prompt.
• Keep descriptions precise because some providers also expose them to the model.

Exact API fields vary by provider, but the conceptual request is:

```
{
  "model": "pinned-model-snapshot",
  "input": "The customer cannot access their account.",
  "generation": {
    "temperature": 0,
```

```
    "top_p": 1,
    "max_output_tokens": 500
  },
  "response_format": {
    "type": "json_schema",
    "name": "support_ticket",
    "strict": true,
    "schema": {
      "type": "object",
      "additionalProperties": false,
      "required": ["category", "priority", "summary", "requires_human"],
      "properties": {
        "category": {
          "type": "string",
          "enum": ["billing", "technical", "account", "other"]
        },
        "priority": {
          "type": "string",
          "enum": ["low", "medium", "high", "critical"]
        },
        "summary": {"type": "string"},
        "requires_human": {"type": "boolean"}
      }
    }
  }
}
```

7. Validation, Normalization, and Bounded Repair

Structured decoding is the first boundary. Application validation is the second.

7.1 Typed validation with Pydantic

```
from typing import Literal

from pydantic import BaseModel, ConfigDict, Field, model_validator


class SupportTicket(BaseModel):
    model_config = ConfigDict(extra="forbid")

    category: Literal["billing", "technical", "account", "other"]
    priority: Literal["low", "medium", "high", "critical"]
    summary: str = Field(min_length=1, max_length=500)
    requires_human: bool
    customer_id: str | None = None

    @model_validator(mode="after")
    def validate_critical_ticket(self):
        if self.priority == "critical" and not self.requires_human:
            raise ValueError("Critical tickets must require human review")
        return self
```

JSON Schema validates structure. The model validator enforces a semantic business rule that may be difficult or undesirable to encode in the generation grammar.

7.2 Canonical JSON

Canonicalization prevents harmless key-order differences from appearing as different outputs:

```
import hashlib
import json
from typing import Any
```

```
def canonical_json(value: Any) -> str:
    return json.dumps(
        value,
        sort_keys=True,
        separators=(",", ":"),
        ensure_ascii=False,
    )


def content_hash(value: Any) -> str:
    payload = canonical_json(value).encode("utf-8")
    return hashlib.sha256(payload).hexdigest()
```

Normalize only differences that are irrelevant to the business contract. Do not sort lists when order has meaning.

7.3 Bounded repair

Never retry indefinitely. Use a small, explicit repair budget:

```
import json
from collections.abc import Callable

from pydantic import ValidationError


Generate = Callable[[str, dict, list[str]], str]


def extract_ticket(
    text: str,
    schema: dict,
    generate: Generate,
    max_attempts: int = 2,
) -> SupportTicket:
    errors: list[str] = []

    for _ in range(max_attempts):
        raw = generate(text, schema, errors)

        try:
            return SupportTicket.model_validate(json.loads(raw))
        except (json.JSONDecodeError, ValidationError) as exc:
            errors = [str(exc)]

    raise RuntimeError("Unable to produce a valid support ticket")
```

For high-risk operations, route the request to a human or fail closed after the repair budget is exhausted.

8. Evaluators

An evaluator determines whether an output is acceptable. Use deterministic evaluators wherever possible and LLM judges only where semantic judgment is necessary.

8.1 Evaluation order

Run inexpensive, objective checks first:

1. Request completed successfully.
2. Output parses.
3. JSON Schema passes.
4. Semantic business rules pass.
5. References and permissions are valid.
6. Task-specific correctness passes.
7. Safety checks pass.
8. Optional human-calibrated LLM judge runs.

8.2 Standard evaluator response

Make evaluators return structured results:

```
{
  "eval_version": "ticket-eval-3",
  "case_id": "ticket-0042",
  "passed": false,
  "score": 0.75,
  "checks": [
    {
      "name": "schema_valid",
      "passed": true,
      "score": 1.0,
      "details": null
    },
    {
      "name": "critical_requires_human",
      "passed": false,
      "score": 0.0,
      "details": "Critical ticket was not routed for human review"
    }
  ],
  "artifacts": {
    "prompt_hash": "...",
    "schema_hash": "...",
    "context_hash": "...",
    "output_hash": "..."
  }
}
```

8.3 Replay stability evaluator

Run each case multiple times and compare normalized results:

```
from collections import Counter
from collections.abc import Callable
from typing import Any


def replay_stability(
    invoke: Callable[[], Any],
    runs: int = 10,
) -> dict:
    outputs = [invoke() for _ in range(runs)]
    canonical = [canonical_json(output) for output in outputs]
    counts = Counter(canonical)
    most_common_count = counts.most_common(1)[0][1]

    return {
        "runs": runs,
        "unique_output_count": len(counts),
        "exact_replay_rate": most_common_count / runs,
        "output_hashes": [
            hashlib.sha256(value.encode("utf-8")).hexdigest()
```

```
                for value in canonical
            ],
        }
```

Useful stability metrics include:

- schema pass rate;
- unique normalized output count;
- exact replay rate;
- semantic agreement rate;
- tool selection agreement;
- action argument agreement;
- refusal agreement;
- average repair attempts.

8.4 Exact and semantic evaluators

Use exact comparison for:

- IDs;
- enums;
- amounts;
- dates after normalization;
- tool names;
- permission decisions;
- deterministic code artifacts.

Use semantic comparison for:

- summaries;
- explanations;
- open-ended classifications with equivalent labels;
- answers where multiple phrasings are correct.

An LLM judge should have:

- a strict rubric;
- examples of passing and failing outputs;
- structured output;
- a pinned judge model;
- periodic calibration against human labels;
- no authority over security or permission decisions.

9. Deterministic RAG

RAG determinism depends on the full retrieval pipeline, not only the generator.

9.1 Version every retrieval artifact

Record:

• document version;
• parser version;
• chunking version;
• embedding model snapshot;
• vector index version;
• metadata schema version;
• reranker version;
• query transformation version.

9.2 Stable chunk identity

Use content-derived or versioned chunk IDs:

```
chunk_id = SHA256(document_id + document_version + section_path + chunk_text)
```

Avoid IDs based only on insertion order.

9.3 Stable retrieval ordering

Approximate search can return equal or nearly equal scores in different orders.
Add an explicit tie-breaker:

```
results = sorted(
    results,
    key=lambda item: (-item.score, item.document_id, item.chunk_id),
)
```

9.4 Canonical context assembly

Build prompts from a stable representation:

```
[document_id, version, section, chunk_id, content]
```

Apply a defined sort order and a fixed token-budget policy. Log which chunks were included and which were dropped.

9.5 Retrieval evaluators

Given true positives (TP), false positives (FP), and false negatives (FN):

```
precision = TP / (TP + FP)
recall    = TP / (TP + FN)
F1        = 2 * precision * recall / (precision + recall)
```

Evaluate:

• recall at k;
• precision at k;
• mean reciprocal rank;
• normalized discounted cumulative gain;
• evidence coverage;
• answer groundedness;
• no-answer accuracy;
• freshness and version correctness.

The first-stage retriever usually prioritizes recall. The reranker and final context selection should improve precision.

Do not generate every evaluation question from the same final chunk used as the answer. Include real user queries, paraphrases, multi-document questions, conflicting information, and unanswerable questions.

10. Deterministic Agents and Tool Use

An agent introduces several decisions:

1. Whether a tool is needed.
2. Which tool should be selected.
3. Which arguments should be supplied.
4. Whether the caller is authorized.
5. How the result should be interpreted.
6. Whether another step is required.

Evaluate these decisions independently.

10.1 Typed tool contracts

```
{
  "name": "create_support_ticket",
  "description": "Create one support ticket after the user confirms submission.",
  "input_schema": {
    "type": "object",
    "additionalProperties": false,
    "required": ["summary", "priority", "idempotency_key"],
    "properties": {
      "summary": {"type": "string", "minLength": 1},
      "priority": {
        "type": "string",
        "enum": ["low", "medium", "high", "critical"]
      },
      "idempotency_key": {"type": "string", "minLength": 16}
    }
  }
}
```

Tool descriptions should define clear responsibility boundaries. Two tools with

overlapping descriptions create routing ambiguity.

10.2 Keep authorization outside the model

The model may propose a tool call. Application code must:

- authenticate the user;
- authorize the tenant, user, resource, and tool;
- validate arguments;
- apply rate and cost limits;
- require confirmation where appropriate;
- execute with an idempotency key;
- log the result without exposing secrets.

10.3 Deterministic workflow around agentic decisions

```
User request
    -> deterministic policy check
    -> model proposes typed action
    -> schema validation
    -> authorization
    -> optional human confirmation
    -> idempotent tool execution
    -> deterministic result normalization
    -> model summarizes result
```

If the path is already known, use a workflow rather than asking an agent to rediscover it.

10.4 Agent evaluators

Measure:

- tool selection accuracy;
- argument validity;
- authorization correctness;
- execution success;
- task completion rate;
- unnecessary tool-call rate;
- duplicate side-effect rate;
- loop and timeout rate;
- cost per successful task;
- human escalation accuracy.

11. Deterministic Code Generation

Do not rely on an LLM to produce the final production source text when exact,

repeatable output is required.

Use a two-stage architecture:

```
Requirements + approved components
    -> LLM produces typed intermediate representation (IR)
    -> deterministic validator
    -> deterministic AST/template compiler
    -> formatter, compiler, linter, and tests
```

11.1 Typed intermediate representation

```
{
  "schema_version": "1.0",
  "page_id": "employee-create",
  "permissions": ["employee:create"],
  "state": {
    "name": "",
    "department_id": null,
    "submitting": false
  },
  "components": [
    {
      "id": "name-input",
      "type": "TextInput",
      "props": {
        "label": "Name",
        "required": true
      },
      "binding": "state.name"
    },
    {
      "id": "save-button",
      "type": "Button",
      "props": {
        "label": "Save",
        "variant": "primary"
      },
      "event": "submitEmployee"
    }
  ]
}
```

The schema should constrain component names to the currently approved catalog.

Component-specific property schemas can be represented as a discriminated union:

```
{
  "oneOf": [
    {
      "properties": {
        "type": {"const": "Button"},
        "props": {"$ref": "#/$defs/ButtonProps"}
      }
    },
    {
      "properties": {
        "type": {"const": "TextInput"},
        "props": {"$ref": "#/$defs/TextInputProps"}
      }
    }
  ]
}
```

11.2 Deterministic compiler

The renderer should be a pure function:

```
files = render(IR, component_catalog_version, renderer_version)
```

Requirements:

• generate an AST instead of concatenating arbitrary strings;
• pin component, renderer, formatter, and compiler versions;
• use stable import and node ordering;
• use deterministic variable and ID allocation;
• exclude timestamps and random UUIDs;
• format with a pinned formatter;
• compile, lint, and test every output;
• cache by normalized input and version hashes.

For the same validated IR and versions, the generated code should be identical.

11.3 Code evaluators

Measure:

• IR schema pass rate;
• component and property validity;
• compile rate;
• type-check rate;
• lint rate;
• unit and integration test pass rate;
• approved dependency usage;
• accessibility rules;
• visual regression results;
• human acceptance and edit rate.

Repair the IR rather than manually patching generated source whenever possible.

12. Self-Hosted Inference

Self-hosting gives more control, but reproducibility still requires environment management.

Pin:

• model commit and tokenizer files;
• inference-server version;
• CUDA, driver, and kernel versions;
• GPU model;
• quantization configuration;
• attention backend;

• container image digest;
• generation configuration.

Seed all relevant random generators:

```
import os
import random

import numpy as np
import torch


SEED = 7

os.environ["PYTHONHASHSEED"] = str(SEED)
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

torch.backends.cudnn.benchmark = False
torch.use_deterministic_algorithms(True)
```

Deterministic operations may be slower. PyTorch also notes that complete reproducibility is not guaranteed across releases, platforms, or CPU/GPU execution. Treat environment pinning as risk reduction, not a universal proof.

13. Observability and Replay

Store enough metadata to reproduce a request without storing unnecessary secrets.

Recommended trace envelope:

```
{
  "request_id": "req-123",
  "timestamp": "2026-08-04T10:00:00Z",
  "application_version": "support-api-17",
  "model": "provider/model-snapshot",
  "tokenizer_version": "...",
  "prompt_version": "ticket-prompt-8",
  "prompt_hash": "...",
  "schema_version": "ticket-schema-3",
  "schema_hash": "...",
  "generation_config": {
    "temperature": 0,
    "top_p": 1,
    "max_output_tokens": 500,
    "seed": 7
  },
  "context": {
    "index_version": "kb-2026-08-04",
    "chunk_ids": ["chunk-a", "chunk-b"],
    "context_hash": "..."
  },
  "tools": {
    "registry_version": "tools-12",
    "calls": []
  },
  "validation": {
    "schema_passed": true,
    "semantic_passed": true,
    "repair_attempts": 0
  },
  "output_hash": "..."
}
```

Do not log passwords, access tokens, private keys, raw secrets, or personal data that is unnecessary for debugging. Store references or redacted values instead.

14. Testing and Release Gates

14.1 Golden dataset

Create a versioned dataset containing:

• normal cases;
• boundary cases;
• ambiguous inputs;
• invalid inputs;
• adversarial inputs;
• no-answer cases;
• permission failures;
• tool failures;
• production incidents;
• cases requiring human review.

Split by user, project, document, or repository when random row-level splitting would leak similar examples into the test set.

14.2 CI evaluation

Every change to the following should trigger evaluation:

• model snapshot;
• prompt;
• schema;
• retrieval configuration;
• component or tool catalog;
• validator;
• workflow;
• inference runtime.

Example release gates:

```
{
  "schema_pass_rate_min": 1.0,
  "semantic_rule_pass_rate_min": 0.99,
  "tool_selection_accuracy_min": 0.95,
  "unauthorized_action_rate_max": 0.0,
  "duplicate_side_effect_rate_max": 0.0,
  "p95_latency_ms_max": 3000,
  "cost_per_success_max_usd": 0.05
}
```

Thresholds must reflect the risk of the specific application. A medical, financial, legal, or permission-changing workflow needs stronger controls and human review than a drafting assistant.

14.3 Production monitoring

Monitor:

• validation failure rate;
• repair and retry rate;
• output distribution drift;
• refusal rate;
• no-answer accuracy;
• tool confusion pairs;
• permission denials;
• loop and timeout rate;
• latency and cost;
• human corrections;
• model and retrieval version changes.

Turn confirmed production failures into regression cases.

15. Anti-patterns

"Temperature zero guarantees identical output"

It reduces sampling variation but does not freeze models, context, infrastructure, retrieval, or external state.

"The prompt says return JSON"

Prompting is not a schema contract. Use strict structured output and application validation.

Parsing critical free text with regex

Use typed output. Regex parsing is fragile when wording changes.

Retrying until something passes

Unbounded retries hide failures, increase cost, and can duplicate side effects.

Letting the model authorize itself

Authentication and authorization must be enforced in code.

Treating an LLM judge as ground truth

Judges are probabilistic and can share biases with the model under test. Calibrate them against human labels and use deterministic checks first.

Comparing only exact strings

Exact matching incorrectly fails semantically equivalent prose. Match the evaluator to the contract.

Using unversioned model aliases

A silent model upgrade can change behavior without an application deployment.

Ignoring retrieval versions

The same generator with different evidence is a different system.

Allowing arbitrary model-generated code to execute

Prefer typed plans, deterministic compilers, sandboxing, tests, and explicit approval.

16. Production Checklist

Model and generation

- [ ] Model snapshot is pinned.
- [ ] Tokenizer and inference runtime are versioned.
- [ ] Sampling is disabled or explicitly configured.
- [ ] Output token limit is set.
- [ ] Seed is recorded when supported.

Prompt and context

- [ ] System prompt and examples are versioned.
- [ ] Prompt hash is recorded.
- [ ] Context ordering is stable.
- [ ] Retrieval and index versions are recorded.
- [ ] Token-budget truncation is deterministic.

Output contract

- [ ] A strict JSON Schema or typed tool contract is used.
- [ ] Unexpected fields are rejected.
- [ ] Semantic validators run after parsing.
- [ ] Repair attempts are bounded.
- [ ] High-risk failures route to a human or fail closed.

Tools and actions

- [ ] Tool responsibilities do not overlap unnecessarily.
- [ ] Authentication and authorization are enforced in code.
- [ ] Arguments are validated before execution.
- [ ] Side effects use idempotency keys.
- [ ] Confirmation is required where appropriate.

Evaluation

- [ ] A versioned golden dataset exists.
- [ ] Schema, semantic, safety, and task evaluators are separate.
- [ ] Replay stability is measured across multiple runs.
- [ ] RAG retrieval is evaluated separately from generation.
- [ ] Tool selection is evaluated separately from execution.
- [ ] LLM judges are calibrated against human labels.
- [ ] Production failures become regression cases.

Observability

- [ ] Model, prompt, schema, context, and tool versions are logged.
- [ ] Inputs and outputs have canonical hashes.
- [ ] Secrets and unnecessary personal data are redacted.
- [ ] Requests can be replayed from stored artifacts.
- [ ] Drift, retries, failures, latency, and cost are monitored.

Recommended Architecture

```
User input
    -> normalization
    -> versioned retrieval/context assembly
    -> pinned model with conservative decoding
    -> schema-constrained output
    -> typed parsing
    -> semantic validation
    -> policy and authorization checks
    -> deterministic workflow/tool/compiler
    -> post-condition checks
    -> canonical response + audit trace
```

The model is one component in this architecture. Reliability comes from the contracts around it.

References

- PyTorch reproducibility
- Hugging Face generation configuration
- OpenAI API structured output reference
- vLLM structured output guide
- JSON Schema
- Pydantic models and validation