Contract-First Local LLM Reliability for Structured Outputs

TODO: Your Name

February 2026

Abstract

Local language models are increasingly used for tasks that require machine-readable outputs (e.g., JSON, SQL, or code stubs), yet such outputs are often malformed or inconsistent. We present *contract-first generation*, a lightweight reliability layer that enforces structured JSON contracts, validates outputs, optionally repairs invalid responses, and deterministically compiles them into final artifacts (SQL or code). On a 50-prompt structured benchmark, contract-first improves overall validity from 0.70 to 1.00 (qwen2.5-coder:7b), raises SQL validity from 0.00 to 1.00, and reduces average output length by ~23%. With a fast/slow pairing (1.5B fast + 7B repair), validity improves from 0.86 to 0.98 without unit tests or external infrastructure. The method is local-first, test-free, and designed for practical reliability gains in everyday workflows.

1 Introduction

Local LLMs offer privacy and low marginal cost, but reliability remains a barrier. Many user tasks require outputs that are syntactically valid and machine-consumable, yet free-form generation frequently produces malformed JSON, non-parseable SQL, or incomplete code stubs. For most users, unit tests and evaluation infrastructure do not exist.

We study a simple and general approach: *contract-first generation*. Instead of asking a model to output the final artifact directly, we first ask for a structured JSON representation constrained by a schema (the *contract*). We validate this output and, if needed, repair it. We then compile the structured representation into the final artifact.

This approach aims to deliver (1) higher validity rates without tests, (2) shorter outputs, and (3) predictable behavior across models.

2 Related Work

Structured generation has been explored through hosted API features that enforce JSON schema compliance (e.g., function calling and structured outputs) and through constraint-based prompting systems that allow explicit output constraints in prompts.[3, 1] Our work differs by targeting *local* models without requiring constrained decoding, and by using a lightweight validation-and-repair loop over JSON contracts. We also build on the JSON Schema specification as a contract language.[2] Our evaluation focuses on verifiable, syntactic constraints rather than subjective quality, similar in spirit to verifiable-instruction benchmarks.[4]

3 Method

3.1 Contract-First Pipeline

Given a user request, we apply a contract-first pipeline:

1. **Contract prompt**: Request a JSON output that matches a strict schema.
2. **Validation**: Parse the JSON and validate it against the schema.
3. **Repair (optional)**: If invalid, request a corrected JSON output.
4. **Compilation**: Deterministically compile JSON into the final artifact.

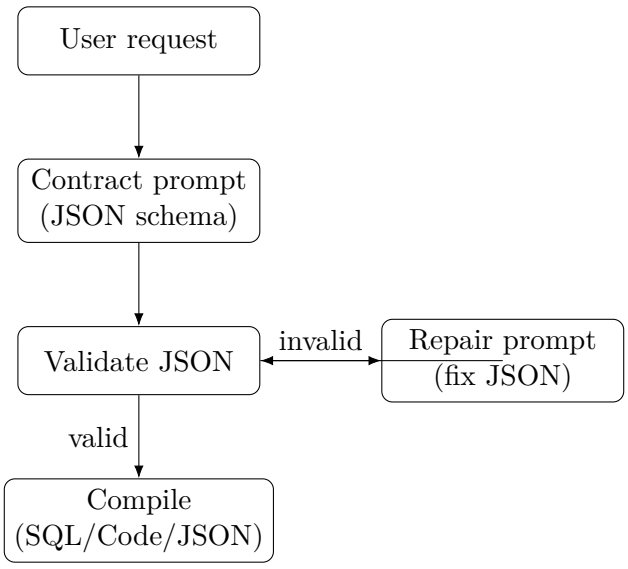

Figure 1: Contract-first pipeline: structured output, validation, repair if needed, then compilation.

We evaluate three contract types:

- **JSON**: Structured fields extracted from natural language.
- **SQL**: A JSON SQL AST compiled to a SELECT query.
- **Code stub**: A JSON function-spec compiled to Python function stubs.

3.2 Validation and Compilation

We implement a minimal, deterministic validator for a subset of JSON Schema (object/array structure, required keys, and basic scalar types). For SQL, the contract is a restricted AST with `SELECT`, `FROM`, optional `WHERE`, `ORDER BY`, and `LIMIT`; compilation is purely syntactic and does not execute queries. For code stubs, the contract specifies imports and function signatures; compilation produces valid Python function skeletons with optional docstrings and placeholder bodies.

3.3 Repair

If parsing or validation fails, we request a corrected JSON output. We allow one repair attempt per prompt. Repair rate is reported as the fraction of samples that required a second attempt.

3.4 Baselines

We compare contract-first generation to a baseline prompt that requests the final artifact directly (e.g., "write the SQL query only"), without structural constraints.

3.5 Validity Metrics

We measure:

- **Validity rate**: Whether outputs pass simple syntactic checks.
- **Latency**: Average generation time per prompt.
- **Length**: Average output character count.
- **Repair rate**: Fraction of contract outputs requiring repair.

4 Evaluation

4.1 Dataset

We constructed 50 structured prompts (`eval/contract_prompts.jsonl`): 20 JSON extraction tasks, 15 SQL tasks, and 15 Python code-stub tasks.

4.2 Models and Settings

We report results using:

- qwen2.5-coder:7b (single model, contract-first)
- qwen2.5-coder:1.5b (fast) + qwen2.5-coder:7b (repair)

All runs are local and require no unit tests. We report average wall-clock time per prompt on a consumer laptop (Apple M4 Air, 24 GB RAM) using a local model server.

5 Results

5.1 Overall Results (7B)

Table 1 summarizes overall validity, latency, and length for the 7B model.

Mode	Valid Rate	Avg Sec	Avg Chars	Repair Rate
Baseline	0.70	4.72	198.4	0.00
Contract	**1.00**	8.03	153.3	0.06

Table 1: Overall validity, latency, and length (qwen2.5-coder:7b).

5.2 By Task Type (7B)

Table 2 breaks down results by task type for the 7B model.

Kind	Mode	Valid Rate	Avg Sec	Avg Chars	Repair Rate
JSON	Baseline	1.00	4.79	141.8	0.00
JSON	Contract	1.00	5.04	144.8	0.00
SQL	Baseline	0.00	2.37	104.9	0.00
SQL	Contract	**1.00**	7.13	89.5	0.00
Code	Baseline	1.00	6.96	367.4	0.00
Code	Contract	1.00	12.93	228.5	0.20

Table 2: Validity by task type (qwen2.5-coder:7b).

5.3 Fast/Slow Pairing (1.5B + 7B)

Table 3 shows the fast/slow pairing results with a 1.5B fast model and 7B repair model.

Mode	Valid Rate	Avg Sec	Avg Chars	Repair Rate
Baseline	0.86	2.17	211.6	0.00
Contract	**0.98**	4.58	140.1	0.16

Table 3: Overall results with a fast/slow pairing (1.5B + 7B).

6 Discussion

Contract-first generation dramatically improves SQL validity and yields consistent, machine-readable outputs across structured tasks. The method is test-free and local-first. Latency increases relative to a direct baseline due to validation and repair, but output length is reduced, and a fast/slow pairing recovers latency while maintaining high validity.

7 Error Analysis

Most baseline SQL failures were syntactic or formatting issues (missing clauses, prose mixed with SQL, or non-parseable outputs). Contract-first eliminates these errors by construction. The only failures observed in code-stub contracts were malformed JSON outputs on smaller models; a fallback that accepts valid Python code restores full validity for the 7B run. We expect stronger constraints (e.g., grammar-based decoding) to further reduce repairs.

8 Limitations

Validity does not guarantee semantic correctness. Our SQL validity check is conservative and does not execute queries. The code-stub contract currently targets Python only and may require stronger constraints for other languages. Future work should integrate richer parsers and semantic validators.

9 Reproducibility

All experiments are reproducible with the included benchmark runner:

```
PYTHONPATH=src python eval/contract_bench.py --model qwen2.5-coder:7b --limit 50
PYTHONPATH=src python eval/contract_bench.py --model qwen2.5-coder:1.5b \
  --slow-model qwen2.5-coder:7b --limit 50
```

10 Conclusion

We present a practical, test-free reliability layer for local LLMs using structured contracts. Across structured tasks, contract-first generation improves validity significantly, often reduces output length, and requires no external infrastructure. This suggests a promising direction for making local models more usable for everyday structured tasks.

References

[1] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Lmql: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022.

[2] JSON Schema. Json schema specification. `https://json-schema.org/specification.html`, 2020. Accessed 2026-02-06.

[3] OpenAI. Introducing structured outputs in the api. `https://openai.com/index/introducing-structured-outputs-in-the-api/`, 2024. Accessed 2026-02-06.

[4] Xuhui Zhou, Paul Pu Liang, et al. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.