SchemaGen: Neuro-Symbolic JSON Schema Generation

Anonymous Authors[1]

Abstract

Large Language Models (LLMs) have become powerful generators of structured data, yet their ability to *design* data contracts—rather than merely comply with them—remains underexplored. We introduce **SchemaBench**, the first benchmark evaluating LLMs as schema architects, testing their capacity to create valid, specific, and semantically correct JSON Schemas across varying structural complexity, constraint hardness, and ambiguity levels. Complementing this, we present **SchemaGraph Critic**, a neuro-symbolic middleware that represents JSON Schemas as heterogeneous graphs and employs a Graph Neural Network (specifically, a Heterogeneous Graph Transformer) to detect structural logic errors invisible to standard validators—such as dangling references, circular dependencies, and constraint conflicts. Unlike constrained decoding, which ensures syntactic validity, our approach validates semantic correctness and generates natural language feedback for iterative LLM refinement. Together, SchemaBench and SchemaGraph Critic establish a rigorous framework for evaluating and improving LLM-generated data contracts.

1. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, including structured data formats like JSON. However, a critical gap exists in evaluating and improving their ability to *design* data contracts—not merely follow schemas, but create them. JSON Schema, the de facto standard for describing JSON document structure, presents unique challenges:

1. **Long-distance dependencies**: Schemas contain references (`$ref`) that can span hundreds of tokens, creating dependencies invisible to autoregressive models.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

2. **Structural logic**: Valid JSON syntax does not guarantee valid schema semantics (e.g., `minItems: 10`, `maxItems: 3` is syntactically correct but logically impossible).

3. **Recursive definitions**: Self-referential schemas require reasoning about infinite structures.

We present two complementary contributions:

- **SchemaBench**: A comprehensive benchmark evaluating LLMs as both "architects" (schema designers) and "builders" (instance generators).
- **SchemaGraph Critic**: A neuro-symbolic middleware using Graph Neural Networks (GNNs) to validate structural logic and provide corrective feedback to LLMs.

Our key insight is that *constrained decoding solves syntax, but not structural logic*. By representing schemas as heterogeneous graphs, we enable GNNs to reason about relationships that are inherently non-sequential.

2. Related Work

2.1. Structured Output Generation from LLMs

Recent work has focused on constraining LLM outputs to valid structures. Constrained decoding techniques ensure syntactically valid JSON through grammar-guided generation. Function calling APIs from major providers allow models to output structured data. However, these approaches guarantee syntactic validity but not semantic correctness.

2.2. JSON Schema Validation

Standard validators (e.g., `ajv`, `jsonschema`) check technical compliance with the JSON Schema specification. They cannot detect logical constraint conflicts, unreachable definitions, or semantic inconsistencies between schema intent and structure.

2.3. Graph Neural Networks for Structured Data

GNNs have shown success in code vulnerability detection, program analysis, and knowledge graph reasoning.

Hu et al. (2020) introduced Heterogeneous Graph Transformers (HGT), which learn different attention weights for different edge types. Our work extends this paradigm to JSON Schema validation, treating schemas as heterogeneous graphs with typed nodes and edges.

2.4. LLM Evaluation Benchmarks

Existing benchmarks focus on code correctness or test compliance with given schemas. **SchemaBench** is the first to evaluate schema *design* capability.

3. SchemaBench: A Benchmark for Schema Design

3.1. Motivation

Existing benchmarks test: "Given schema S, generate valid instance I." We ask the harder question: "Given requirements R, design schema S and generate instance I that satisfies S."

This evaluates the model as:

1. **The Architect**: Can the model create a valid, specific, and semantically correct JSON Schema?
2. **The Builder**: Can the model generate an instance that strictly adheres to its own schema?

3.2. Benchmark Architecture

3.2.1. Tracks (Scenario Categories)

We organize scenarios by complexity into three tracks:

Table 1. SchemaBench Tracks

Track	Challenges
A: Structural	Flat configs, nested objects, recursive trees, polymorphic unions
B: Constraints	Regex patterns, numerical bounds, format validation
C: Ambiguity	Reasonable defaults, implied fields, domain knowledge

3.2.2. Scenario Specification

Each scenario contains:

- `prompt_template`: Natural language description of requirements
- `required_constraints`: Expected JSON Schema keywords (e.g., `minimum`, `pattern`, `$ref`)
- `gold_keys`: Expected property names in the generated schema

3.3. Evaluation Pipeline

We implement a multi-gate evaluation system:

Gate 1: Syntax & Meta-Validity. Is the output valid JSON? Is the schema a valid JSON Schema (Draft 2020-12)?

Gate 2: Self-Consistency. Does the generated instance validate against the generated schema? This includes strict format checking (email, date, URI).

Gate 3: Semantic Alignment. We measure:

- **Constraint Recall (CR)**: Does the schema use the expected constraint types?

$$\text{CR} = \frac{|\text{Used} \cap \text{Required}|}{|\text{Required}|} \tag{1}$$

- **Specificity Score**: Ratio of constrained fields to total fields.
- **Key Coverage**: Does the schema define expected properties?

3.4. Dataset Statistics

Table 2. SchemaBench Dataset

Category	Scenarios	Difficulty
Structural: Flat	5	Easy
Structural: Nested	8	Medium
Structural: Recursive	6	Hard
Structural: Polymorphic	4	Hard
Constraint Hardness	10	Medium–Hard
Ambiguity Resolution	7	Variable
Total	**40**	—

4. SchemaGraph Critic

4.1. Motivation

Standard validators check if a schema is *technically* legal. They cannot determine if a schema is "good," efficient, or free of structural hallucinations.

The core insight:

- **LLMs** see JSON as a sequence of tokens (1D). They struggle with long-distance dependencies.
- **GNNs** see JSON as a topology (graph). They instantly see that Node A refers to Node Z, regardless of token distance.

4.2. System Architecture

The SchemaGraph Critic operates in a feedback loop:

1. **Generation**: LLM generates a candidate JSON Schema
2. **Graphification**: Parser converts JSON to a heterogeneous graph
3. **Inference**: SchemaGNN produces validity score and defect nodes
4. **Feedback**: Translator converts tensor outputs to natural language
5. **Refinement**: Corrective prompt sent to LLM for iteration

4.3. Heterogeneous Graph Representation

4.3.1. Node Types

We define 11 node types representing schema elements:

Table 3. Node Types in Schema Graphs

Type	Description
`OBJECT`	Object schema block
`ARRAY`	Array schema block
`STRING`, etc.	Primitive type schemas
`REF`	Reference pointer (`$ref`)
`DEFINITION`	Reusable definition
`LOGIC`	Logical combinator (anyOf, oneOf, allOf)

4.3.2. Edge Types

We define 5 edge types capturing relationships:

Table 4. Edge Types in Schema Graphs

Type	Semantics
`CONTAINS`	Object property nesting
`ITEMS`	Array element schema
`REFERS_TO`	Reference resolution
`LOGIC`	Logical alternatives
`ADDITIONAL`	Extra property schema

4.3.3. Node Feature Engineering

Each node has a 404-dimensional feature vector:

Table 5. Node Features (404 dimensions)

Feature	Dims	Source
Semantic embedding	384	MiniLM-L6
Type one-hot	11	Node type
Depth	1	Nesting level
Constraint flags	8	Constraint presence

4.4. SchemaGNN Model

4.4.1. Backbone: Heterogeneous Graph Transformer

We use HGT layers (Hu et al., 2020) because they learn different attention weights for different edge types. This is crucial because `REFERS_TO` edges are "teleportation tunnels" for information flow, while `CONTAINS` edges represent standard hierarchical relationships.

Architecture:

- Input projection: Linear$(404 \to 256)$
- HGT layers: 3 layers, 4 attention heads
- Residual connections with LayerNorm

4.4.2. Prediction Heads

Global Critic Head (Graph-level binary classification):

$$p(\text{valid}) = \sigma\left(\text{MLP}([\bar{h}_{\text{mean}}; \bar{h}_{\text{max}}])\right) \tag{2}$$

where $\bar{h}_{\text{mean}}$ and $\bar{h}_{\text{max}}$ are mean and max pooled node embeddings.

Local Debugger Head (Node-level error detection):

$$p(\text{error}_i) = \sigma(\text{MLP}(h_i)) \tag{3}$$

Outputs per-node probability of being the root cause of an error.

4.5. Training Data Synthesis

We cannot train on valid schemas alone—the GNN must see mistakes to learn to catch them.

4.5.1. The Corruptor

We systematically break valid schemas with 7 corruption types:

4.6. Loss Function

Combined loss for joint training:

$$\mathcal{L} = \lambda_g \cdot \text{BCE}(\hat{y}, y) + \lambda_l \cdot \frac{1}{|V|} \sum_{i \in V} \text{BCE}(\hat{e}_i, e_i) \tag{4}$$

Table 6. Corruption Types for Training

Type	Description
`DANGLING_REF`	$ref to non-existent definition
`CIRCULAR_REF`	Infinite reference loop
`TYPE_MISMATCH`	Type conflicts with structure
`CONSTRAINT_CONFLICT`	Impossible constraints
`MISSING_REQUIRED`	Required property undefined
`INVALID_PATTERN`	Malformed regex
`WRONG_ITEMS_TYPE`	Invalid items schema

where $\hat{y}$ is the predicted validity score, $\hat{e}_i$ is the predicted error probability for node i, and $\lambda_g = 1.0$, $\lambda_l = 0.5$.

4.7. Feedback Translation

The GNN outputs tensors, but the LLM needs text. Our translator:

1. Maps node IDs to JSON paths via stored metadata
2. Generates severity levels from probabilities
3. Applies issue templates based on node types
4. Produces structured feedback prompts

5. Experiments

5.1. Experimental Setup

Models Evaluated on SchemaBench:

- GPT-4, GPT-4-Turbo
- Claude 3.5 Sonnet
- Llama 3.1 (70B, 405B)
- Gemini Pro 1.5

SchemaGNN Training:

- Source schemas: SchemaStore, GitHub repositories (10K+ schemas)
- Training split: 80/10/10 (train/val/test)
- Hardware: NVIDIA A100 (40GB)
- Training time: $\sim$4 hours for 50 epochs

5.2. SchemaBench Results

5.3. SchemaGraph Critic Results

5.4. End-to-End Pipeline Evaluation

Research Question: Does GNN-guided feedback improve LLM schema generation?

Table 7. SchemaBench Results by Track (Pass Rates %)

Model	Gate 1	Gate 2	Gate 3
GPT-4-Turbo	—	—	—
Claude 3.5 Sonnet	—	—	—
Llama 3.1 405B	—	—	—
Gemini Pro 1.5	—	—	—

Results pending experimental runs.

Table 8. SchemaGNN vs. Baseline Performance

Method	Acc.	Prec.	Recall	F1
Combined Baseline[†]	75.4%	96.4%	68.6%	80.2%
SchemaGNN (Ours)	**78.1%**	83.9%	**84.1%**	**84.0%**

[†]Combined baseline includes reference checking, cycle detection, and constraint validation.

Protocol:

1. LLM generates initial schema (Round 0)
2. SchemaGraph Critic analyzes and provides feedback
3. LLM refines schema (Round 1–N)
4. Measure improvement across rounds

6. Analysis and Discussion

6.1. Why GNNs Outperform Text-Based Validators

Our results demonstrate that SchemaGNN achieves 84.0% F1 compared to 80.2% for the combined deterministic baseline. The key advantage is **recall**: SchemaGNN detects 84.1% of invalid schemas, while baselines only catch 68.6%.

This gap exists because deterministic methods check for specific patterns, while the GNN learns to recognize structural anomalies holistically. For example, consider an LLM-generated "Binary Tree" schema where `left` is defined but `right` refers to a non-existent node:

6.2. Error Type Analysis

As shown in Table 9, the model achieves 100% precision across all corruption types—when it flags an error, it is always correct. Recall varies by corruption type:

- **Graph-structural errors** (circular refs 98.1%, dangling refs 92.1%): Detected most reliably because they manifest as clear topological anomalies.
- **Constraint errors** (conflicts 89.4%, patterns 84.3%): Require combining constraint flag features with structural context.
- **Semantic errors** (missing required 72.8%): Hardest to detect as they require understanding schema intent.

Table 9. Per-Corruption-Type Detection (SchemaGNN)

Corruption Type	**Prec.**	**Recall**	**F1**
`CIRCULAR_REF`	100.0%	98.1%	99.0%
`DANGLING_REF`	100.0%	92.1%	95.9%
`CONSTRAINT_CONFLICT`	100.0%	89.4%	94.4%
`TYPE_MISMATCH`	100.0%	84.9%	91.8%
`INVALID_PATTERN`	100.0%	84.3%	91.5%
`WRONG_ITEMS_TYPE`	100.0%	82.1%	90.2%
`MISSING_REQUIRED`	100.0%	72.8%	84.3%

Table 10. Ablation Study: Architecture Comparison

Architecture	**Global F1**	**Accuracy**
HGT (Ours)	**82.6%**	**76.5%**
GCN (Homogeneous)	76.7%	65.7%
GAT (Homogeneous)	5.8%	36.3%

HGT significantly outperforms homogeneous baselines, validating the importance of edge-type-aware attention.

6.3. Ablation: Heterogeneous vs. Homogeneous

Table 10 shows that HGT outperforms GCN by 5.9% F1 and GAT fails entirely (5.8% F1). This validates our hypothesis: the model must distinguish between edge types. A `REFERS_TO` edge is a "teleportation tunnel" that should propagate information differently than a `CONTAINS` edge representing hierarchy.

6.4. Limitations

1. **Schema Coverage**: Current training focuses on JSON Schema Draft 2020-12.
2. **Semantic Understanding**: The GNN detects structural errors, not semantic misalignment with requirements.
3. **Training Data Bias**: Synthetic corruption may not capture all real-world error patterns.

7. Conclusion

We presented **SchemaBench**, the first benchmark for evaluating LLM schema design capability, and **SchemaGraph Critic**, a neuro-symbolic middleware that validates structural logic in JSON Schemas using Graph Neural Networks.

Our contributions:

1. A rigorous evaluation framework distinguishing syntax from semantics
2. A novel graph-based representation of JSON Schema structure
3. A feedback loop enabling iterative schema refinement

Table 11. Validator Comparison on Binary Tree Schema

Validator	**Result**	**Reason**
Standard (`ajv`)	✓	Syntactically correct
Constrained Decoding	✓	All brackets matched
SchemaGraph Critic	×	`REFERS_TO` edge points to non-existent node

This work advances the reliability of LLM-generated structured outputs, enabling safer deployment in applications requiring data contracts.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning by improving the reliability of LLM-generated structured data. Potential positive impacts include safer API integrations, reduced data validation errors in production systems, and better tooling for developers. We do not foresee specific negative societal consequences beyond general concerns about over-reliance on automated systems.

References

Hu, Z., Dong, Y., Wang, K., and Sun, Y. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020 (WWW)*, pp. 2704–2710. ACM, 2020.

A. SchemaBench Scenario Examples

A.1. Example: Recursive File System (Hard)

Prompt: Create a recursive JSON Schema for a File System Node. A Node must have 'name' (string), 'type' (enum: 'file', 'directory'). If type is 'file', it must have 'size' (integer > 0). If type is 'directory', it must have 'children' (array of Nodes). The schema must use recursion ($ref).

Required Constraints: `$ref`, `enum`, `if`, `then`, `minimum`

Gold Keys: `name`, `type`, `children`, `size`

A.2. Example: Polymorphic UI Components (Hard)

Prompt: Create a schema for a 'Page' containing a list of 'components'. The components array can contain mixed types: Button (`type: 'button', label: string, onClick: string`), Image (`type: 'image', src: url, width: integer`), Text (`type: 'text', content: string`). Use 'oneOf' to ensure that a Button cannot have a 'src' and an Image cannot have an 'onClick'.

Required Constraints: `oneOf`, `const`, `required`

B. SchemaGNN Hyperparameters

Table 12. Training Configuration

Parameter	**Value**
Hidden dimension	256
Attention heads	4
HGT layers	3
Dropout	0.1
Learning rate	1×10^{-4}
Batch size	32
Epochs	50
Global loss weight (λ_g)	1.0
Local loss weight (λ_l)	0.5

C. Feedback Translation Templates

The translator uses node-type-specific templates:

- **REF**: "This reference may point to a non-existent or incorrectly named definition. Verify that the $ref target exists."
- **ARRAY**: "This array definition may have invalid items schema or constraint conflicts. Verify minItems/maxItems are consistent."
- **LOGIC**: "This logical operator may have inconsistent options. Ensure all options are valid and don't conflict."
- **OBJECT**: "This object may have structural issues. Check that required properties are defined."