ThinkJSON: Multi-Reward GRPO for Reliable JSON Schema Adherence in Small Language Models

Abstract

LLMs are increasingly used to convert semi-structured operational text (batch records, QC checklists, deviation logs) into structured JSON for analytics and regulatory traceability in biomanufacturing and pharma (Labant, 2025; Erdmann et al., 2022; Aher et al., 2025). In these settings, "mostly correct" is useless. The output must be (i) valid JSON, (ii) faithful to an approved schema (no missing required fields, no hallucinated keys), and (iii) auditable: every value must be explainable and defensible to QA and regulators (Liu et al., 2024b,a; Erdmann et al., 2022).

We show two complementary routes. ThinkJSON-Base trains a 1.5B Qwen-2.5 model end-to-end on fully synthetic paired data (noisy narrative $\rightarrow$ gold JSON) with GRPO then light SFT (Team, 2024; Madaan et al., 2023). ThinkJSON-Pro adapts a 7B Qwen model with LoRA using noisy, realistic schemas and adds a semantic judge reward. Both runs are compute-feasible: $\sim$20 GPU-hours on $8\times$H100 + $\sim$3 hours SFT on $1\times$A100 for ThinkJSON-Base, and single-A100 LoRA for ThinkJSON-Pro (3.5 h wall-clock).

We evaluate using **parsability-aware** metrics. We report not only Match/Noise, but also Parse Success (`json.loads` passes) and *Adjusted Match* = Match $\times$ ParseSuccess and *Adjusted Noise* = Noise $\times$ ParseSuccess $+100\% \times (1 -$ ParseSuccess). Under these stricter metrics, a frontier model (DeepSeek-R1, $\sim$671B) unsurprisingly wins (DeepSeek-AI, 2025), but ThinkJSON-Pro (7B + LoRA) is the strongest sub-10B alternative: $\sim$81.9% parse success with usable adjusted match while running on a single A100. Gemini-style constrained decoding looks strong on raw Match/Noise but collapses once parseability and adjusted noise are enforced (Google AI Team, 2025; Dong et al., 2024).

Takeaway: multi-reward GRPO makes strict schema adherence an internal behavior of the model, not something bolted on at inference time. This yields auditable, parseable JSON from mid-size models that can actually run on-prem in regulated environments.

1 Introduction

Regulated manufacturing (biomanufacturing, pharma QA, cleanroom operations) still runs on paper or semi-structured Word / PDF artifacts (Labant, 2025). Even incremental digitization has compliance value, but only if the structured record is *exact*, traceable, and survives audit (Erdmann et al., 2022; Aher et al., 2025).

This creates a sharp requirement for LLMs:

> Given messy text, output **one JSON object** that (i) parses, (ii) exactly matches a known schema (required keys present, no illegal keys), (iii) has values grounded in the source text, and (iv) includes an auditable reasoning trail that can be inspected by QA.

Prompt engineering alone is fragile (Liu et al., 2024b). Post-processing scripts and regex cleanup become brittle as schemas evolve. Grammar-constrained decoding, or "JSON mode" inference APIs, can force syntactic validity but often add latency, struggle with deep nesting, and still hallucinate semantically wrong values (Willard and Louf, 2023; Dong et al., 2024; Geng et al., 2025; Google AI Team, 2025). In short: correctness at runtime is expensive and still not guaranteed.

We claim that strict schema adherence should be *trained*, not duct-taped. We introduce a reinforcement-learning setup that explicitly rewards four behaviors:

1. exact output framing,
2. structural completeness,
3. semantic agreement,

4. and auditability.

We show this can be done on models small enough to run on-prem, without sending batch records to a proprietary frontier LLM (DeepSeek-AI, 2025; Li et al., 2024).

2 Task and Contributions

Task. Input: unstructured or semi-structured process text x (batch log, QC note, deviation write-up). Output:

> <think> rationale mapping fields in the schema to spans in x </think>
> <answer> { strictly valid JSON following the schema } </answer>

The <answer> block is what ETL consumes. It *must* parse with `json.loads()`. The <think> block is kept for audit. This split matches reality: quality reviewers and regulators routinely ask "show me where you pulled that field from" (Erdmann et al., 2022; Aher et al., 2025).

Practical constraints. The model must survive: (i) nested schemas, (ii) required vs. optional keys, (iii) timestamps and escaped characters, (iv) operator shorthand ("line3 CIP tank passed"), (v) layout noise (OCR scraps, tables in text) (Labant, 2025; Liu et al., 2024b).

Contributions.

1. **Multi-reward GRPO for schema adherence.** We define verifiable rewards for (a) format discipline via <think>/<answer>, (b) structural completeness and no junk keys, (c) key–value match, and (d) a judge-LLM semantic score (Wang et al., 2025; DeepSeek-AI, 2025; Shao et al., 2024; Wang et al., 2023). We optimize them jointly with GRPO in a compute-efficient way (Dann et al., 2023).

2. **Two training routes with realistic budgets.**
   - ThinkJSON-Base: GRPO + brief SFT on a 1.5B Qwen-2.5 base using fully synthetic paired data (narrative ↔ gold JSON) generated using instruction-style prompting (Team, 2024; Madaan et al., 2023).
   - ThinkJSON-Pro: GRPO with LoRA adapters on a 7B Qwen base using noisy, real schemas plus a semantic judge reward. Training completed in 3.5 h on a single A100.

3. **Parsability-aware evaluation.** We multiply semantic Match by Parse Success and inflate Noise when parsing fails. This exposes real deployability gaps that plain Match/Noise numbers hide (Geng et al., 2025; Google AI Team, 2025).

4. **Audit-ready reasoning traces.** The forced <think> block gives an immediately reviewable map from text spans to schema fields. That satisfies GxP-style traceability demands (Erdmann et al., 2022; Aher et al., 2025).

3 Method

3.1 Two Training Routes

We use the same RL skeleton but two data regimes.

ThinkJSON-Base (Synthetic → SFT). We automatically build triplets

$$(\text{noisy narrative text},\ \text{rationale},\ \text{gold JSON})$$

using Qwen-14B/32B style prompting to generate both the structured JSON instance and multiple paraphrased narratives with layout noise, OCR-like glitches, timestamp weirdness, and table fragments (Team, 2024; Madaan et al., 2023).

We then apply GRPO on a 1.5B Qwen-2.5 base. After RL converges, we run a short supervised fine-tuning (SFT) pass to stabilize naming and timestamp style. Total cost: ∼20 GPU-hours on 8×H100 for GRPO and ∼3 hours on 1×A100 for SFT.

ThinkJSON-Pro (Real → LoRA). We switch to real noisy schemas (deep nesting, optional sub-blocks, inconsistent naming, escaped sequences) and deployment-style text that encodes the same facts without surfacing the schema. We fine-tune LoRA adapters on a 7B Qwen-2.5 base under GRPO. Here we also add a *judge-LLM reward* (Section 3.3) because literal string equality is too strict in production text (Li et al., 2024; Geng et al., 2025; Wang et al., 2025).

All of ThinkJSON-Pro trains on a single A100. That matters because regulated orgs usually cannot ship batch records off-prem or expose them to a proprietary frontier LLM for either training or inference (Erdmann et al., 2022; Aher et al., 2025).

3.2 Output Contract

Every generation must follow:

<think> ... step-by-step mapping from text spans to schema fields ... </think>
<answer> { JSON only, no commentary } </answer>

Hard rules: exactly one <think> block, exactly one <answer> block, no output after </answer>, no JSON outside <answer>. Any violation → zero reward.

This mirrors "reason then answer" RL setups for math and step-by-step reasoning (DeepSeek-AI, 2025; Shao et al., 2024; Wang et al., 2023), except the "answer" is now governed JSON, not a scalar.

3.3 Multi-Reward GRPO

We train with GRPO (DeepSeek-AI, 2025; Dann et al., 2023): sample k candidates per prompt, score them, compute group-relative advantages, and update toward higher-reward candidates with a KL penalty.

For each candidate completion c and gold JSON g:

1. Format reward $R_{\text{fmt}} \in \{0, 1\}$. Check the contract: (a) exactly one <think> block, (b) exactly one <answer> block, (c) no stray text after </answer>, (d) JSON appears *only* inside <answer>. Fail any: $R_{\text{fmt}} = 0$.

2. Parse & structure reward $R_{\text{struct}} \in [0, 1]$. Extract <answer>, run `json.loads`. If parsing fails: 0. Else: reward coverage of all required keys, penalize illegal keys, and reward structural completeness down to nested fields.

3. Key–value reward $R_{\text{kv}} \in [0, 1]$. Over overlapping keys between predicted JSON and gold g: reward exact value matches. Also compute a length/shape ratio $\frac{\min(|\hat{J}|,|g|)}{\max(|\hat{J}|,|g|)}$ to punish bloated hallucinated blobs (Liu et al., 2024b; Geng et al., 2025).

4. Judge reward $R_{\text{judge}} \in [0, 1]$. Only in ThinkJSON-Pro. A fixed judge LLM returns 1.0 for exact, 0.8 for minor formatting drift, down to 0.0 for fabricated/missing content. This handles fuzzy mapping in real logs ("CIP line3 sterile") vs structured schema keys (Wang et al., 2025; Li et al., 2024).

We clip and weight these rewards into $R_{\text{total}} \in [0, 1]$, then apply GRPO. We do *not* modify GRPO internals.

Algorithm 1 Reward extraction for one completion c

Require: completion c, gold JSON g, schema S
Extract <think> and <answer> blocks
if blocks missing OR trailing text after </answer> **then**
  $R_{\text{fmt}} = 0$; **return** all zeros
else
  $R_{\text{fmt}} = 1$
end if
Try `json.loads` on <answer> → $\hat{J}$
if parse fails **then**
  $R_{\text{struct}} = R_{\text{kv}} = R_{\text{judge}} = 0$; **return**
end if
Check required / illegal keys vs. $S \rightarrow R_{\text{struct}}$
Compare overlapping key–values with g; compute shape ratio → R_{kv}
if using judge LLM **then**
  $R_{\text{judge}} = \text{JudgeLLM}(\hat{J}, g)$
else
  $R_{\text{judge}} = 0$
end if
$R_{\text{total}} = w_1 R_{\text{fmt}} + w_2 R_{\text{struct}} + w_3 R_{\text{kv}} + w_4 R_{\text{judge}}$
return all rewards

4 Experimental Setup

4.1 Models and Compute

ThinkJSON-Base: Qwen-2.5 1.5B base (Team, 2024). GRPO for ∼20h on 8×H100. Short SFT (∼3h on 1×A100). Synthetic paired corpus (Section 3.1): each sample has narrative text, rationale, and gold JSON. Synthetic data is produced with self-instruct style prompting (Madaan et al., 2023) and reasoning distillation in the style of DeepSeek-R1 (DeepSeek-AI, 2025; Shao et al., 2024).

ThinkJSON-Pro: Qwen-2.5 7B base (Team, 2024). LoRA adapters + GRPO with judge reward on noisy, realistic schemas drawn from internal manufacturing-style records and publicly documented schema-alignment corpora such as the Schema Reinforcement Learning dataset (Anonymous, 2025). Single A100 end-to-end. This reflects a real on-prem budget where data cannot leave the facility (Erdmann et al., 2022; Aher et al., 2025).

Baselines:

- DeepSeek-R1 (∼671B) (DeepSeek-AI, 2025): frontier RL-for-reasoning model.
- Gemini 2.0 Flash in structured-output / JSON mode (Google AI Team, 2025).
- Qwen2.5-7B-instruct baseline (Team, 2024) (no RL schema reward).
- XGrammar / JSONFormer: grammar-constrained decoding libraries. On our long, nested schemas they incurred high latency

Corpus	Train	Test
Synthetic paired (narrative → JSON)	100,000	20,000
Realistic noisy schemas (overall)	100,000	20,000
Complex subset	stratified within real	
Custom subset	stratified within real	
Escape subset	stratified within real	

Table 1: Dataset sizes and splits. Subset proportions are preserved in both train and test.

and frequent non-completions; we therefore track timeouts rather than accuracy.

4.2 Datasets

Reproducibility note. To reproduce subsets: compute schema depth, operators, and array-nesting from the JSON Schema AST; flag `patternProperties`/regex and escaped tokens in examples; then stratify with a fixed seed.

Synthetic paired corpus. We generate **100,000** training and **20,000** test pairs (narrative → gold JSON) using self-instruct prompting and layout noise, covering deep nesting, optional blocks, timestamps, escaped characters, and table fragments. Each item contains (*i*) unstructured narrative, (*ii*) brief rationale, and (*iii*) schema-faithful JSON.

Realistic noisy schemas. We construct **100,000** train and **20,000** test items from public JSON schemas collected via the *JSON Schema Store* (jso) and a small set of hand-authored manufacturing-style schemas (no proprietary records). For each schema, we synthesize deployment-style narratives that obscure structure while preserving facts, then supervise to the schema-conformant JSON.

Splits used in tables (Complex / Custom / Escape). We stratify the real-schema pool into three disjoint subsets:

- **Complex**: depth $\geq$ 4 *or* presence of `anyOf`/`oneOf`/`allOf`/`if{then{else` *or* nested arrays $\geq$ 2 levels *or* $|\texttt{properties}| \geq 50$.
- **Custom**: our hand-authored manufacturing schemas reflecting QA/Batch-record structure.
- **Escape**: schemas whose examples/patterns include escaped characters (¨, , \n, Unicode escapes) or heavy `patternProperties`/regex.

Train/test is 100k/20k overall with per-subset proportions preserved. We deduplicate by schema `$id` and a canonicalized shape hash. Public schemas retain their original licenses; only machine-generated narratives are included.

4.3 Metrics

Classic: **Match** = fraction of correctly populated fields. **Noise** = fraction of junk or extraneous tokens/keys.

These look fine even when a model emits garbage that is not valid JSON (Geng et al., 2025). So we add:

- **Parse Success**: % of generations whose `<answer>` parses with `json.loads()`.
- **Adjusted Match**: Match × ParseSuccess.
- **Adjusted Noise**: Noise × ParseSuccess $+100\% \times (1 - \text{ParseSuccess})$.

If the JSON does not parse, Adjusted Match $\to 0$ and Adjusted Noise $\to 100\%$. That reflects production reality: unparseable output is 100% waste (Liu et al., 2024b; Geng et al., 2025).

We also track **ValidJSON** (syntactic validity) and **NoOutput** (model refuses or returns nothing).

5 Results

We report results in three views: (1) a controlled synthetic benchmark where ground truth JSON is clean and fully aligned with the source text, (2) a noisy real-schema setting that matches deployment conditions, and (3) a parse-aware view that exposes which models actually produce JSON you can use in ETL without human repair.

All reported scores use the train/test sizes in Table 1 and the subset rules of §4.2.

We summarize each view with its own table to keep the comparisons readable.

5.1 Synthetic Benchmark: Clean Paired Data

The synthetic benchmark pairs each unstructured narrative with an exact gold JSON instance. Narratives include paraphrases, reordered facts, and light layout noise, but they still reflect the schema cleanly. This benchmark measures whether a model can (a) populate the right keys, (b) avoid inventing keys, and (c) avoid emitting garbage tokens.

We evaluate:

- **Match (%)**: fraction of fields correctly mapped into the schema.
- **Noise (%)**: proportion of malformed / extraneous tokens or keys.
- **ValidJSON (%)**: percent of generations that are syntactically valid JSON.
- **NoOutput (%)**: percent of samples where the model failed to produce a structured answer at all.

Table 2 shows that ThinkJSON-Base (1.5B) is already competitive with and in several cases better than much larger baselines, despite using only $\sim$20 hours of GRPO on $8\times$H100 plus $\sim$3h SFT on one A100.

Observation: the synthetic regime isolates "can you follow the schema when life is easy." Under that condition, ThinkJSON-Base shows that multi-reward GRPO + light SFT is enough to beat brute parameter count. You do not need 70B+ parameters to get near-perfect structural discipline when the input is not adversarial.

5.2 Realistic Noisy Schemas: Raw Quality

The real-schema benchmark reflects how batch records and QA notes look in practice. Problems include: nested optional blocks, quirky naming, escaped characters, and implicit semantics ("CIP line3 sterile" instead of a clean key like `"sterility_CIP3":"pass"`).

We first report naive "quality looking at content only," which is what most LLM structured-output papers show:

- **Match (%)**: fraction of fields correctly populated.
- **Noise (%)**: malformed / extraneous tokens, schema-violating keys.

These are in Table 3. Every strong model looks basically the same, which is exactly the trap. You could stare at these values and convince yourself that Gemini 2.0 Flash, DeepSeek-R1, and ThinkJSON-Pro are indistinguishable.

The problem with Table 3 is that it ignores whether the JSON is even valid. In production, invalid JSON is literal garbage. You cannot ETL it, you cannot diff it, you cannot store it without manual repair. Treating InvalidJSON as "just another small error" is fantasy.

5.3 Parse-Aware View: What You Can Actually Ship

To correct that, we introduce parse-aware metrics:

Parse Success (%): percent of generations whose `<answer>` block passes `json.loads()` with no exceptions.

Adjusted Match (%): Match $\times$ ParseSuccess. If the model produces semantically correct values but half of them are in invalid JSON, its Adjusted Match tanks.

Adjusted Noise (%): Noise $\times$ ParseSuccess $+100\% \times (1 - \text{ParseSuccess})$. Here invalid JSON counts as 100% noise. This is how downstream systems experience it.

We exclude XGrammar/JSONFormer from this table due to frequent timeouts on the longest schemas; when they did complete, outputs parsed but latency was out-of-band for ETL.

Table 4 shows these. This is where the gap blows wide open.

Model	Match (%)	Noise (%)	ValidJSON (%)	NoOutput (%)
ThinkJSON-Base (ThinkJSON-Base, 1.5B)	∼62.4	∼0.27	high (well-formed blocks)	low
DeepSeek-R1 (671B) (DeepSeek-AI, 2025)	∼41.4	∼11.1	high	moderate
Gemini 2.0 Flash (70B) (Google AI Team, 2025)	∼42.9	∼10.9	mid	mid
Distilled DeepSeek-style Qwen-7B	∼30−40	∼25+	mid	higher
Distilled DeepSeek-style Qwen-1.5B	∼25−35	∼30+	lower	higher

Table 2: Synthetic benchmark (clean paired data, ∼6.5K samples). **Match**: proportion of correctly populated fields. **Noise**: proportion of malformed / extraneous tokens, spurious keys, or text outside the schema. **ValidJSON**: percent of outputs that *parse* as JSON. **NoOutput**: model returns nothing structured.
ThinkJSON-Base (1.5B) gets the best Match (∼62.4%) and the lowest Noise (∼0.27%), outperforming all larger baselines here. DeepSeek-R1 and Gemini 2.0 Flash produce JSON more often than distilled Qwen baselines, but with lower Match and far higher Noise. This matters for regulated extraction because a 1.5B model that you can host locally behaves better than models two orders of magnitude larger.

Model	Match (%)	Noise (%)	Compute / Deployment Notes
DeepSeek-R1 (671B) (DeepSeek-AI, 2025)	96.9	4.0	Frontier-scale RL model (671B). Runs on very large clusters.
Gemini 2.0 Flash (70B) (Google AI Team, 2025)	96.9	4.1	Constrained decoding / structured-output mode. Vendor-served.
ThinkJSON-Pro (ThinkJSON-Pro, 7B LoRA)	95.6	5.3	7B Qwen-2.5 with LoRA adapters. GRPO w/ judge reward. Single A100. On-prem viable.
ThinkJSON-Pro merged (7B variant)	95.0	5.5	Merged checkpoint variant of ThinkJSON-Pro. Single A100. Slightly higher drift.
Qwen2.5-7B instruct (Team, 2024)	94.6	6.5	Vanilla instruction-tuned Qwen2.5-7B. No schema-aware RL. Plain SFT only.

Table 3: Noisy real-schema setting, **without** enforcing parsability. All advanced models cluster around ∼95–97% Match and ∼4–7% Noise. If you only look at these two columns you would think the structured-output problem is solved. You would also (wrongly) conclude that Gemini 2.0 Flash and DeepSeek-R1 behave about the same. That illusion disappears as soon as you ask "does the JSON actually parse."

Model	Parse Success (%)	Adjusted Match (%)	Adjusted Noise (%)
DeepSeek-R1 (671B) (DeepSeek-AI, 2025)	∼99.9	∼96.8	∼4.1
Deployment Reality: Frontier-scale RL, 671B parameters; heavy compute budget.			
ThinkJSON-Pro (ThinkJSON-Pro, 7B LoRA)	∼81.9	∼67.0	∼18.5
Deployment Reality: Runs on a single A100; private, near-real-time, on-prem viable.			
ThinkJSON-Pro merged (7B variant)	∼75.0	∼75.0	∼25.0
Deployment Reality: Merged checkpoint variant of ThinkJSON-Pro; simpler but trades off some stability.			
Qwen2.5-7B instruct (Team, 2024)	∼67.1	∼63.7	∼31.3
Deployment Reality: Plain SFT baseline; no structure-aware RL; frequent key drift and JSON errors.			
Gemini 2.0 Flash (70B) (Google AI Team, 2025; Dong et al., 2024)	∼20.8	∼20.0	∼80.1
Deployment Reality: Constrained decoding fails on deep nested schemas; unusable for automated ETL.			

Table 4: Parse-aware results on noisy real schemas. Higher values are better for Parse Success and Adjusted Match; lower is better for Adjusted Noise.

DeepSeek-R1 (DeepSeek-AI, 2025) leads with ∼99.9% Parse Success and ∼96.8% Adjusted Match. ThinkJSON-Pro (7B LoRA) reaches ∼81.9% / ∼67% on a single A100, enabling on-prem use. Gemini 2.0 Flash (Google AI Team, 2025; Dong et al., 2024) drops to ∼20.8% Parse Success on deep schemas, pushing Adjusted Noise to ∼80%. Qwen2.5-7B Instruct (Team, 2024) lacks multi-reward GRPO and breaks JSON frequently.

5.4 Interpretation and Deployment Consequences

Three conclusions follow from Tables 2, 3, and 4.

1. Small models can learn discipline. On synthetic, controlled input, ThinkJSON-Base (1.5B) beats larger baselines on both Match and Noise, and it does it after ∼23 GPU-hours total. This says the behavior is learnable with multi-reward GRPO and SFT. You do not need to rely on prompt engineering plus regex.

2. Parse Success is the kill shot. In realistic data, everyone claims ∼95%+ Match, but only DeepSeek-R1 and ThinkJSON-Pro produce JSON that actually parses most of the time. Gemini 2.0 Flash looks "good" on content, then faceplants on structure once schemas get deeply nested. This is why we define Adjusted Match and Adjusted Noise. If the JSON does not parse, it is 100% noise in production. That is what ETL systems experience, so that is what we report.

3. On-prem viability matters. DeepSeek-R1 is excellent but frontier-scale and not generally something you self-host in a regulated plant. ThinkJSON-Pro (7B with LoRA) is the only sub-10B model with a parse-success rate above 80% and an Adjusted Match in the usable zone (roughly two-thirds of DeepSeek-R1) while running on a single A100. That is the difference between "cool paper result" and "we can actually deploy this in QA this quarter."

4. Hybrid fallback is obvious. ThinkJSON-Pro is not perfect. ∼18% of outputs still break parse. In practice you catch those failures cheaply: run a `json.loads()` gate. If parse fails, route *just that record* through a slower constrained-decoding engine (e.g., XGrammar) (Dong et al., 2024; Willard and Louf, 2023). Given their speed profile in our setup, XGrammar/JSONFormer are suitable as a fallback only, not as the primary path.

That gives you low latency for the majority of records plus a safety net. This hybrid is cheaper and more auditable than "let Gemini guess and hope."

6 Discussion and Limitations

Parsability is the gate. If `json.loads()` fails, the output is 100% noise to downstream systems. Reporting only "Match" and "Noise" hides that. Our adjusted metrics expose this and should become standard in structured-output work (Liu et al., 2024b; Geng et al., 2025).

Reward shaping vs. brute-force scale. ThinkJSON-Base (1.5B, ∼23 GPU-hours total including SFT) rivals or beats huge baselines on controlled data. This supports the claim that targeted, verifiable rewards plus GRPO can stand in for brute-force scale on schema-bound tasks (DeepSeek-AI, 2025; Dann et al., 2023; Shao et al., 2024).

LoRA is the realistic path. ThinkJSON-Pro shows ∼81.9% parse success with usable adjusted match on noisy real schemas, training and serving on a single A100. That is concretely useful to QA/compliance teams who cannot ship internal

batch records to a vendor LLM (Erdmann et al., 2022; Aher et al., 2025).

Still below 90% parse success. An ∼18% fallout rate is still expensive for ETL. A straightforward next step is a hybrid: run ThinkJSON first, then fall back to a slower constrained-decoding engine (e.g., XGrammar) only on failures (Dong et al., 2024; Willard and Louf, 2023). That keeps latency low on the happy path while patching worst cases.

Judge-LLM drift. The semantic judge adds signal, but also risk. Reward hacking is possible, and drift can silently change what "good enough" means. Long-term, you want a versioned judge and audit trails (Wang et al., 2025; Li et al., 2024).

Synthetic → real gap. Synthetic narratives cannot fully copy the chaos of scanned batch records, multilingual fragments, or handwritten tables. ThinkJSON-Pro narrows that gap, but systematic robustness to OCR trash and multilingual operator jargon is still open.

Chain-of-thought leakage. We force a `<think>` block for auditability. Regulators like this (Erdmann et al., 2022), but some orgs consider it sensitive IP. Future work: redactable or hashed rationales that still satisfy "show me why you set this field" without dumping all internal reasoning.

7 Conclusion and Ethical Considerations

We introduced ThinkJSON, a multi-reward GRPO pipeline that trains mid-size open LLMs to emit parseable, schema-faithful JSON plus an auditable rationale. The method: (1) encodes strict output structure as a rewardable behavior, (2) works under realistic compute budgets (1.5B and 7B), (3) reaches ∼81.9% parse success with LoRA on a single A100 for noisy schemas, and (4) materially closes the distance to a 671B frontier RL model in regulated data extraction tasks (DeepSeek-AI, 2025).

This matters because regulated manufacturing needs structured records, traceability, and on-prem deployment (Labant, 2025; Erdmann et al., 2022; Aher et al., 2025). Pure prompt engineering is not reliable enough (Liu et al., 2024b), and pure constrained decoding is often too slow or too brittle at inference (Willard and Louf, 2023; Dong et al., 2024; Google AI Team, 2025).In our setting with long, nested schemas, XGrammar and JSONFormer were noticeably slower and often failed to complete for large unstructured inputs within our inference budget. Reinforcement learning with verifiable schema rewards is the middle path: you turn schema adherence into model behavior, not a bolt-on hack.

Ethics / compliance. We do not claim automatic signoff for regulated release. The generated JSON and the `<think>` trace are review artifacts, not the final batch record. Human QA must still confirm high-risk fields. Also, the judge-LLM reward can encode bias; any production deployment must version and audit the judge. Finally, because ThinkJSON-Pro trains and runs locally, it supports data sovereignty and chain-of-custody requirements in GxP-style environments (Erdmann et al., 2022; Aher et al., 2025).

References

Json schema store. https://www.schemastore.org/json/. Accessed 2025-10-30.

V. Aher and 1 others. 2025. An overview on pharmaceutical regulatory affairs using artificial intelligence. *IJPS Journal*.

Anonymous. 2025. Schema reinforcement learning dataset. GitHub repository.

C. Dann and 1 others. 2023. Reinforcement learning can be more efficient with multiple rewards. In *Proceedings of ICML*.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Y. Dong and 1 others. 2024. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*.

N. Erdmann and 1 others. 2022. Ai maturity model for gxp application: A foundation for ai validation. *Pharmaceutical Engineering (ISPE)*.

S. Geng and 1 others. 2025. Generating structured outputs from language models: Benchmark and studies. *arXiv preprint arXiv:2501.10868*.

Google AI Team. 2025. Generate structured output with the gemini api. Technical Documentation.

M. Labant. 2025. Smart biomanufacturing: From piecemeal to all of a piece. *GEN Biotechnology*.

D. Li and 1 others. 2024. Large language model-driven structured output: A comprehensive benchmark and spatial data generation framework. *ISPRS International Journal of Geo-Information*.

M. Liu and 1 others. 2024a. User-centered constraints on large language model output. *arXiv preprint arXiv:2406.xxxxx*.

M. Liu and 1 others. 2024b. "We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. *arXiv preprint arXiv:2406.xxxxx*.

A. Madaan and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Z. Shao and 1 others. 2024. Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Qwen Team. 2024. Qwen2.5: A party of foundation models. Technical Report.

P. Wang and 1 others. 2023. Math-shepherd: A label-free step-by-step verifier for llms in mathematical reasoning. *arXiv preprint arXiv:2312.08935*.

Z. Wang and 1 others. 2025. Verifiable format control for large language model generations. *arXiv preprint arXiv:2502.04498*.

B.T. Willard and R. Louf. 2023. Efficient guided generation for large language models. In *Proceedings of NeurIPS Workshop*.

Appendix

.1 A.1 JSON Schema Construction

```
You are an expert in building a
    hierarchical JSON schema and object
    for the domain {DOMAIN}.
Your task is to create:

1. A multi-level JSON Schema describing:
   - ROOT (level 0)
   - SECTION (level 1)
   - SUBSECTION (level 2)
   - DETAIL_N (level 3+)
   Each level may contain tables (2D
    layouts) and checkbox elements (MCQs,
     confirmations),
   with nested components reflecting
    complex structures.

2. A JSON Object matching this schema
    exactly, including:
   - "id", "title", "level", and "level_
    type"
   - Array of "component" objects (
    paragraphs, tables, checkboxes)
   - Recursive "children" arrays
   - Special "properties" (variables,
    content) for data, logs, metrics, or
     formulas

Formatting:
- Escape quotes as (\")
- Replace newlines with \\n
- No trailing commas or extra data
```

.2 A.2 Text Generation from JSON Objects

```
You are an expert in generating
    hierarchical text documents from
    JSON Object data points.

Task: Convert the JSON Object into a
    narrative-style text document.

Input: {Domain, JSON Schema, JSON Object
    }
Output: <text> ... </text>

Rules:
1. Map JSON levels and components to
    text layout styles.
2. Obscure structure by embedding data
    in natural sentences.
3. Include all data fields (titles,
    variables, metadata).
4. Add filler paragraphs with unrelated
    context or definitions.
5. End each paragraph naturally, not
    abruptly.
```

.3 A.3 Schema-based Extraction Evaluation

```
You are an AI assistant extracting
    structured data from unstructured
    text.

Inputs:
1. Source text
2. Blank schema
3. Filled schema

Goals:
1. Compare Text + Blank Schema  Filled
    Schema
2. Explain the population process step
    by step
3. Output reasoning only (Chain-of-
    Thought)
4. Verify that reasoning exactly
    reconstructs the Filled Schema

Output Format:
Chain of Thought Explanation: """
```

.4 A.4 Hierarchical Data Mapping

```
Role: Expert data extractor mapping
    hierarchical text to a JSON schema.

Input: Text + Blank JSON Schema
Output: <think> (Reasoning) and <answer>
     (Filled JSON)

Rules:
1. Always produce both <think> and <
    answer>.
2. Preserve hierarchy (ROOT  SECTION
    SUBSECTION  DETAIL_N)
3. Escape quotes, replace newlines with
    \\n, use double quotes only.
4. State reasoning briefly: "Direct
    mapping from text to schema."
```