Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Self-Healing JSON Generation in Small Language Models: A Systematic Evaluation of Syntax Repair, Schema Coercion, and Validator Feedback

Ahmad Abdelrahman

ahmedissa.it@hotmail.com

independet researcher https://orcid.org/0009-0000-5251-5274

Research Article

Keywords: Small language models, JSON generation, structured information extraction, syntactic repair, JSON Schema validation, schema-guided coercion, validator-guided feedback, self-healing pipelines

Posted Date: November 9th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11641294/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Self-Healing JSON Generation in Small Language Models: A Systematic Evaluation of Syntax Repair, Schema Coercion, and Validator Feedback

Ahmed Yousef Mohmmad Abdelrahman

Abstract

Small language models offer an affordable way to extract structured information, including on devices with limited computing resources. However, their responses often contain JSON that cannot be parsed or does not satisfy the required schema. We investigate a self-healing approach that operates on generated text alone, without changing or accessing the model's decoder. The approach combines three steps: (L1) a tolerant lexer and bracket-stack parser for syntax repair, (L2) schema-guided conversions that preserve the original meaning, and (L3) a retry mechanism that provides JSON Schema validation errors to the model. We test these steps on 120 seeded text-to-JSON tasks spanning four schema families, using three instruction-tuned models ranging from 360M to 1.5B parameters and six experimental conditions. On average, only 16.7% of raw responses satisfy the schema. Applying L1 and L2 raises validity to 67.8% without another model call. The complete pipeline achieves 78.1% validity with an average of 1.57 calls per task.

The results reveal several important distinctions. Qwen2.5 frequently produces correct JSON inside Markdown code fences, making many failures easy to correct without involving the model again. SmolLM2-360M, by contrast, generally avoids code fences but often misunderstands the requested structure or copies the schema itself; validator feedback is more useful in these cases. Feedback on its own performs poorly (21.1% validity), largely because it does not break persistent formatting habits. Compared with blind resampling at a similar call budget, feedback improves results for the weakest model ($p = 0.003$), although the overall difference is not statistically significant ($p = 0.060$). More importantly, producing valid JSON does not guarantee accurate information: 45.6% of valid self-healed responses still contain at least one incorrect field, or 38.4% under a more forgiving comparison. We observe empty strings, fabricated phone numbers, and default enum choices among these errors.

Keywords: Small language models; JSON generation; structured information extraction; syntactic repair; JSON Schema validation; schema-guided coercion; validator-guided feedback; self-healing pipelines.

1 Introduction

Turning free-form text into structured records is a common use of language models. Applications include processing registration forms, extracting invoice details, organizing calendar entries, collecting bibliographic metadata, and preparing arguments for tool calls. In each case, another program needs to interpret the result, usually by checking it against a JSON Schema. Smaller models are appealing for this work because they can reduce costs and latency and can sometimes run locally, which also helps with privacy [1, 2]. Yet these models are often less dependable when asked to return JSON that is both correctly formatted and compliant with a specific schema.

One established solution is constrained decoding. By restricting which tokens can be generated at each step, it helps the model remain within a grammar or schema [3, 4, 5, 6, 7]. Although effective at enforcing structure, this approach comes with practical limitations:

- it needs access to the logits inside the serving loop, which many hosted APIs and embedded runtimes do not expose;
- support for JSON Schema features varies considerably between engines [8];
- forcing a format can distort the model's distribution [9] and reduce task performance [10].

A more accessible alternative is to let the model generate its answer normally, repair formatting problems afterward, validate the result, and provide the validation errors when a retry is necessary [11, 12]. This workflow is widely used in practice, but the contribution of each step has received relatively little controlled evaluation for small models.

This paper examines those steps separately. We describe a three-layer repair pipeline in Section 3 and evaluate it against records with known ground truth in Section 4. Six experimental conditions let us measure the effect of each layer, while a blind-retry baseline tests whether validator feedback offers any benefit beyond simply generating another answer. Specifically, we make four contributions:

1. A precise, model-agnostic **self-healing pipeline** with three layers: a lenient syntactic repairer (L1), a schema-guided coercer that never invents content (L2), and validator-guided re-prompting (L3).
2. A **layer-by-layer ablation** on three small open models. It shows that the deterministic layers deliver most of the gain at zero model cost: 83.3% of all repaired outputs needed no additional call.
3. A **failure taxonomy**, derived automatically from the repairs, which reveals two distinct failure regimes. One is a formatting habit (code fences) that is trivial to repair. The other is a lack of schema understanding (copying the schema, wrong types, missing fields) that is hard to repair.
4. Evidence that **validity is not correctness**. Validator feedback lets models produce schema-valid but wrong values, such as empty strings, placeholder phone numbers, and a default enum value. We therefore argue that structured-output methods should be evaluated by field-level accuracy, not by validity alone.

2 Related Work

Constrained and guided decoding. Grammar-constrained generation compiles a grammar or a JSON Schema into a token-level mask. Examples are the finite-state approach of Outlines [3], the query language LMQL [4], the structured-generation runtimes SGLang [7] and XGrammar [5], GBNF grammars in llama.cpp [13], and, earlier, incremental parsing for text-to-SQL [6]. Jsonformer [14] fills only the value slots of a fixed JSON skeleton. JSONSchemaBench [8] compares such engines on about 10K real-world schemas and finds large differences in coverage and efficiency. Park et al. [9] show that naive constrained decoding distorts the model's distribution and propose a correction. Our pipeline is orthogonal to this line of work. It requires only the generated text, so it applies where logits are not available, and its L2 and L3 layers can also be stacked on constrained decoding to address *semantic* errors that a grammar cannot exclude.

Format following and structured-output evaluation. IFEval [15] evaluates verifiable formatting instructions, StructuredRAG [16] measures how reliably models follow JSON response formats, and Tam et al. [10] report that strict format restrictions can reduce reasoning performance. These works mostly score whether the format is respected. We additionally score whether the content is correct, and we show that the two can diverge substantially once repair is applied.

Self-correction. Self-Refine [17], Reflexion [18], and Self-Debugging [19] improve outputs through iterative feedback. Huang et al. [20] show that *intrinsic* self-correction, without an external signal, is unreliable. Our L3 layer is a minimal form of *extrinsic* feedback: the validator's error list. The blind-retry control separates the value of this information from the value of simply sampling again. Retrying with validation errors is common practice in libraries such as Instructor [12], and lenient JSON repair is available in tools such as json_repair [11]. We contribute a controlled measurement of what each step achieves for small models.

3 Method

Consider an input passage x, a JSON Schema S, and a model response $y = LM(p(x, S))$, where p is the extraction prompt. A downstream program can use the result only when it can be parsed into an object o that satisfies S. Our aim is not merely to produce acceptable JSON, but to obtain an object that accurately reflects x while minimizing additional model calls. To do so, we apply three layers in order of increasing cost (Algorithm 1).

3.1 L1: Lenient syntactic repair

The first layer removes surrounding Markdown fences and any introductory text before the first { or [. It then tokenizes the remaining text with a lenient lexer designed to recognize several common departures from strict JSON:

- double, single, and typographic quotes;
- `//`, `#`, and `/* */` comments;
- Python and JavaScript literals (`True`, `None`, `undefined`, `NaN`);
- bare words, which become keys when followed by a colon and string values otherwise.

Unescaped quotation marks inside a string require special handling. For example, in the phrase "The "Best" Paper", the lexer treats a quote as the end of the string only when the following non-space character indicates a structural boundary, a newline, or the end of the text. The assembler then rebuilds the output using a stack of brackets:

- it inserts missing commas between adjacent values and drops trailing or duplicated commas;
- it supplies `null` for a key that is immediately followed by a closing bracket, and corrects mismatched brackets;
- it stops at the end of the first top-level value, ignoring trailing chatter;
- on truncated input, it removes a dangling key or separator and closes all open containers.

Each modification is recorded with a named repair flag. These flags allow us to identify the types of syntax errors present in the model outputs, as discussed in Section 5.2. Appendix B provides the unit cases used to test L1.

3.2 L2: Schema-guided coercion

Once the text can be parsed, L2 examines its values against the target schema. It makes only conversions intended to preserve the information already present, including:

- numeric strings to numbers, including currency symbols, thousands separators, and trailing units (`"$1,204.50"` → `1204.5`); integral floats to integers;
- `"yes"`/`"no"`, `"true"`/`"false"`, and 0/1 to booleans;
- case-insensitive matching of enum values, and renaming keys under case and separator normalisation (`orderId` → `order_id`);
- dropping properties forbidden by `additionalProperties: false`;
- wrapping a scalar into an array, splitting `"A, B and C"` for string arrays, and unwrapping a one-element list where an object is expected;
- rewriting a string into a required pattern when an unambiguous parse exists, for calendar dates and 12/24-hour times (`"October 8, 2025"` → `"2025-10-08"`, `"4:45 PM"` → `"16:45"`).

L2 is deliberately conservative: it does not create missing information. Its sole exception is adding an explicit null when the schema requires a nullable property that was omitted.

3.3 L3: Validator-guided re-prompting

If the result still fails validation, L3 asks the model to correct it. The next prompt includes the previous response (after L1 and L2 when available) and up to six validator errors, written as a field path followed by its error message. Each new answer passes through L1 and L2 again. The process stops when a valid result is obtained or the retry budget K is exhausted.

Algorithm 1. Self-healing JSON generation

```
Require: text x, schema S, model LM, retry budget K
 1  C ← p(x, S);  y ← LM(C)
 2  for r = 0, 1, …, K:
 3      o ← StrictParse(y) or else L1(y)            # syntactic repair
 4      if o ≠ ⊥:  o ← L2(o, S)                      # schema coercion
 5      E ← Validate(o, S)
 6      if E = ∅:  return o
 7      if r = K:  return ⊥
 8      C ← C ‖ ⟨assistant: o (or y)⟩ ‖ ⟨user: Render(E)⟩
 9      y ← LM(C)                                    # L3
```

4 Experimental Setup

Benchmark. We built a benchmark of 120 extraction tasks using a fixed random seed. The tasks are evenly distributed across four schema families with different structural requirements (Table 1). For each task, we generate a ground-truth record and express its contents through a natural-language template. This gives us a known correct answer for every field, allowing us to evaluate factual accuracy as well as schema validity. The input wording intentionally differs from the representation required by the schema: dates may be written in words, times may use a 12-hour clock, prices may contain currency symbols, and some fields are implied rather than stated directly. Nullable values can also be absent from the source. Each prompt includes the full schema and asks the model to return only the JSON object (Appendix A).

Family	Structure	What it tests
contact	flat, 5 fields	nullable field, email pattern, integer
order	nested object, array of objects	enum, ID patterns, currency-formatted numbers
event	flat with string array	date/time patterns, implied boolean, nullable location
paper	array of objects, string array	author objects {first, last}, integer range, boolean

Table 1. Schema families (30 tasks each).

Models and decoding. We evaluate SmolLM2-360M-Instruct [2], Qwen2.5-0.5B-Instruct, and Qwen2.5-1.5B-Instruct [1]. All three run through Hugging Face Transformers [21] in half precision on a single NVIDIA T4 GPU in Google Colab, using their respective chat templates. Generation uses temperature 0.7, top-p 0.95, a maximum of 384 new tokens, and batches of 32; other settings follow each model's generation configuration. We cache the first response to each task, ensuring that all six conditions start from exactly the same answer for a given model and input.

Conditions. We test six conditions, with a maximum of K = 2 additional model calls for the retry-based approaches.

1. **Raw**: strict `json.loads`.
2. **+L1**: syntactic repair.
3. **+L2**: syntactic repair and schema coercion; no extra model calls.
4. **L3 feedback only**: validator feedback with strict parsing, without L1 or L2.
5. **L1+L2 + blind retry**: resample the *original* prompt while the output is invalid.
6. **Self-healing**: L1+L2+L3.

Metrics. We report six evaluation measures. Proportions include 95% Wilson confidence intervals, and comparisons between paired conditions use the exact McNemar test.

- *Parse*: a JSON value was obtained.
- *Valid*: the value conforms to *S*.

- *Field accuracy*: the share of ground-truth leaf values reproduced exactly. The comparison is type-strict, case- and whitespace-insensitive for strings, and order-insensitive for string lists.
- *Usable accuracy*: field accuracy for valid outputs and 0 otherwise, which is what a consumer that rejects invalid output actually receives.
- *Exact*: valid, with every field correct.
- *Cost*: model calls and generated tokens per task.

5 Results

5.1 Overview

Table 2 and Figure 1 provide an overview of the results. Without repair, only 45.3% of responses can be parsed as JSON, and just 16.7% satisfy the schema. Raw validity varies substantially by model, from 3.3% for Qwen2.5-0.5B to 37.5% for Qwen2.5-1.5B.

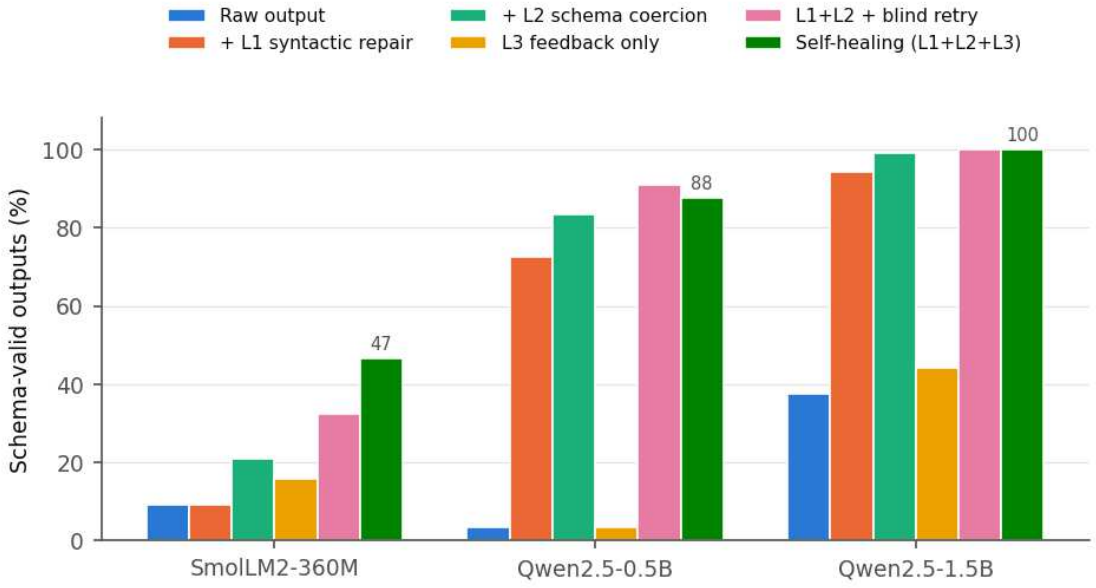


Figure 1. Schema-valid outputs per model and condition. Numbers above the bars give the full self-healing pipeline.

The first two layers account for most of the improvement. L1 alone increases average validity to 58.6%, and adding L2 brings it to 67.8%, with no extra model calls. Over the same progression, usable accuracy rises from 15.8% to 63.2%. With feedback-based retries, the complete system achieves 78.1% validity and 71.3% usable accuracy, requiring 1.57 model calls per task on average. Among initially invalid outputs that the pipeline successfully repairs, 83.3% are fixed by L1 and L2 without another call.

These gains are not evenly distributed across models. For Qwen2.5-1.5B, L1 and L2 already achieve 99.2% validity, leaving little room for retries. For Qwen2.5-0.5B, much of the improvement comes from removing formatting wrappers: L1 raises validity from 3.3% to 72.5%. SmolLM2-360M follows a different pattern. L1 makes no difference, but feedback raises validity from 20.8% to 46.7% (95% CI [38.0, 55.6]). Even then, only 10.0% of its outputs match every field exactly.

Model	Condition	Parse	Valid	Usable acc.	Exact	Calls	Tokens
SmolLM2-360M	Raw output	93.3	9.2	8.3	1.7	1.00	145
	+ L1 syntactic repair	98.3	9.2	8.3	1.7	1.00	145
	+ L2 schema coercion	98.3	20.8	18.8	5.8	1.00	145
	L3 feedback only	90.8	15.8	13.6	3.3	2.77	347
	L1+L2 + blind retry	98.3	32.5	28.2	7.5	2.51	339
	Self-healing (L1+L2+L3)	97.5	**46.7**	39.5	10.0	2.41	259
Qwen2.5-0.5B	Raw output	3.3	3.3	2.5	0.0	1.00	97
	+ L1 syntactic repair	100.0	72.5	64.2	28.3	1.00	97
	+ L2 schema coercion	100.0	83.3	73.9	32.5	1.00	97
	L3 feedback only	3.3	3.3	2.5	0.0	2.93	289
	L1+L2 + blind retry	100.0	**90.8**	79.9	35.0	1.27	117
	Self-healing (L1+L2+L3)	100.0	87.5	76.8	32.5	1.31	119
Qwen2.5-1.5B	Raw output	39.2	37.5	36.7	31.7	1.00	101
	+ L1 syntactic repair	100.0	94.2	92.0	81.7	1.00	101
	+ L2 schema coercion	100.0	99.2	96.8	85.0	1.00	101
	L3 feedback only	44.2	44.2	43.2	35.8	2.22	230
	L1+L2 + blind retry	100.0	**100.0**	97.5	85.0	1.01	101
	Self-healing (L1+L2+L3)	100.0	**100.0**	97.5	85.0	1.01	101
Average	Raw output	45.3	16.7	15.8	11.1	1.00	114
	+ L1 syntactic repair	99.4	58.6	54.8	37.2	1.00	114
	+ L2 schema coercion	99.4	67.8	63.2	41.1	1.00	114
	L3 feedback only	46.1	21.1	19.8	13.1	2.64	289
	L1+L2 + blind retry	99.4	74.4	68.5	42.5	1.60	186
	Self-healing (L1+L2+L3)	99.2	**78.1**	71.3	42.5	1.57	160

Table 2. Main results on 120 tasks per model (percentages, except calls and generated tokens per task). Usable acc. counts field accuracy only for schema-valid outputs. Best validity per model in bold.

5.2 Two failure regimes

The initial errors fall into two quite different patterns, as Figure 2 illustrates.

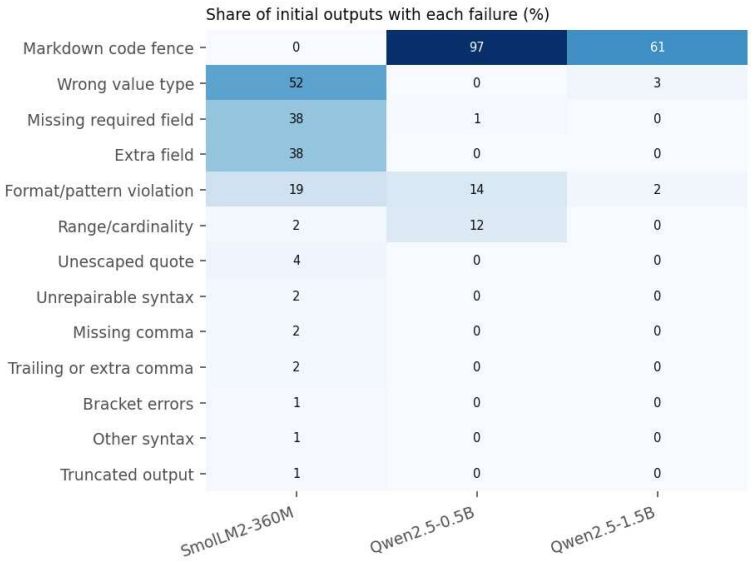


Figure 2. Failure taxonomy of the initial outputs: the share of outputs exhibiting each failure. An output can show several failures. Syntactic categories come from L1 repair flags; schema categories come from validator errors after L1.

Formatting habit (Qwen2.5). 96.7% of Qwen2.5-0.5B and 60.8% of Qwen2.5-1.5B answers wrap the JSON in a triple-backtick `json` fence, even though the prompt asks for the object only. The JSON inside is almost always syntactically perfect: no Qwen output needed comma, quote, or bracket repair. A strict parser rejects these answers, while L1 accepts all of them. This single habit explains most of the gap between Qwen's raw and repaired validity. The remaining Qwen errors are mostly format violations (14.2% of 0.5B outputs keep `"October 8, 2025"` or `"4:45 PM"`), which L2 normalises (Figure 4a). Another 12.5% of 0.5B outputs contain an empty `authors` list, which no deterministic rule can fill.

Missing schema understanding (SmolLM2-360M). SmolLM2-360M never emits a code fence, and 93.3% of its raw answers parse. Only 9.2% are valid, however. Its failures are semantic:

- wrong value types in 52.5% of outputs;
- missing required fields in 38.3%;
- extra fields in 38.3%.

A closer look at the extra fields shows that 40 of the 120 SmolLM2 responses (33.3%) reproduce the schema itself, including keys such as type, properties, required, and additionalProperties. Some responses copy it directly; others place extracted values inside the properties object. L2 can remove forbidden keys, but it cannot reliably recover information that the model placed in the wrong part of the structure. This limits the improvement available from deterministic conversion alone.

Traditional syntax problems—unescaped quotation marks, missing or extra commas, mismatched brackets, and truncated output—occur, but each affects fewer than 5% of SmolLM2 responses. Across the models in this experiment, code-fence wrappers and schema-related mistakes are far more common than the syntax damage usually targeted by JSON repair tools.

5.3 Why feedback alone fails

Validator feedback by itself is surprisingly ineffective. It achieves only 21.1% validity while using 2.64 calls per task, compared with 67.8% validity from L1 and L2 using a single call. For Qwen2.5-0.5B, feedback alone leaves validity unchanged at 3.3%, even after as many as two retries. The explanation is straightforward: all 116 retried responses still end inside Markdown code fences. The model corrects or repeats the JSON but keeps presenting it in the same wrapper. Removing that wrapper in code is simple; asking the model to abandon the habit is not. In this setting, deterministic repair makes feedback useful rather than competing with it.

5.4 Is it the feedback or the second sample?

A retry offers two possible benefits: the model gets another chance to generate an answer, and it may learn from the specific error message. To separate these effects, we compare validator-guided retries with blind retries that resample the original prompt under the same two-call budget. Across all models, feedback reaches 78.1% validity versus 74.4% for blind resampling. Both approaches use nearly the same number of calls (1.57 and 1.60, respectively), but feedback produces fewer generated tokens per task (160 versus 186; Figure 3).

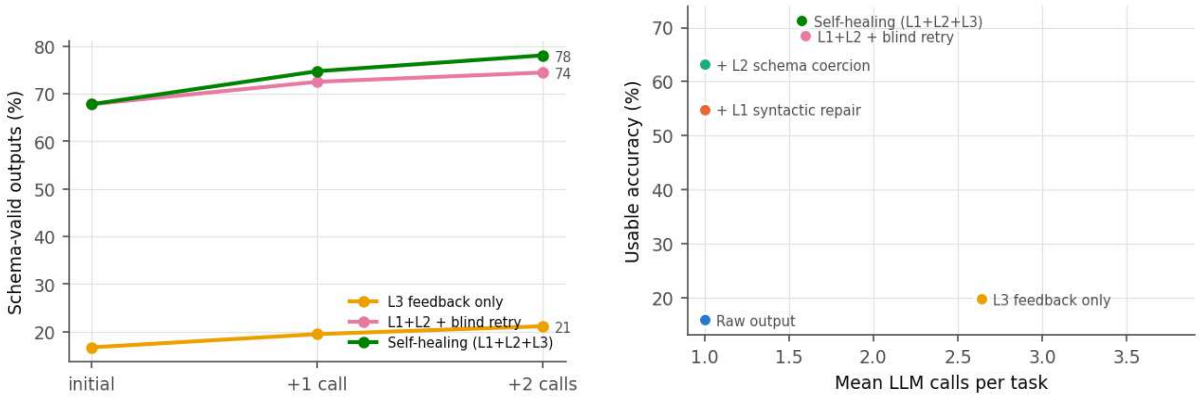


Figure 3. Left: cumulative schema validity after each additional call for the three retry strategies. Right: usable accuracy against mean model calls per task. Both averaged over models.

The paired comparison is mixed:

- **SmolLM2-360M:** feedback clearly helps (46.7% vs. 32.5%). It rescued 24 tasks that blind retry did not, against 7 in the other direction (exact McNemar $p = 0.003$).
- **Qwen2.5-0.5B:** the direction reverses (87.5% vs. 90.8%; 3 vs. 7 tasks, $p = 0.34$).
- **Qwen2.5-1.5B:** both conditions reach 100%.

The combined results favor feedback on 27 tasks and blind retry on 14, but the difference does not meet the conventional 5% significance threshold ($p = 0.060$). We interpret this as evidence that feedback is especially valuable when an error can be explained and the model can act on the explanation—for example, when a required phone field is missing. When failures are infrequent and largely random, another sample may work just as well. Our results therefore do not support a general claim that feedback is better than resampling for every small model.

5.5 Validity is not correctness

Passing schema validation does not necessarily mean that the extracted information is right. Of the responses accepted after self-healing, 45.6% have at least one incorrect field. The error rate is 15.0% for Qwen2.5-1.5B, 62.9% for Qwen2.5-0.5B, and 78.6% for SmolLM2-360M. This helps explain why overall exact-match accuracy is only 42.5%, despite a validity rate of 78.1%.

Some apparent mistakes may be harmless wording differences. For instance, an output of "Main Auditorium" differs from a ground-truth value of "the Main Auditorium" under strict comparison. We therefore repeat the analysis using a more forgiving rule that ignores leading articles and accepts substring matches. Even then, 38.4% of valid outputs contain an error. For Qwen2.5-1.5B, the error rate falls from 15.0% to 5.8%, but the larger pattern remains. Four types of error appear repeatedly:

- **Empty-string placeholders.** 30 of the 105 valid Qwen2.5-0.5B outputs contain an empty string, e.g. `"name": ""`, `"city": ""`. These satisfy `type: string` while carrying no information. Feedback even *converges* to them: in Figure 4c, the validator flags only the empty email (a pattern violation), the model repairs that field, and the empty name and city pass.

- **Invented values.** SmolLM2-360M fills required fields with plausible placeholders, such as the phone number `"123-456-7890"`, or invents a location for an online event. Feedback turns these into valid outputs (Figure 4b).
- **Default enum value.** In 24 of the 27 valid `order` outputs of SmolLM2-360M the status is `"pending"`, the first enum value in the schema. It is correct in only 11 of them.
- **Boundary and arithmetic errors.** Examples are copying extra context into a field (`"title": "Onboarding Workshop 2025"`, `"venue": "CVPR 2019"`) and converting 12-hour times to 24-hour times incorrectly (`4:45 PM` → `"04:45"`).

These examples highlight a limitation shared by any system optimized primarily for validator acceptance, including constrained decoding: an obvious formatting failure can become a less visible content error. We therefore recommend reporting field-level accuracy alongside schema validity, particularly for small models. Additional lightweight checks—such as rejecting empty required strings and verifying that extracted values are supported by the source text—could catch many of the cases above and provide more informative feedback for L3.

(a) Qwen2.5-0.5B, event: repaired by L1 + L2 with no extra call (correct).

```
initial: ```json {"title": "Onboarding Workshop", "date": "October 8, 2025",
          "start_time": "4:45 PM", "duration_minutes": 90, "is_virtual": true, ...} ```
healed:  {"title": "Onboarding Workshop", "date": "2025-10-08", "start_time": "16:45", ...}
```

(b) SmolLM2-360M, contact: valid after one feedback round, but the phone number is invented.

```
text:    "... Noah can be reached by phone at +1-510-660-6341."
initial: {"name": "Noah Haddad", ..., "phone": ["123-456-7890"], "city": "Cairo"}
healed:  {"name": "Noah Haddad", ..., "phone": "123-456-7890", "city": "Cairo"}
```

(c) Qwen2.5-0.5B, contact: feedback repairs the flagged field only; empty strings pass.

```
text:    "New sign-up from Berlin: Diego Hassan (age 60). Contact email: diego.hassan@acme.io."
initial: ```json {"name": "", "email": "", "age": 60, "phone": null, "city": ""} ```
healed:  {"name": "", "email": "diego.hassan@acme.io", "age": 60, "phone": null, "city": ""}
```

Figure 4. Real outputs from our runs (shortened with "..."). Example (a) is a correct repair; (b) and (c) are schema-valid but wrong.

5.6 Effect of schema structure

The success of self-healing also depends on the kind of record being extracted. As Table 4 in Appendix C shows, validity reaches 96.7% for orders, 82.2% for events, and 81.1% for contacts, but only 52.2% for papers.

Interestingly, orders are the most deeply nested records, yet they are also the easiest to validate after repair. Their structure largely follows how the details appear in the input. Paper records pose a different challenge: the model must turn author names into separate {first, last} objects. SmolLM2-360M produces no valid paper records; 17 of its 30 initial responses represent authors as strings, and eight place them in a dictionary. Qwen2.5-0.5B returns an empty author list in 15 of 30 cases. Splitting a name such as "Noah Rahimi" into {"first": "Noah", "last": "Rahimi" requires a semantic decision, so we do not include it in L2. These results suggest that the mismatch between the wording of the source and the required output structure can matter more than nesting depth alone.

6 Discussion

Practical recommendations. For practical deployments, the results point to a sensible order for handling structured extraction with small models:

1. Always apply deterministic repair and coercion first. It is free, and it removed most failures in our runs.
2. Add feedback-based retries only for models whose remaining errors are semantic. For models whose errors are stochastic, a plain resample is as effective and simpler.
3. Validate content as well as shape: forbid empty strings for required fields, and check that extracted values occur in the source.

4. Prefer schemas that mirror the surface of the source text. Where a structural transformation is unavoidable (such as splitting names), do it in code after extraction.

Relation to constrained decoding. Constrained decoding would address many of the code-fence and syntax issues handled by L1, but it would not resolve all of the problems identified here. A model could still return empty strings, invent plausible values, or select a default enum option that is valid but incorrect. Our findings about content accuracy therefore also matter for systems that enforce JSON structure during generation.

7 Limitations

- **Model coverage.** We report three models from 360M to 1.5B parameters from two families. Larger small models (SmolLM2-1.7B, Qwen2.5-3B) exceeded the memory of a single 16 GB T4 at our batch size and are left for future runs. The model-dependence of our findings means they should not be extrapolated to other families without testing.
- **Benchmark.** The benchmark is synthetic, template-based, English-only, and has 120 tasks. This gives exact ground truth, but it covers neither the long tail of real-world schemas [8] nor noisy inputs such as OCR text.
- **Single configuration.** We use one prompt, one temperature (0.7), and one seed. Greedy decoding, system prompts that explicitly forbid fences, or few-shot examples could change the raw failure profile, notably the fence habit.
- **Strict field matching.** Strict comparison may undercount correct answers. Our lenient re-scoring bounds this effect at about 7 percentage points overall.
- **No constrained-decoding baseline.** The two approaches are complementary, and measuring L2/L3 on top of constrained decoding is a natural next step.

8 Conclusion

Small language models can often be made much more reliable at producing schema-valid JSON through simple post-generation processing, without changing the decoding process. In our experiments, deterministic syntax repair and schema-guided coercion provided most of the improvement without additional model calls. Validator feedback helped most when the model's remaining errors involved schema understanding, but it did little when failures came from persistent formatting habits. The larger lesson is that valid JSON is not always useful JSON. Even accepted outputs contained empty placeholders, fabricated values, and incorrect enum selections. Evaluations of structured extraction should therefore measure the accuracy of individual fields as well as schema compliance, and practical systems should check the extracted content against its source. The notebook, benchmark generator, and generated responses are released to make the reported results reproducible.

References

[1] Qwen Team. Qwen2.5 Technical Report. arXiv:2412.15115, 2024.

[2] L. B. Allal, A. Lozhkov, E. Bakouch, G. M. Blázquez, G. Penedo, L. Tunstall, et al. SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model. arXiv:2502.02737, 2025.

[3] B. T. Willard and R. Louf. Efficient Guided Generation for Large Language Models. arXiv:2307.09702, 2023.

[4] L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting Is Programming: A Query Language for Large Language Models. In Proceedings of the ACM on Programming Languages (PLDI), 2023.

[5] Y. Dong, C. F. Ruan, Y. Cai, R. Lai, Z. Xu, Y. Zhao, and T. Chen. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. arXiv:2411.15100, 2024.

[6] T. Scholak, N. Schucher, and D. Bahdanau. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In Proceedings of EMNLP, 2021.

[7] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In Advances in Neural Information Processing Systems, 2024.

[8] S. Geng, H. Cooper, M. Moskal, S. Jenkins, J. Berman, N. Ranchin, R. West, E. Horvitz, and H. Nori. Generating Structured Outputs from Language Models: Benchmark and Studies. arXiv:2501.10868, 2025.

[9] K. Park, J. Wang, T. Berg-Kirkpatrick, N. Polikarpova, and L. D'Antoni. Grammar-Aligned Decoding. In Advances in Neural Information Processing Systems, 2024.

[10] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H.-y. Lee, and Y.-N. Chen. Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models. In Proceedings of EMNLP: Industry Track, 2024.

[11] S. Baccianella. json_repair: Repair invalid JSON documents. https://github.com/mangiucugna/json_repair, 2023.

[12] J. Liu et al. Instructor: Structured outputs for LLMs. https://github.com/567-labs/instructor, 2023.

[13] G. Gerganov et al. llama.cpp: LLM inference in C/C++ (GBNF grammars). https://github.com/ggml-org/llama.cpp, 2023.

[14] 1rgs. Jsonformer: A Bulletproof Way to Generate Structured JSON from Language Models. https://github.com/1rgs/jsonformer, 2023.

[15] J. Zhou, T. Lu, S. Mishra, S. Brahma, S. Basu, Y. Luan, D. Zhou, and L. Hou. Instruction-Following Evaluation for Large Language Models. arXiv:2311.07911, 2023.

[16] C. Shorten, C. Pierse, T. B. Smith, E. Cardenas, A. Sharma, J. Trengrove, and B. van Luijt. StructuredRAG: JSON Response Formatting with Large Language Models. arXiv:2408.11061, 2024.

[17] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-Refine: Iterative Refinement with Self-Feedback. In Advances in Neural Information Processing Systems, 2023.

[18] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In Advances in Neural Information Processing Systems, 2023.

[19] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching Large Language Models to Self-Debug. In International Conference on Learning Representations, 2024.

[20] J. Huang, X. Chen, S. Mishra, H. S. Zheng, A. W. Yu, X. Song, and D. Zhou. Large Language Models Cannot Self-Correct Reasoning Yet. In International Conference on Learning Representations, 2024.

[21] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, et al. Transformers: State-of-the-Art Natural Language Processing. In Proceedings of EMNLP: System Demonstrations, 2020.

Appendix A Prompts

System: *You are a precise information-extraction assistant.*

User: *Extract the information from the text below into a JSON object that conforms to this JSON Schema:* ⟨schema as indented JSON⟩ *Text:* """⟨text⟩""" *Respond with the JSON object only.*

Repair turn (L3): *Your previous answer is not valid against the JSON Schema. Problems found by the validator:* ⟨up to 6 lines "- path: message"⟩ *Fix these problems using the original text. Respond with the corrected JSON object only.*

Appendix B L1 unit cases

Failure	Model output	After L1
Markdown fence	```json\n{"a": 1}\n```	{"a": 1}
Prose wrapper	Sure! Here is the JSON:\n{"a": 1}\nHope this helps!	{"a": 1}
Trailing commas	{"a": 1, "b": [1, 2,],}	{"a": 1, "b": [1, 2]}
Python-style	{'a': True, 'b': None, 'c': 'O'Brien'}	{"a": true, "b": null, "c": "O'Brien"}
Unquoted tokens	{a: 1, status: shipped}	{"a": 1, "status": "shipped"}
Missing comma	{"a": 1\n "b": 2}	{"a": 1, "b": 2}
Comments	{"a": 1, // first\n "b": 2 /* second */}	{"a": 1, "b": 2}
Unescaped quote	{"t": "The "Best" Paper", "y": 2020}	{"t": "The \"Best\" Paper", "y": 2020}
Truncated (array)	{"a": 1, "b": {"c": [1, 2	{"a": 1, "b": {"c": [1, 2]}}
Truncated (key)	{"a": 1, "b	{"a": 1}

Table 3. Hand-written failure cases and the output of L1 (all pass in the notebook).

Appendix C Additional results

Family	Raw	+L1	+L2	L3 only	Blind retry	Self-healing
contact	18.9	65.6	66.7	27.8	75.6	81.1
order	21.1	76.7	91.1	23.3	100.0	96.7
event	8.9	46.7	64.4	10.0	65.6	82.2
paper	17.8	45.6	48.9	23.3	56.7	52.2

Table 4. Schema validity (%) per schema family, averaged over models.

Failure type	SmolLM2-360M	Qwen2.5-0.5B	Qwen2.5-1.5B
Markdown code fence	0.0	96.7	60.8
Wrong value type	52.5	0.0	3.3
Missing required field	38.3	0.8	0.0
Extra field	38.3	0.0	0.0
Format/pattern violation	19.2	14.2	2.5
Range/cardinality	1.7	12.5	0.0
Unescaped quote	4.2	0.0	0.0
Unrepairable syntax	1.7	0.0	0.0
Missing comma	1.7	0.0	0.0
Trailing or extra comma	1.7	0.0	0.0
Bracket errors	0.8	0.0	0.0
Other syntax	0.8	0.0	0.0
Truncated output	0.8	0.0	0.0

Table 5. Failure taxonomy (% of initial outputs) per model; the data behind Figure 2.