JQBENCH: A BENCHMARK FOR READING AND EDITING JSON FROM NATURAL LANGUAGE AND/OR EXAMPLES

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce JQBENCH, a new benchmark for evaluating language models on JSON querying and transformation tasks, where the intent can be given specified using natural language and/or examples. Whereas JQBENCH is mainly aimed at using the `jq` tool, it can be used to evaluate other programming languages that query and/or transform JSON. Benchmarks are automatically created from two rich sources of data: Stack Overflow discussions (751 instances with instructions and examples, called JQSTACK) and the Spider dataset for SQL generation from natural language (893 instances with instructions and JSON Schema, called JQSPIDER). We describe and analyze the automated pipeline for benchmark creation, and perform extensive baseline experiments on different models to analyze the complexity and failure modes. Using implicit feedback, the best model (Claude Opus 4.1) scores 77% on the JQSTACK benchmarks and 81% on the JQSPIDER benchmarks. Additionally, we show (1) that access to the documentation surprisingly does not help, (2) `jq` performs comparable to Python, and (3) that automatic feedback (and therefore examples) is crucial. Besides the final benchmarks, we release the intermediate artifacts from each generation step (including failed or invalid conversions) as well as an LLM-friendly version of the documentation, to facilitate further research on JSON querying and transformation.

1 INTRODUCTION

JSON has become the de facto standard for structured data exchange—powering web APIs, databases, event streams, and configuration files—and now underpins modern AI workflows, serving as a common input and output representation for large-language-model (LLM) inference and agentic workflows. A common subset of tasks involves queries and transformations of JSON representations, which can be performed with tools such as `jsonpath` or `jq`.

Consider, for example, a `jq` expression (left) that operates on a social media dataset (right) and selects all users with more than 100 followers and extracts the titles of their posts:

```
.users[]
 | select(.followers > 100)
 | {name, posts: [.posts[].title]}
```

```
{"users": [{"name": "A",
            "followers": 42,
            "posts": [{"title": "X"}]}]}
```

This short query filters on a numerical attribute and simultaneously traverses nested arrays while yielding a new structure. When given the simple task to "find all elements that are present in both the arrays" and three input–output examples like `[[1, 2, 3, 4], [2, 4, 6, 8, 10]]` → `[2, 4]`, models struggle to generate correct expressions. GPT-5 arrives at

```
[[.[0][] as $item | select(.[1] | index($item))]]
```

which does not come close to the simple solution `.[0] - (.[0] - .[1])`.

This illustrates both the expressive power of `jq` and the challenges of generating such transformations from natural language, especially as the constraints grow more complex. Surprisingly, there is no benchmark that jointly captures natural-language prompts and executable JSON queries and transformations.

In this paper, we propose JQBENCH, a benchmark for JSON querying, filtering, and transformation from natural language and/or examples, with a specific focus on the jq query language. The flexible and potentially complex nature of JSON data, the variety of signals that can be part of the input specification—like natural language, examples and JSON schemas—and the expressive yet concise and relatively uncommon nature of jq make JQBENCH an interesting benchmark for different research directions: (1) prompting and agentic workflows using small and large language models, (2) fine-tuning of small language models, or even (3) symbolic inductive programming.

To this end, we collected Stack Overflow questions tagged with jq, as well as NL-to-SQL tasks from an improved version of the Spider dataset, and use automated pipelines to convert them into JQSTACK and JQSPIDER, respectively. From Stack Overflow, our pipeline distills realistic developer problems into machine-checkable tasks by extracting natural-language context, compiling candidate jq expressions and input–output examples, and uses an agent to generate and validate multiple test cases. From Spider, we automatically transform relational databases into JSON databases with associated JSON schemas and derive equivalent jq programs, enabling benchmarks that require reasoning over thousands of nested records. Together these sources yield a diverse corpus of 751 (JQSTACK) and 893 (JQSPIDER) JSON querying and transformation tasks that combine authentic language, rich structure, and automatically verifiable solutions. Additionally, from the JQSTACK creation process, we release 3641 easier tasks and their solutions that can be used in fine-tuning research, both directly and as a seed for synthetic data generation.

Experiments on different models reveals that JQBENCH is sufficiently challenging: Highest baseline of 77% for Claude 4.1 on JQSTACK and 81% on JQSPIDER. Furthermore, the novelty of jq and the unique JSON-processing setting challenge weaker models, while complex JSON operations remain difficult even for stronger ones. We show interesting lessons learned from JQBENCH, including the potential of jq, the importance of implicit feedback based on examples, the "documentation trap" for capable models in agentic loops, and feasibility of JQBENCH as an interesting PBE benchmark.

In summary, we make the following contributions:

1. We present JQBENCH = JQSTACK ∪ JQSPIDER, a benchmark for generating jq expressions that query, filter and transform from natural language and/or examples that covers complex JSON operations over diverse real-world scenarios.
2. We develop an automated pipeline that extracts authentic tasks from Stack Overflow and Spider, synthesizes and executes jq programs and input examples, and verifies correctness through execution feedback.
3. We perform thorough baseline evaluations on different large and smaller models, to understand and analyze properties of both JQBENCH and the current state of language models on jq. Among other experiments, we compare implicit feedback versus explicit feedback (tools), we compare the uncommon jq language versus the very common Python language, and we study the importance of the natural language instruction.

2 JQBENCH

This section describes the two collections of problems that make up JQBENCH: JQSTACK (diverse problems from Stack Overflow with input and output examples) and JQSPIDER (problems adapted from the Spider dataset with large inputs and a schema). For a primer on jq, we recommend reviewing the manual (jq, 2025) and formal specification (Färber, 2024).

All prompts and agent tool signatures used in this section are shown in Appendix A. We use OpenAI's GPT-4.1 (henceforth called *the model*) for all data generation. Besides the final dataset, we release the generation pipeline and all intermediate artifacts, including candidates without a successful conversion and cached model responses, to facilitate further research and development on JQBENCH.

2.1 JQSTACK

Each data point (u, I, E) in JQSTACK consists of an instruction u, two or more inputs I and one or more jq expressions E.

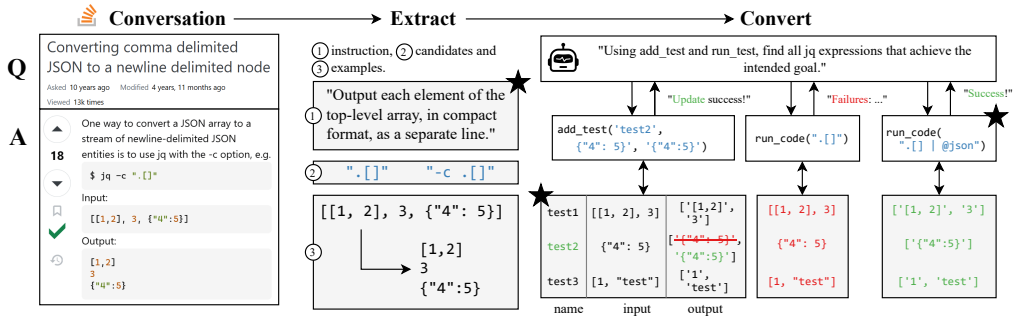

Figure 1: Overview of extraction and conversion of Stack Overflow conversations to JQSTACK tasks. Components tagged with a star (★) are part of the benchmark.

2.1.1 CREATION

We create JQSTACK from all Stack Overflow posts tagged with `jq` in four steps: (1) information extraction, (2) annotation, (3) conversion to test cases and (4) filtering. An overview of the extraction and conversion steps—the core creation process—is shown in Figure 1.

Extraction From each Stack Overflow discussion (question + answers) we instruct the model to generate a `jq` task by extracting (1) direct quotes from the discussion that capture the key problem and solutions, (2) a concise and precise description of the user's intent, (3) all candidate `jq` expressions that satisfy this intent, and (4) optionally, any provided input and output examples.

Annotation Some tasks are not valid, for example, because the instruction involves shell variables or streaming data (which we currently do not support). Intents and candidate expressions are annotated with four properties: (1) if the expression expected additional environment inputs, (2) if it expects streaming input, (3) if it expects multiple input files, and (4) if any information is missing from the task description to map `i` to `o`. Additionally, the model is allowed to mention other problems with the task and/or the environment that cause it to not be a valid task. This retains 5060 tasks.

Conversion Given the intent, the candidate expressions and the candidate examples, we then instruct a model in an agentic loop to write a `jq` expression that satisfies the intent. The agent is given two tools: `add_test(n, i, o)` adds a named test `n` where input `i` is expected to produce output `o` and `run_tests(e)` runs `jq` expression `e` on all tests, providing compilation and execution results. Running `add_test` tool twice with the same name will update the test, which the agent can use to correct test outputs after reflecting on their results. All candidate `jq` expressions are executed on the final tests to determine successes. This yields 4392 valid conversions.

Filtering Many tasks are too trivial to challenge large and capable models, for example, a question to *print the value of a field named "text"* with a target expression `.text`. We therefore use a simple filter to remove tasks solvable by a `jq` generation prompt without interaction with three models (GPT-4.1, GPT-4.1 mini and Phi-4 14B) at temperature 0. All 3793 candidates that are solved by at least one of these models are filtered, leaving **751 challenging tasks that we call JQSTACK**.

2.1.2 ANALYSIS

Figure 2 shows the distribution of number of expressions and tests over tasks. Multiple expressions can be generated, as we (1) instruct the model to generate different expressions based on the candidates extracted from the post and (2) re-evaluate all expressions that are suggested on the final tests. During the extraction step, most tasks only have a single (83.9%) or no (10%) example input. The model still suggests multiple tests, often to evaluate specific (edge) cases (like empty lists or objects).

Each task can be classified based on properties of the input and output JSON. The output can be a **subset** (removing values) or **superset** (adding values) of the input, it can be a new JSON object using only leaf values **extracted** from the input or using all leaf nodes and some **augmented** values, it can

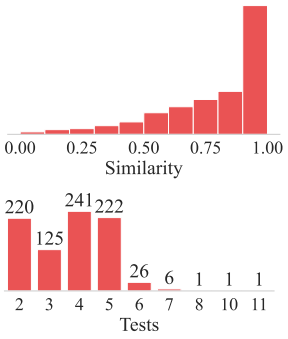

Figure 2: Distributions of (top) similarity to SO solutions and (bottom) # of tests.

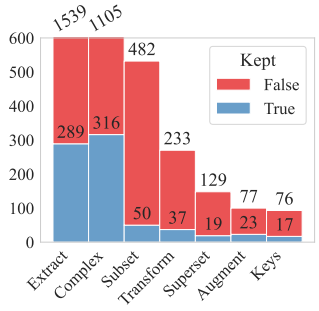

Figure 3: Distribution of failed, filtered or kept per type of task.

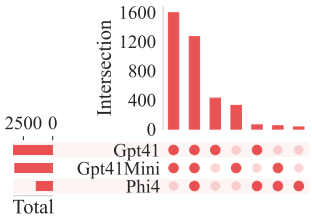

Figure 4: Contribution of each model to task filtering.

have the same structure but with some **transformed** leaf nodes, or it can be a complex transformation that fits into none of these boxes. Complex transformations typically extract nodes, transform them, and then build a new JSON object. Based on the suggested tests, Figure 3 shows the number of failed conversions, number of filtered and number of kept tasks. Most tasks involve extracting or filtering (subset) the input JSON. Tasks that involve changing the structure (extract and complex) are notably harder, as more of them are kept.

Figure 4 shows the number of tasks filtered by each model, as well as their intersections. Even the smaller models uniquely solve at least some problems. Notably, the performance between GPT-4.1 and it's smaller variant is surprisingly small (-2%).

2.2 JQSPIDER

Each data point (u, s, e, d) in JQSPIDER consists of an instruction u, the JSON Schema s, a `jq` expression e, and the dataset d.

2.2.1 CREATION

We create JQSPIDER in three steps from the repaired Spider dataset (Yang et al., 2025): (1) converting each database schema to a JSON Schema, (2) converting SQL databases to JSON databases that adhere to the generated JSON schema, (3) converting SQL queries to `jq` expressions.

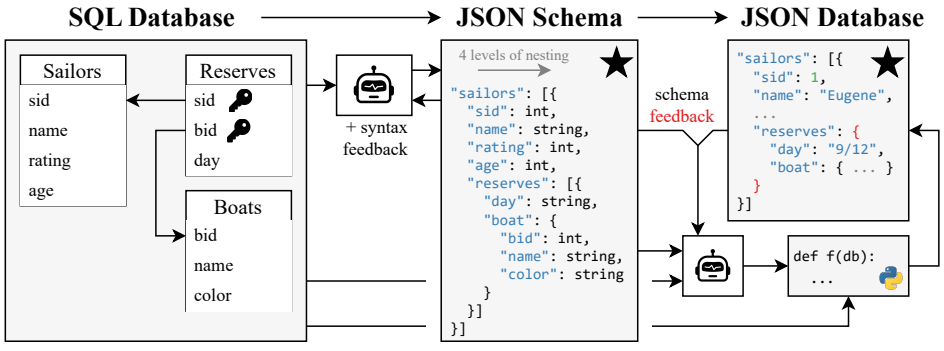

Figure 5: Overview of SQL Database to JSON Database conversion. Components tagged with a star (★) are part of the benchmark.

JSON Schemas We use the model to convert a SQL schema (columns, column types and foreign keys for each table) to a JSON schema. We instruct the model to choose an appropriate *root table* and to leverage nested objects and arrays to represent one-to-one and one-to-many relations. The

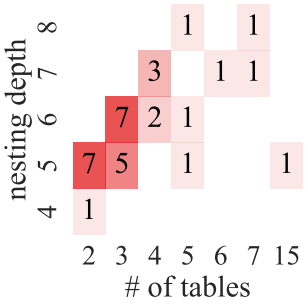

Figure 6: Relation between # of tables and nesting depth.

Figure 7: Distribution of output types.

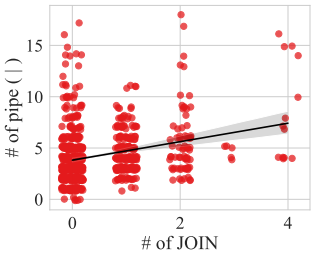

Figure 8: Correlation between # of `JOIN` in SQL and `|` in `jq`.

`jsonschema` Python package[1] is used to provide validation feedback. During small-scale empirical testing, we found that LLMs are able to generate more interesting schemas than symbolic heuristics.

JSON Databases We then use the model to convert the SQL database to a JSON file that adheres to the generated schema. First, we symbolically generate a trivial JSON format that encodes each table separately. Second, we iteratively instruct the model to generate a Python function that converts this JSON file into a new JSON file that adheres to the given schema, using compilation and execution feedback in each iteration. Figure 5 shows an overview of this conversion. Out of 202 databases, this process succeeds for 197 of them.

Queries Finally, given the natural language instruction, SQL query, JSON Schema and expected output of the SQL query on the original database, we iteratively instruct the model to write a `jq` expression, again using compilation and execution feedback in each iteration. This process is very similar to the conversion step in Figure 1 without the `add_test` tool—the tests are provided by the original SQL queries. This yields **893 benchmark tasks**.

2.2.2 Analysis

Figure 6 shows that SQL databases with more tables yield more deeply nested JSON, generally requiring chained `jq` expressions. Figure 7 shows that the output types are quite diverse. List outputs tend to be lists of records. Integers are often `COUNT` queries. Figure 8 shows that more SQL JOINs correspond to more pipe characters in `jq`, confirming the need for long chains.

3 Experiments

We perform extensive experiments to evaluate both our dataset(s) and the current performance of LLMs on writing `jq` expressions.

3.1 Implementation Details and Metrics

We use the `jq` Python binding[2] to execute expressions as `jq.all(e, i)`. This causes all results to be wrapped in a list, for example, `jq.all(".foo", {"foo": 1}) == [1]`. We therefore provide four simple examples in the prompt (including the one above) and also consider a successful evaluation if *each* prediction `jq.all(e, i) == [o]` (or vice versa) for the expected output `o`.

On JQSpider, the keys of record-style outputs are ignored, meaning that `[{"a": 1, "b": 2}] == [{"x": 1, "y": 2}]`. Unless the original query mentions `ORDER BY`, we also ignore order.

We use all but one of the tests as input–output examples. This strategy is common in programming-by-example (Li & Ellis, 2024).

[1] `https://pypi.org/project/jsonschema/`
[2] `https://pypi.org/project/jq/`

3.2 BASELINES

We evaluate different models on JQBENCH in different (agentic) settings that leverage possible reward signals for feedback. The default setting (✔) is based on SELF-DEBUG (Chen et al., 2023) and simply provides feedback based on the available information. Compilation feedback is always provided, execution feedback is provided when inputs are available, and test results are provided when outputs are available. Additionally, we use the following tools in baseline experiments:

- `run_code(e: str, i: str)` runs expression `e` on serialized JSON object `i` and prints the output. This tool can be used even when inputs are not available, as the agent can synthesize its own inputs that it (thinks it) knows the output to. On JQSPIDER, for example, when only a JSON schema is provided because the whole file is too large and there is therefore no way to validate some outputs, it can synthesize smaller examples to test hypotheses.
- `search_docs(k: str[])` searches a (parsed) version of the documentation for keywords `k` and returns the name of all documentation sections that matches any of the keywords.
- `print_docs(s: str[])` prints the sections `s` of the documentation in a Markdown format. Examples of documentation sections are shown in Appendix B, and we release the parsed documentation.

In addition to the `jq` solvers, we compare its performance against using Python to solve JQBENCH. The model is instructed to return a Python function that accepts a single argument (the JSON object). On JQSPIDER, we instruct to return a value or a list of records to match the output format of the expected `jq` expression. The structure of the Python prompt mimics that of the `jq` prompt as close as possible (see Appendix A).

We run all setups over a maximum of eight iterations at temperature 0.

3.3 RESULTS

Table 1 and Table 2 show performance and configuration statistics of different configurations on JQSTACK and JQSPIDER, respectively. The following paragraphs describe these results in more detail.

Contrasting JQSTACK and JQSPIDER. The two `jq` benchmarks reveal notably different behavior and therefore facilitate different areas of research. JQSTACK, which draws from real Stack Overflow questions, exhibits a wider spread of model performance: value-match scores range from roughly 11% (Phi 4) to 77% (Claude 4.1). In contrast, JQSPIDER—derived from Spider database queries—shows strikingly low variability: all models except for Phi 4 are in the 72%–81% range, and even Phi 4 achieves 44%. JQSTACK relies more on deep knowledge of `jq`, including many built-in functions (116 versus 56 different operators and functions in solutions) and the ability to define custom operators (99 versus 2 tasks where a solution defines a custom function). JQSPIDER relies less on deep knowledge of different `jq` operations and more on longer chains of piping `map` and `select` operations together (median of 4 pipes per task versus 3 for JQSTACK).

Language novelty as a performance bottleneck. The difference in performance between Python and `jq` on JQSTACK highlights that models often understand the task, but some do not know how to express a solution using `jq`. This is especially visible for smaller models (57% $\rightarrow$ 11% on Phi 4) but even GPT-5 suffers from the language bottleneck (76% $\rightarrow$ 68%). Phi 4 getting lower value feedback rates (v?) compared to GPT models is due to the fact that it does not obtain an executable expression within 8 iterations. Only Claude, which is known for its strong coding performance, suffers barely any performance loss (-2%). This is reinforced by its low feedback rates: only 5% of `jq` expressions failed to compile and only 11% failed to execute. These results highlight how JQBENCH is a useful testbed for studying how models learn unfamiliar grammars (Cassano et al., 2024; Zhang et al., 2025).

Language structure as a potential performance booster. On JQSPIDER, Claude (+3%), GPT 4.1 (=) and GPT 4.1 mini (+1%) achieve better performance using `jq`. Inspection of the results highlight, among other properties, (1) that filtering of `jq` expressions tends to return all data points that satisfy some criterion whereas Python code is more inclined to greedily return a single result, and (2) that `jq` biases the model towards selecting exactly the required properties instead of returning superfluous

MODEL	CONFIGURATION [1]			# FEEDBACK			# TOOLS			PERFORMANCE		
	A文	🔍🖨	✔	c?	e?	v?	🔍	🖨	</>	c?	e?	v?
Claude 4.1	Jq			–	–	–	–	–	3.62	1.00	0.96	0.74
GPT 4.1	Jq			–	–	–	–	–	2.29	0.89	0.71	0.28
GPT 4.1 mini	Jq			–	–	–	–	–	4.40	0.92	0.77	0.38
GPT 5	Jq			–	–	–	–	–	0.51	0.91	0.66	0.26
Claude 4.1	Jq	✓		–	–	–	1.95	1.42	3.63	0.92	0.86	0.57
GPT 4.1	Jq	✓		–	–	–	0.19	0.04	1.52	0.87	0.65	0.26
GPT 4.1 mini	Jq		✓	0.56	1.51	1.31	–	–	–	0.95	0.80	0.47
GPT 4.1	Jq		✓	0.60	1.15	1.26	–	–	–	0.97	0.84	0.52
Claude 4.1	Jq		✓	0.05	0.11	0.19	–	–	–	1.00	0.96	**0.77**[2]
Phi 4	Jq		✓	4.71	1.52	0.70	–	–	–	0.45	0.26	0.11
GPT 5	Jq		✓	0.31	0.62	0.83	–	–	–	0.99	0.94	0.68
Claude 4.1	Jq	✓	✓	0.38	0.08	0.11	3.40	2.68	–	0.63	0.59	0.45
GPT 4.1	Jq	✓	✓	0.44	0.87	0.94	1.05	0.55	–	0.95	0.84	0.52
GPT 4.1 mini	Jq	✓	✓	0.29	0.74	0.73	1.54	0.31	–	0.96	0.85	0.59
GPT 5	Jq	✓	✓	0.25	0.50	0.71	0.60	0.26	–	0.98	0.94	0.70
Claude 4.1	Python		✓	–	0.02	0.08	–	–	–	0.99	0.97	0.79
GPT 4.1	Python		✓	–	0.11	0.21	–	–	–	1.00	0.98	0.76
GPT 5	Python		✓	–	0.05	0.21	–	–	–	1.00	0.98	0.76
Phi 4	Python		✓	0.01	0.45	1.24	–	–	–	0.94	0.87	0.57

[1] A文 language, 🔍🖨 documentation mode, ✔ implicit mode, (c?) compile, (e?) execution, (v?) value, </> test expression. [2] Best overall `jq` performance.

Table 1: Results for JQSTACK under different configurations.

MODEL	CONFIG[1]		FEEDBACK		TOOLS	PERFORMANCE		
	A文	✔	c?	e?	</>	c?	e?	v?
Claude 4.1	Jq		–	–	1.72	1.00	0.96	0.77
GPT 4.1	Jq		–	–	0.83	0.99	0.94	0.75
GPT 5	Jq		–	–	0.00	0.98	0.91	0.72
Claude 4.1	Jq	✓	0.03	0.08	–	1.00	1.00	**0.81**[2]
GPT 4.1	Jq	✓	0.06	0.12	–	1.00	1.00	0.79
GPT 4.1 mini	Jq	✓	0.01	0.20	–	1.00	1.00	0.77
GPT 5	Jq	✓	0.01	0.11	–	1.00	1.00	0.78
Phi 4	Jq	✓	2.00	1.28	–	0.84	0.74	0.43
Claude 4.1	Python	✓	0.00	–	–	1.00	0.99	0.78
GPT 4.1	Python	✓	–	–	–	1.00	0.99	0.79
GPT 4.1 mini	Python	✓	–	–	–	1.00	0.99	0.76
GPT 5	Python	✓	–	–	–	1.00	0.99	0.80

[1] A文 language, ✔ implicit mode, (c?) compile, (e?) execution, (v?) value, </> test expression. [2] Best overall performance.

Table 2: Results on JQSPIDER.

record elements. In general, the concatenative nature of `jq` presents an interesting property of more advanced code generating methods, such as using code execution as a feedback signal between generating different parts of the code (Ellis et al., 2019; Verbruggen et al., 2025) where the program before each pipe character (`|`) provides all context for the next step to be added.

`jq` as a safe JSON processor. On JQSTACK, Claude almost matches the Python performance using `jq` (-2%) and exceeds Python (+3%) in JQSPIDER. This highlights that, given a powerful enough model, `jq` offers competitive expressive power, standing on essentially equal footing with Python for complex JSON processing tasks, while offering some significant advantages: `jq` has no runtime dependencies, it can operate on streaming data, and it acts as a domain-specific language (DSL) that

can be safely executed without requiring a complicated sandbox. For example, during our Python experiments, models wrote code that wrote to standard output (despite that not being part of the tool specification) and that created new files.

Implicit feedback is crucial. Moving from implicit feedback to letting the model use the `run_code` tool sees a stark decline in performance: -24% on GPT 4.1 (which is not very proficient at using tools), -9% on GPT 4.1 mini, -42% on GPT-5 and even Claude, which is very proficient at using tools, achieves 3% less. We identify four key reasons why letting models explicitly ask for feedback (based on the examples it sees in the prompt) is not working as expected. First, the model does not use any tool calls: GPT 4.1 and GPT 5 did not use any tools in 43.5% and 79.6% of tasks. Second, the model does not (properly) leverage the provided examples: GPT 4.1 did not use all examples in 100% of tasks and when it did, it only used 1.7 out of the available 2.8 tools (on average). Third, the model ignores incorrect outputs: GPT 4.1 used one of the inputs from the examples but ignored the value feedback on 38.2% of tasks. Fourth, the model fails to generate correctly serialized JSON objects. Examples are predicting `["foo":"bar"]` or repetitively predicting the target string `"1,2\n3,4"` instead of the escaped version `"\"1,2\n3,4\""`.

The documentation trap. Claude, which the most proficient at `jq` according to JQBENCH, performs worse when having access to the documentation. The primary cause is getting stuck in a loop of requesting documentation, rather than solving the problem. This is can be observed by the almost triple the documentation request rate by Claude compared to next most eager model (GPT 4.1 mini). After a certain level of proficiency, using SELF-DEBUG pays off more than doing retrieval-augmented generation over the documentation (Zhou et al., 2023). This trap does not hold for models that are less proficient and less eager: GPT 4.1 mini, the model with tool calling that performs worse without documentation and does the second best with the documentation ($47\% \rightarrow 59\%$).

JSON by example. Omitting the natural language instruction converts each task into a programming-by-example (PBE) task, where the goal is to learn a program p such that $p(\texttt{i}) = \texttt{o}$ for all example pairs $(\texttt{i}, \texttt{o})$. PBE is a popular area of research on both symbolic (Gulwani, 2011; Cropper, 2019) and neural fronts (Chen et al., 2018; Shi et al., 2022) with recent attention to LLMs (Li & Ellis, 2024). When instruction simply states that "the query should match the following (input JSON $\rightarrow$ output JSON) examples," the value match changes as $77.5\% \rightarrow 64\%$ (Claude), $67.9\% \rightarrow 57.9\%$ (GPT 5) and $52.4\% \rightarrow 44.5\%$ (GPT 4.1). Whereas a significant decrease—the instruction is expected to provide a strong signal—these results indicate that JQSTACK poses an interesting PBE benchmark.

4 RELATED WORK

4.1 NL-TO-CODE BENCHMARKS

The closest related benchmarks to JQBENCH are DOCSPIDER (Özer et al., 2025) and JSONSCHEMABENCH (Geng et al., 2025), which both target natural-language interfaces for document or schema-centric JSON tasks, but with different emphases.

DOCSPIDER adapts the Spider text-to-SQL dataset to document databases by converting relational data into MongoDB collections and pairing natural-language questions with MongoDB queries. Like DOCSPIDER, we also translate SQL to another representation (`jq` expressions). However, JQBENCH additionally draws on real-world Stack Overflow questions and a wide variety of organically shaped JSON inputs paired with `jq` expressions. This yields tasks that span ad-hoc filtering, restructuring, and rich transformations across highly diverse and variably nested JSON, going far beyond the relatively homogeneous datasets that underpin DOCSPIDER.

JSONSCHEMABENCH evaluates constrained decoding methods for reliably generating JSON outputs that comply with complex schemas. Like JQBENCH, it can be used to assess whether language models respect schema constraints and reason about schema adherence. However, JQBENCH goes significantly further: beyond testing conformance, it stresses the generation of complete *query and transformation programs* (e.g., `jq` or Python) that perform rich filtering, aggregation, and editing on structurally diverse and variably nested data.

Other benchmarks combine natural language with domain-specific expression languages, such as MONGOSH query expressions (MongoDB Education AI, 2025) or Vega-Lite visualization specifications (Luo et al., 2021), illustrating the value of NL-to-JSON benchmarks but within narrower domains. Finally, several benchmarks target NL-to-CLI programs. Terminal-Bench (Team, 2025) is a benchmark for evaluating the ability of agents to operate in terminal environments, including tasks such as *building an initramfs for a kernel*, but targets a small number of complex, multi-step system-administration tasks, with limited data manipulation tasks. NL2Bash (Lin et al., 2018) contains 12K one-line Linux shell commands—such as `top -p \$(pgrep -d',' http)` —mined from Stack Overflow posts. It only contains 2 `jq` commands, however.

Crucially, no existing benchmark supports the breadth of JSON queries and transformations enabled by JQBENCH, which spans filtering, creation, and complex schema-aware reasoning across diverse real-world data.

4.2 BENCHMARK CONSTRUCTION

Constructing high-quality benchmarks is challenging. For example, Yang et al. (2025) found that the widely used Spider dataset (Yu et al., 2018) contains over 30% incorrect NL-to-SQL mappings, highlighting how manual curation can introduce substantial errors. It tempting to avoid these issues by using purely synthetic data; however, Fürst et al. (2024) finds model robustness can suffer when datasets omit authentic user queries.

Researchers have drawn on several approaches for mitigating issues in benchmark construction. Benchmarks, such as ODEX, draw upon natural language queries from Stack Overflow to build a NL-python benchmark (Wang et al., 2023), but with manual test case construction. BIGCODEBENCH (Zhuo et al., 2025) used ODEX as a seed, to synthetically generate more (instruction, code) solutions in an LLM + human annotation loop. Early work like Berant et al. (2013) used weak supervision, learning logical forms from question–answer pairs without full annotations. More recently, researchers have leveraged techniques that include program execution–based filtering, self-consistency checks (Wang et al., 2023), and LLM-driven self-refinement (Madaan et al., 2023).

JQBENCH adopts and extends these ideas by using agentic LLM pipelines that not only synthesize, execute, and automatically validate `jq` programs and their input/output examples, but also extract diverse real-world tasks from sources such as Stack Overflow. This approach lowers the cost of benchmark creation, improves the reliability of final tasks, and grounds the benchmark in authentic problem-solving scenarios.

5 LIMITATIONS

JQBENCH has a few limitations. We currently supports only a single JSON input per task, and do not handle multiple-input or multi-file `jq` programs, though future extensions could relax this restriction to better capture real-world multi-source workflows. The automatic nature of JQBENCH construction, which relies on an LLM to generate a solution from a rich input signal, may have limited the number of challenging tasks in the dataset. We can expand any remaining failed automatic conversions with human annotations and stronger models. Based on the English data sources, JQBENCH is available in English only.

6 CONCLUSION

We present JQBENCH, an automatically constructed benchmark for evaluating language models on JSON querying and transformation tasks. The benchmark includes diverse and challenging real-world problems drawn from Stack Overflow (JQSTACK) and Spider (JQSPIDER) datasets, and supports both NL and PBE tasks in a low-resource language setting. A baseline set of experiments reveal novel insights into effects of output language used for inference, potentially adverse effects of tools in agentic workflows, struggles with novel languages in small language models. Finally, our benchmark will enable future research in execution-guided decoding, exploratory data analysis, language novelty, and tool proficiency.

REFERENCES

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013. URL https://aclanthology.org/D13-1160.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708, 2024.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2023.

Andrew Cropper. Playgol: learning programs through play. In *International Joint Conference on Artificial Intelligence*, 2019.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Michael Färber. A formal specification of the jq language, 2024. URL https://arxiv.org/abs/2403.20132.

Jonathan Fürst, Catherine Kosten, Farhad Nooralahzadeh, Yi Zhang, and Kurt Stockinger. Evaluating the data model robustness of text-to-sql systems based on real user queries, 2024. URL https://arxiv.org/abs/2402.08349.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models, 2025. URL https://arxiv.org/abs/2501.10868.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages*, pp. 317–330, 2011.

jq. jq manual (version 1.8). https://jqlang.org/manual/, 2025. Accessed: 2025-09-24.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2024.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Koiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga (eds.), *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1491/.

Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pp. 1235–1247, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457261. URL https://doi.org/10.1145/3448016.3457261.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

MongoDB Education AI. Natural language to mongodb shell (mongosh) benchmark. Hugging Face Dataset, 2025. URL: https://huggingface.co/datasets/mongodb-eai/natural-language-to-mongosh.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022.

The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments. https://github.com/laude-institute/terminal-bench, April 2025.

Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, and Sumit Gulwani. Execution-guided within-prompt search for programming-by-example. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=PY56Wur7S0.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=wKqdk1sOMY.

Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. Automated validating and fixing of text-to-sql translation with execution consistency. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725271. URL https://doi.org/10.1145/3725271.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Jipeng Zhang, Jianshu Zhang, Yuanzhe Li, Renjie Pi, Rui Pan, Runtao Liu, Zheng Ziqiang, and Tong Zhang. Bridge-coder: Transferring model capabilities from high-resource to low-resource programming language. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10865–10882, 2025.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ZTCxT2t2Ru.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

Arif Görkem Özer, Recep Firat Cekinel, Ismail Hakki Toroslu, and Pinar Karagoz. Docspider: a dataset of cross-domain natural language querying for mongodb. *Natural Language Processing*, pp. 1–32, 2025. doi: 10.1017/nlp.2024.63.

A PROMPTS

(Currently in the supplementary material.)

B DOCUMENTATION

B.1 addpath.md

```
# `addpath(pathArray)`

Ensures path exists, creating objects/arrays; returns modified input.

## Example 1

**Command**: `jq 'addpath(["a",0,"b"])'`
**Input**: `{}`
**Output**: `{"a":[{"b":null}]}`
```

B.2 optional-object-identifier-index.md

```
# `.foo?`

Optional object identifier index. Like `.foo` but suppresses errors when the input is
↪  not an object; produces `null` instead (and when iterated inside arrays, missing
↪  values simply vanish if further filtered).

## Example 1

**Command**: `jq '.foo?'`
**Input**: `{"foo": 42}`
**Output**: `42`

## Example 2

**Command**: `jq '.foo?'`
**Input**: `{"bar": 1}`
**Output**: `null`

## Example 3

**Command**: `jq '[.[] | .a?]'`
**Input**: `[{}, {"a":1}]`
**Output**: `[1]`
```