

How Good Are LLMs at Processing Tool Outputs?

Kiran Kate^{1*}, Yara Rizk^{1*}, Poulami Ghosh², Ashu Gulati³,
Tathagata Chakraborti¹, Zidane Wright¹, Mayank Agarwal¹

¹IBM Research, USA

²Persistent Systems

³Grab

kakate@us.ibm.com, yara.rizk@ibm.com

Abstract

Most realistic task automation problems require large language models (LLMs) to call tools, which often return complex JSON responses. These responses must be further processed to derive the information necessary for task completion. The ability of LLMs to do so is understudied. In this paper, we study the tool response processing task and LLMs' abilities to process structured (JSON) responses. We created a dataset for this task, and evaluated 15 open and closed weight models using multiple prompting approaches¹. Our results show that JSON processing remains a difficult task even for frontier models across multiple prompting strategies. The optimal response processing strategy depends on both the nature and size of the tool outputs, as well as the complexity of the required reasoning. Variations in processing approaches can lead to performance differences ranging from 3% to 50%.

1 Introduction

Tool-augmented large language models (LLMs) (Qin et al., 2023; Li et al., 2023; Xu et al., 2023; Lu et al., 2025; Shen, 2024) can perform tasks beyond their parametric knowledge by calling external tools. A typical multi-turn tool-calling scenario is shown in Figure 1. Given a user's request, an LLM may need to use multiple tools in sequence to address it. Each tool call generated by the LLM is executed and the tool returns data that contains information useful for either responding to the user or for performing the next steps leading to the final answer. Real-world enterprise tools may be existing REST APIs which return large JSON objects as their response (Qin et al., 2024; Zhong et al., 2025). The process of extracting useful information from their response, then, becomes non-trivial.

¹<https://github.com/LongFuncEval/toolJSONprocessing>

A common approach is to simply append the tool response to the LLM context (Yao et al., 2023). This works well for scenarios with small and simple tool outputs (Li et al., 2023; Liu et al., 2024b; Basu et al., 2025). For real-world APIs with long and complex outputs, the response is either added as is or is pruned to fit in the context (Qin et al., 2023; Zhong et al., 2025). Sometimes, LLMs are used to summarize the tool output (Aksitov et al., 2023) or to generate code to process it (Song et al., 2023).

When building real-world agentic systems, it is crucial to understand which of these tool response processing approaches works best. This analysis remains a gap in the literature despite the existence of many works focused on understanding LLMs' tool selection and end-to-end task execution abilities. In this paper, we address this gap by isolating the tool response processing task, creating an evaluation dataset, and analyzing LLMs' performance.

The task is formulated as question-answering (QA) on structured tool responses. Specifically, we consider the JSON response format which is popular among real-world tools (Qin et al., 2024). We manually create the QA dataset based on real tool responses, obtained from RapidAPI². We evaluate 15 high performing LLMs across model families, model sizes, reasoning abilities and context lengths. We compare different prompting techniques including prompting the model to generate the final answer versus to generate Python code to parse the JSON, utilizing chain of thought (CoT) reasoning, and adding various arguments (response, schema, etc.) to the prompt. We structure our analysis across several key dimensions: response length, prompting approach, and question type.

Our experimentation and analysis led to three main conclusions: 1) JSON processing is a non-trivial task even for frontier models (GPT-4o highest accuracy is 77%), 2) models can handle this task better by generating JSON parsing code instead

²<https://rapidapi.com/>

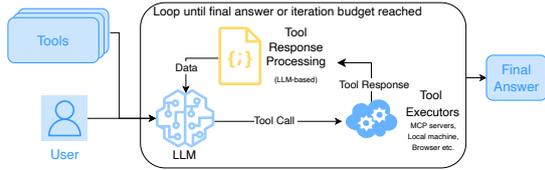


Figure 1: A typical multi-turn tool use pipeline. Tool response processing is shown here as an explicit step to highlight its importance. This paper studies LLM-based approaches to tool response processing and measures the correctness of output "Data" from the processing step by creating a handcrafted evaluation dataset for it.

of deriving the answer by looking at the JSON when the processing involves reasoning over the data (ranging from +3% to +50%), 3) including a schema of the structured data in the prompt improves performance (up to +12%), and 4) including even a condensed version of the response is better than excluding the response entirely when it doesn't fit in the model's context window. Section 4 provides detailed results and ablation studies.

Our contributions are as follows:

- We formulated tool response processing as a QA task and created an evaluation dataset;
- We evaluated 15 open and closed weight models on this task with different processing approaches focusing on 4 main control points;
- We performed an in-depth analysis of LLM performance and derived practical insights to inform real-world applications.

2 Tool Response Processing

The ability of LLMs to extract and/or derive useful information from tool responses is crucial to the success of end-to-end agentic systems. As shown in Figure 1, our goal is to isolate the tool response processing component within an agent and evaluate the state of the art approaches for that task.

2.1 Dataset Creation

The task is formulated as a QA task on tool responses. The dataset consists of triplets (tool response, natural language question, answer) where the tool response is in JSON format and the answer to the question is based on the data in the response. The queries are categorized into one of three types: extractive (returns a value in the JSON given a key), filtering (returns multiple entries corresponding to filtering criteria) and aggregation (combines multiple entries by performing an aggregation operation). Figure 2 shows an example of these three types of

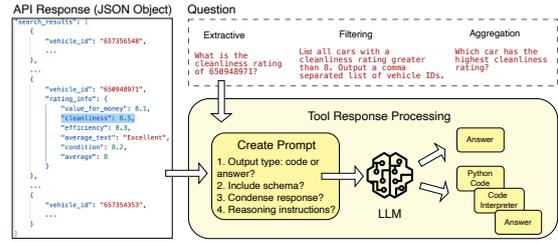


Figure 2: Tool response processing approaches based on prompting an LLM and varying four control points: the type of model output (generate code or the final answer); whether the schema of the API response is included in the prompt; whether the full response is included, condensed or excluded; and whether any reasoning instructions are included (e.g, chain-of-thought type prompts).

questions (with additional examples in Figures 12, 13, 14 in the Appendix).

To perform our study, we collected multiple responses from six REST API endpoints from RapidAPI, covering domains such as hotel booking, car rentals, flight seats, SEC filings, and product search. For the travel related endpoints, we extract the API responses from the ComplexFuncBench dataset (Zhong et al., 2025). We manually reviewed those responses to understand the JSON structure, created question templates and wrote Python code to retrieve the ground truth answers.

Our dataset contains 1298 QA-samples (Table 1), with 580 extractive (derived from 14 templates), 394 filtering (18 templates), and 324 aggregation (18 templates) questions. The JSON responses varied in length (averaging 24,000 to 74,000 characters) and complexity.

2.2 Common Approaches

We consider multiple processing approaches from the literature (Aksitov et al., 2023; Song et al., 2023), all relying on zero-shot prompting of the LLM. The variations in the approaches, captured in Figure 2, stem from four control points. (1) *Model output type*: we can instruct the model to either generate the answer from the response (referred to as *answer generation*) or generate Python code to parse the JSON response to obtain the answer (referred to as *code generation*). The latter approach is based on the observation that LLMs are very familiar with the JSON format and are good at generating code. Such generated code is then executed to get the answer. (2) *Prompt type*: we consider two main templates, one with reasoning instructions (CoT (Wei et al., 2022)), and one without. (3)

Endpoint	Total Samples	Extr.	Filt.	Aggr.	Avg. Response Length (#Characters)
Booking - Get Hotels	13	6	5	2	25,008
Booking - Search Car Rentals	87	34	34	19	24,048
Booking - Get Seat Map	184	29	89	66	53,698
Real Products - Shoes	228	75	75	78	55,968
SEC-Filings	267	84	98	85	74,319
Booking - Get Rooms	519	352	93	74	65,341
Total	1298	580	394	324	-

Table 1: Dataset statistics including sub-type of queries (extractive, filtering or aggregation) per endpoint

Schema: we evaluate the prompt template with and without the JSON schema of the response. (4) *Tool response*: for code generation with schema setups, we consider the impact of including the response (whether in full or a reduced version) vs. excluding it. This enables us to study cases where the response doesn’t fit in the context of an LLM.

We include the example prompt templates in the Appendix C.3. We do not consider approaches based on text summarization or RAG (retrieval augmented generation). During our preliminary testing of such approaches, we observed information loss (particularly in the relationships derived from the structure), reducing our confidence in the suitability of these approaches for JSON processing.

3 Experimental Setup

3.1 Metrics

In the various research questions (RQs) we define, we evaluate performance on the following three metrics: 1) *Exact match* compares the predicted answer to the ground truth answer with some relaxations such as white spaces, casing, order of elements in lists, duplicates, rounding errors. This metric is often considered the strictest as it penalizes the model for verbose outputs and semantic variations. 2) *Contains* is a disjunction between *exact match* and recall based accuracy (which checks whether the ground truth exists within the predicted answer); this gives us an upper bound on programmatic match. 3) *LLM as a judge* employs an LLM to compare the predictions to the ground truth, mostly focusing on their semantic equivalence; this metric is lenient and has a high false positive rate. We used Llama-3-3-70b-instruct as the judge (see Appendix C.3 for the prompt).

3.2 Models

We evaluate 13 open weight high performing models (see Table 5 in the Appendix) with varying sizes (8B to 480B+ active parameters) and 2 closed

weight models: GPT-4o and Claude-4-Sonnet. Inference is done using greedy decoding with the temperature set to 0 and generate a maximum of a 1000 tokens; all our prompts are zero-shot (see Appendix C.3). Since the tool responses in our dataset are large, we only consider models with 65K context or greater.

4 Study Findings

4.1 Impact of Response Length (RQ1)

First, we evaluated whether long responses make the task harder. We designed a controlled setup where we can create a JSON object of a given size from the dataset mentioned in Section 2.1 by merging multiple JSON records from the API responses. The size is indicated by the number of tokens (as per Llama-3-1-70b-instruct tokenizer) and exponentially increases from 10K to 80K tokens.

Figure 3 shows the performance of LLMs based on the *exact match* metric as the JSON response token size is varied. Performance generally declines as response size increases, with GPT-4o experiencing a 7% drop and Mistral-large showing the steepest decline at 91%. LLMs are known to have limitations in handling longer contexts effectively, even for simpler tasks (Kamradt, 2023; Hsieh et al., 2024). This finding confirms the challenge of dealing with tool responses as they become larger.

RQ1: Does the length of the tool response impact the performance of LLMs?

Insight: LLM performance degrades as the length of the tool responses increases.

4.2 Impact of Answer Location (RQ2)

Long context evaluation of LLMs also includes the impact of the position of the key in retrieval tasks. Behaviors such as lost in the middle (Liu et al., 2024a) and recency bias (Peysakhovich and Lerer, 2023) have been reported on other tasks.

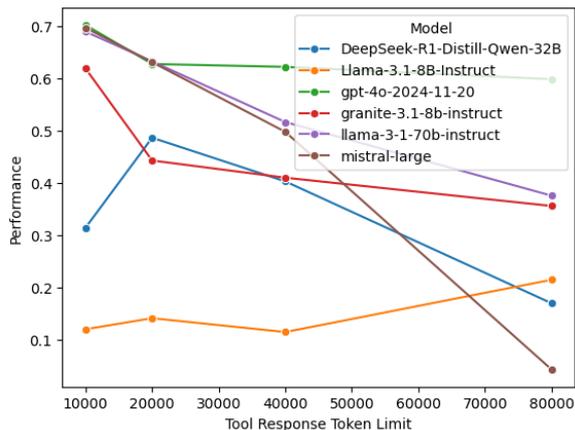


Figure 3: The exact match accuracy across models with increasing JSON size shows that models struggle with longer responses.

To evaluate the impact of the answer’s position on LLM performance, we note that API responses are typically a list of records of some entity type (e.g., flights in flight search). The position of the answer record in the tool response from the triplets (tool response, natural language query, answer) in our dataset can be varied. For the 80K token size setting of RQ1, we vary the position of the answer record (p) from 1 to 8.

Results, in Figure 4, indicate that recency bias was observed in answer position in the JSON (higher values of p denote more recent information). Some models suffered more than others, the performance variation based on position ranges from 5% (GPT-4o) to 75% (Mistral-large).

RQ2: Does the location of the answer impact the performance of LLMs?

Insight: LLMs exhibited recency bias.

4.3 Impact of Model Output Type (RQ3)

In the remaining subsections, we evaluate the performance of various prompting approaches and models on the task of processing JSON tool responses. First, we study the impact of the model output type, two possibilities specifically. The first option is to prompt the model to directly process the JSON response and output the answer (termed “answer” in the Figures). The second option is to prompt the model to generate Python code that is then executed to parse the JSON response and return the answer (termed “code” in the figures).

The “answer” and “code” settings in Figure 6

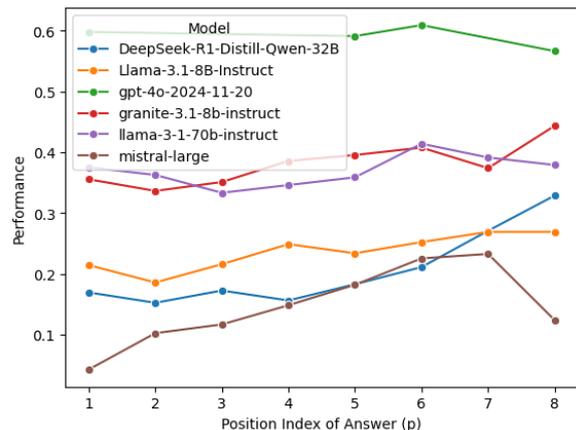


Figure 4: The exact match accuracy of models is impacted by the response’s position indicating that models suffer from recency bias.

capture the results of this experiment. As seen from the results, the best performance out of these two options is close to 76%, underscoring the complexity of the task. The performance difference between the two output types ranges from 2% to 47% confirming that output type is an important factor to take into consideration and models such as GPT-oss-120b are able to work with one much better than the other. In general, the total accuracy when generating code is better for 13 out of 15 models. However, for extractive questions, answer generation is better for 10 out of 15 models including a strong code model such as Claude-4-Sonnet. This shows that the choice of output type is not straight forward and it depends on the type of processing required. As the processing needs become more complex (filtering and aggregation), code generation is more accurate for all the models.

Furthermore, multi-turn conversations increase in length, answer generation based approaches will cause the context to grow much more quickly than code generation based approaches. As such, code generation may be more suitable for longer conversations. Figure 5 shows the performance of both settings when considering the three metrics: exact match, contains, and LLM as a judge (as defined in section 3.1). Exact match is a strict metric that penalizes models for any formatting discrepancies. The contains metric serves as an upper bound for the performance of specific approaches.

We observe that when generating the answer, unlike when generating code, the model is more likely to output superfluous text beyond the answer (as indicated in the gap between exact match and

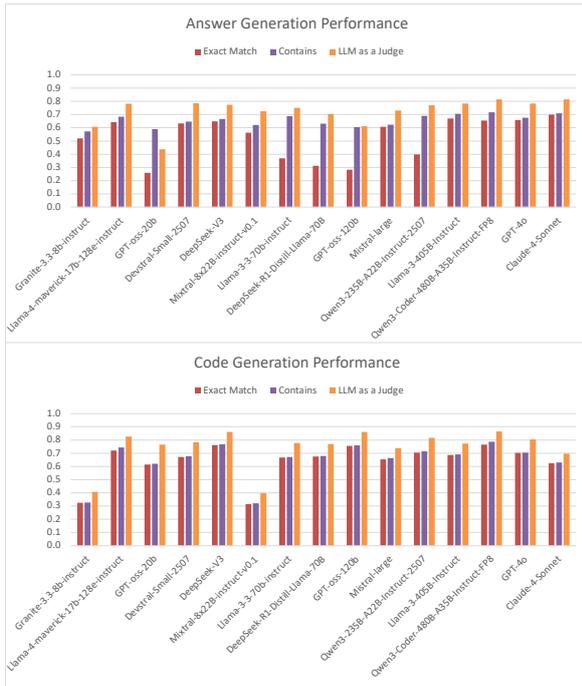


Figure 5: Performance comparison across different metrics for all 15 models when generating the answer (top) and generating the code (bottom) show that exact match is the strictest and LLM as a judge is the most lenient.

contains). This makes it challenging to leverage answer generation in scenarios with strict formatting requirements, such as when the output of one tool must be passed as the input to the next tool. LLM as a judge is often the most lenient of all these metrics. As can be seen, even with the most lenient semantic comparison, performance remains much below 90% across the board.

RQ3: How does the model output type impact the LLMs’ performance?

Insight: Code generation performs better for filtering and aggregation while answer generation is better for extractive.

4.4 Impact of Response Schema (RQ4)

We build on the setup from RQ3 and add the API response JSON schema to determine how the schema impacts the performance of both answer and code generation approaches. The schema captures the structure of the JSON response and information such as data types, default values, and enum values of individual data fields. This experiment is to understand if models are able to use this extra knowledge and perform better. We generate the schema from an API response (details in Appendix).

Task type	No Change	T to F (↓)	F to T (↑)
Extractive	0.85	0.14	0.01
Filtering	0.66	0.08	0.26
Aggregation	0.58	0.02	0.40
Total	0.73	0.09	0.18

Table 2: Change in distribution for samples when switching from answer to code generation (with schema using GPT-4o); lower (↓) is better for True (T) to False (F) and higher (↑) is better for False to True

We can see from Figure 6 that both prompting for the answer and code generally benefit from adding the schema information (all 15 for code generation, 7 out of 15 for answer generation). The performance difference is more significant for code generation than answer generation.

Table 2 shows how the distribution of correct vs. incorrect output varies when we change the setup from answer generation with schema to code generation with schema for GPT-4o (similar distribution patterns were observed for other models). Ideally, we want all the samples that were false to become true ("F to T" column) and none of the true become false ("T to F" column). Results show that the filtering and aggregation samples, which were problematic for answer generation, were mostly positively impacted by the change. The extractive category has a significant number of samples going from true to false which is not ideal. Thus, if it were possible to identify a query as extractive, filtering or aggregation, we can route it to the appropriate approach and attain the best of both worlds.

RQ4: Does adding response schema impact LLM performance?

Insight: LLM performance improves with addition of schema. Higher improvement for code than answer generation.

4.4.1 Classifying Queries

We briefly digress to evaluate whether we can accurately predict whether a question of type extractive, filtering or aggregation. We zero-shot prompted an LLM giving it the question and the API response and ask it to predict the category (see Appendix C.3 for the prompt).

Results in Table 3 show that some LLMs can reasonably accurately predict the class of a query. One

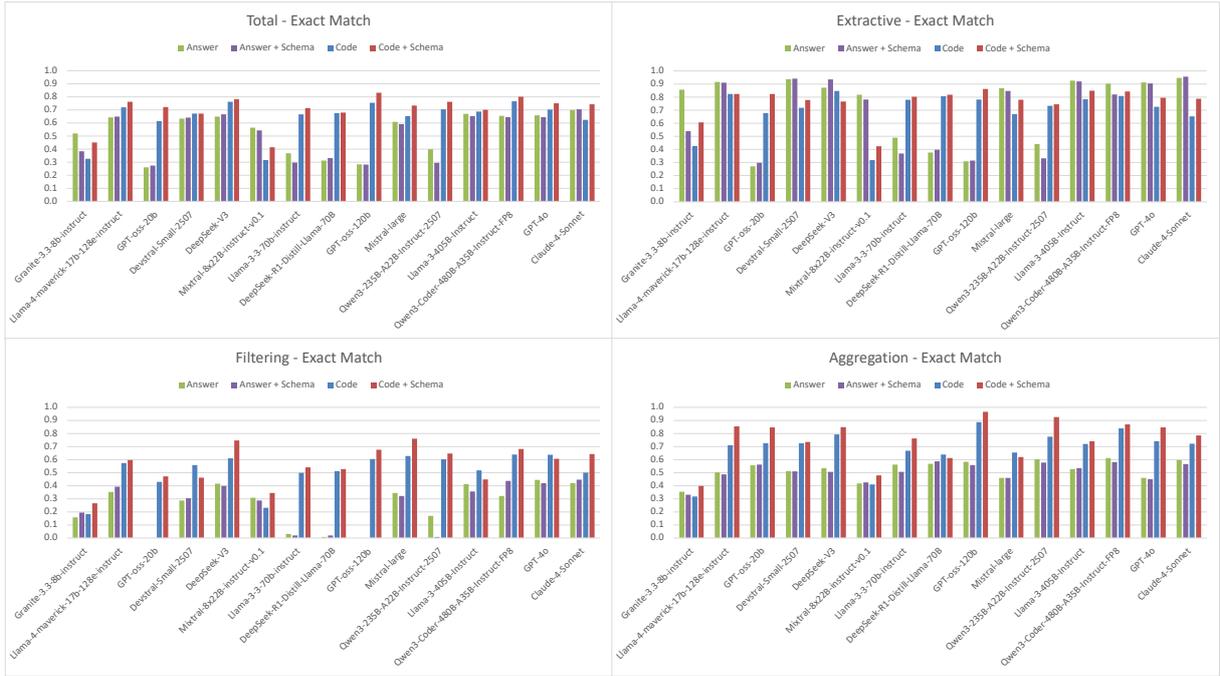


Figure 6: The exact match accuracy comparing model output types shows that generating code is more accurate for filtering and aggregation questions. Adding the response schema generally improves performance across models and question categories. Models are ordered from left to right in increasing number of active parameters.

Model	Total	Extr.	Filt.	Aggr.
DeepSeekV3	0.80	0.69	0.95	0.84
GPT-4o	0.77	0.69	0.70	0.99
GPT-oss-120B	0.76	0.62	0.78	0.99
Llama-3-3-70b-instr.	0.62	0.69	0.73	0.38

Table 3: Models’ accuracy in predicting a question’s category (extractive vs. filtering vs. aggregation). DeepSeekV3 is the most accurate overall.

caveat is that this dataset is small, when considering the question templates to category pairs (50 total unique templates from which 1200+ samples are derived). This may not be representative enough of the space of processing needs and categories that would be encountered in realistic settings.

Maintaining dual processing pipelines significantly increases system complexity. Practitioners must carefully balance this engineering overhead against potential performance gains. As models continue to advance, they will increasingly handle such queries directly, reducing the need for intricate supporting systems. However, certain applications may still benefit from this approach, particularly when constrained by model choices (e.g., open-source models, which remain less capable than proprietary frontier models, often require system-level enhancements to achieve acceptable performance).

4.5 Impact of the JSON Response (RQ5)

Next, we focus on the code generation setup and study the impact of including the API response in the prompt. Specifically, we compare three setups: using the full response (which is not always possible when the response length exceeds the context length), reducing the API response before adding it to the prompt (using a heuristic algorithm described in the Appendix), and not using the response at all (as one could argue the response schema has most of the information needed to generate code).

As seen in Figure 7, passing the full response achieved the highest accuracy across all models. However, when that is not possible due to context length constraints, using a condensed response is better than not adding a response at all. Upon investigation, we observed that the schema does not always include all the information about the values of the keys that may require the model to include specific Python code to properly parse the API response (e.g., “price” is a string and all the values in the response start with “USD”).

Figure 8 shows the percentage of wrong answers attributed to code failures. This allows us to understand whether the code based approaches are failing at the code execution level or at the logical level. Most models are able to generate executable

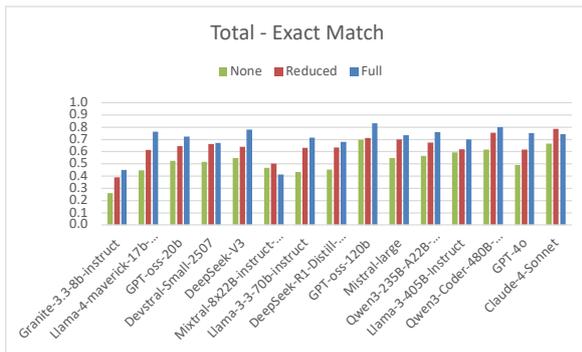


Figure 7: The performance (exact match reported) of the code generation approach on all models is highest when including the full response. However, when that is not possible due to context window constraints, including a reduced version of the response is better than excluding it entirely. It is not possible to perform this experiment on answer generation since the full response must be in the prompt for that setting to work.

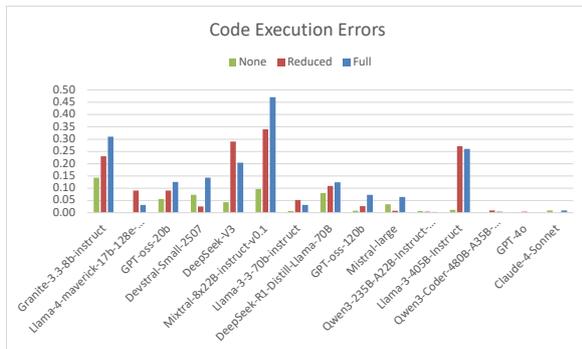


Figure 8: Code execution error rates for the code generation approach indicate that some models are more likely to fail at generating executable code than others. GPT-4o and Claude-4-Sonnet have almost 0% errors indicating that their incorrect answers on some samples can be attributed to semantic understanding of the task or related issues.

code, especially the more capable models like GPT-4o and Claude-4-Sonnet. However, as the context grows, so too does the probability of the wrong answers being caused by failing code.

RQ5: How does the JSON response impact code generation?

Insight: Adding the full JSON response, whenever the context size permits it, will lead to the best possible performance across most model families.

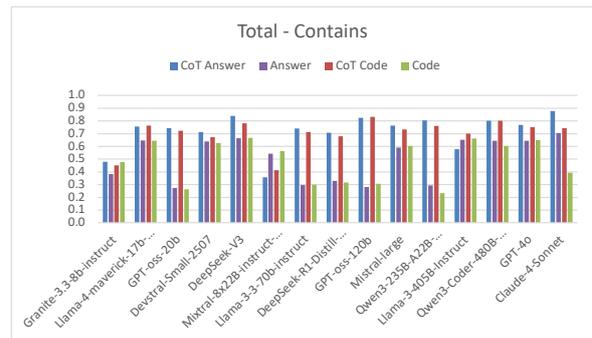


Figure 9: Performance results (contains metric) of the 15 models on reasoning-based prompting (CoT vs. none) show that both answer and code generation benefit from CoT when questions are filtering or aggregation.

4.6 Impact of Reasoning Prompt (RQ6)

Another control point is the prompt we pass to the models, particularly related to whether the prompt includes reasoning-based instructions. Since some questions, especially in the filtering and aggregation categories, may require a level of reasoning for the LLM to determine the answer, we investigated the impact of using a CoT prompt.

Figure 9 reports the results for the 15 models we tested; all models are either confirmed or speculated to have been trained on reasoning traces. We also focus on the contains metric since reasoning instructions will cause models to output superfluous text that will disadvantage the exact match metric. Results show that the benefits of CoT depend on the model used. Breaking down the results further based on the type of question (reported in the Appendix) provide a better understanding of CoT's impact: extractive doesn't benefit but filtering and aggregation do. We also observe that combining reasoning with answer generation can be a good substitute when code generation is not possible.

RQ6: Does CoT impact performance?

Insight: CoT improves LLM performance on filtering and aggregation questions which require more reasoning.

4.7 Error Analysis (RQ7)

4.7.1 Qualitative Analysis

To understand the kinds of errors made in both answer and code generation setups, we manually annotated 50 failures from GPT-4o and Llama-3-3-70b-instruct. Most of the errors were in 4 categories: 1) misinterpretation of the JSON structure,

2) semantic ambiguity, 3) format misalignment, and 4) code execution failure.

Misinterpretation of the JSON structure led to the model picking a value for the right key but from a wrong record. For example, when asked for area of a hotel room given the type of room, the model answered with area of a different room type.

Semantic ambiguity is common for cases when the JSON has multiple keys that have similar meaning. The hotel room availability endpoint response has three semantically similar keys: “name”, “room_name”, and “name_without_policy”. Such noise is representative of real-world APIs and existing tool evaluation datasets manually remove such ambiguity (Zhong et al., 2025).

Format misalignment is when models fail to follow a formatting instruction from the prompt. For example, when asked to return just the value for a key, the model returns a small JSON output with the key and value. Many of the questions in the dataset contain explicit formatting instructions. Frontier models such as GPT-4o make less formatting errors compared to Llama-3-3-70b-instruct.

Common reasons for code execution failure were wrong data type assumptions and incorrect JSON traversal. We expect information such as JSON schema to fix some of these errors and it can be confirmed from Figure 6.

4.7.2 Simplifying the JSON

To validate our observation that complex and large JSON structures were causing a lot of LLM errors, we simplify the JSON using what we call an oracle projection. Since we have the ground truth code to generate the ground truth answer, we use that code to generate a simplified JSON structure. That projection (for both the schema and the response) is then passed to the LLMs when doing answer generation and code generation.

In Figure 10, we compare the performance of a subset of the LLM models on the projected schema and response vs. the full schema and response. We see that the simplified JSON does in fact lead to better performance across all models for both answer and code generation settings.

The projection algorithm we used assumes the ground truth is known which is not realistic. As such this projection is an oracle and provides an upper bound on the performance gains we can achieve if we were able to simplify the JSON structure before prompting the LLM. Furthermore, the average number of characters when using full responses

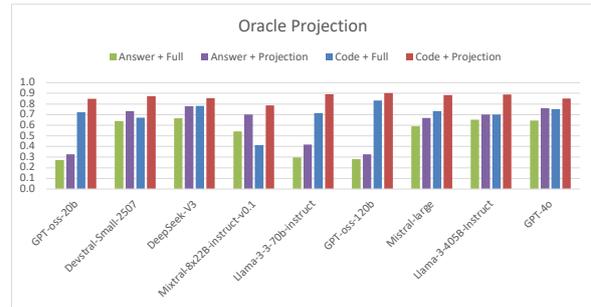


Figure 10: Simplifying the JSON using an approach that ensures the presence of the ground truth answer validates the hypothesis that complex JSON structures are causing LLMs to make mistakes in processing the responses.

was 12x (60,968) higher than the average number of characters when using projected responses (5,056). Therefore, if we can simplify the response, we would also be able to reduce the inference cost while achieving higher accuracy (between 8 and 38 point improvement depending on the model).

RQ7: What are the common types errors observed when LLMs fail?

Insight: The complexity of the JSON structure, output format and semantic ambiguity are the top three observed errors.

5 Related Work

5.1 Tool Calling Evaluation Datasets

The literature has many datasets created to study tool calling functionality in LLMs and agentic systems. Most focused on evaluating LLMs’ ability to formulate tool calls (Basu et al., 2025; Patil et al., 2025; Li et al., 2023). Others focused on the end-to-end task execution of agents with some data samples included tool calling (Trivedi et al., 2024; Zhou et al., 2023; Zhong et al., 2025). None of these datasets enable the detailed evaluation of the tool response processing step.

5.2 Tool Response Processing Approaches

LLMs and tool use has been extensively studied, but most work focuses on the LLMs’ ability to plan and formulate the tool calls correctly (Patil et al., 2023; Schick et al., 2023; Li et al., 2023). Benchmarks using real-world tools (Qin et al., 2024; Zhong et al., 2025) acknowledge long and complex responses; Qin et al. (2024) uses LLMs to

prune parts of a long response and truncates it further to 1024 tokens. [Song et al. \(2023\)](#) has a code generation component to process API responses, however they do not evaluate that component in isolation or extensively on a large dataset.

[Gu et al. \(2024\)](#) evaluates LLMs on tasks on multiple structured formats including JSON, but their dataset is synthetically generated (failing to capture some properties of real-world tools), and they do not include evaluation of code generation. Similarly, [Shorten et al. \(2024\)](#) evaluate LLMs' abilities to parse JSON responses, showing varying degrees of proficiency. However, their evaluation is restricted to one class of models and direct-prompting only. While table processing using LLMs fits within the structured output processing space ([Sui et al., 2024](#); [Chen et al., 2024](#)), tables exhibit flatter structures than JSON structures which are highly nested, making them quite complex for the models to understand.

6 Conclusion

Tool response processing is an important but non-trivial step in a tool-augmented LLM workflow. Understanding the characteristics of such processing and knowing when to use which approach is key to designing an effective system. This paper analyzes structured tool responses approaches and shows that even frontier models have room for improvement. Moreover, the performance varies significantly depending on the type of processing required and the prompting technique used. Our work can serve as guidance on how to approach this task and to drive future performance improvements.

Limitations

Our work deals with a specific challenge in tool-augmented LLMs and hence we focus our experiments on that task. However, it would be important to evaluate the different tool response processing approaches in an end-to-end setting. Moreover, the processing approaches we considered are limited to answer and code generation, but there are other approaches such as RAG on the API response or an agentic approach to code generation.

Ethics Statement

We do not foresee any ethical concerns or opportunities to misuse this work. Furthermore, the data we used (and modified) to evaluate models does not

contain any PII or harmful language. One possible consideration is whether the generate code may cause security concerns. However, there is nothing in our task that would prompt an LLM to attempt a security breach since the prompts all related to extracting answers from JSON responses. Whenever code is generated, there is always concern.

References

- Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Kopparapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, et al. 2023. Rest meets react: Self-improvement for multi-step reasoning llm agent. *arXiv preprint arXiv:2312.10003*.
- Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. [Nestful: A benchmark for evaluating llms on nested sequences of api calls](#).
- Si-An Chen, Lesly Miculicich, Julian Eisenschlos, Zifeng Wang, Zilong Wang, Yanfei Chen, Yasuhisa Fujii, Hsuan-Tien Lin, Chen-Yu Lee, and Tomas Pfister. 2024. [Tablerag: Million-token table understanding with language models](#). *Advances in Neural Information Processing Systems*, 37:74899–74921.
- Zhouhong Gu, Haoning Ye, Xingzhou Chen, Zeyang Zhou, Hongwei Feng, and Yanghua Xiao. 2024. [Structext-eval: Evaluating large language model's reasoning ability in structure-rich text](#).
- Cheng-Ping Hsieh, Simeng Sun, Samuel Krizan, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. [Ruler: What's the real context size of your long-context language models?](#) *arXiv preprint arXiv:2404.06654*.
- Gregory Kamradt. 2023. [Needle in a haystack - pressure testing llms](#).
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. [Api-bank: A comprehensive benchmark for tool-augmented llms](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. [Lost in the middle: How language models use long contexts](#). *Transactions of the Association for Computational Linguistics*, 12:157–173.
- Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Nieves, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024b.

[Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets.](#)

- Hailun Lu, Xingming Li, Xuanyu Ji, Zhigang Kan, and Qingyong Hu. 2025. Toolfive: Enhancing tool-augmented llms via tool filtering and verification. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE.
- Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2025. The berkeley function calling leaderboard (bfc1): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning*.
- Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.
- Alexander Peysakhovich and Adam Lerer. 2023. [Attention sorting combats recency bias in long context language models.](#)
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations*.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.
- Zhuocheng Shen. 2024. Llm with tools: A survey. *arXiv preprint arXiv:2409.18807*.
- Connor Shorten, Charles Pierson, Thomas Benjamin Smith, Erika Cardenas, Akanksha Sharma, John Trengrrove, and Bob van Luijt. 2024. [Structuredrag: Json response formatting with large language models.](#)
- Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.
- Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Binfeng Xu, Xukun Liu, Hua Shen, Zeyu Han, Yuhan Li, Murong Yue, Zhiyuan Peng, Yuchen Liu, Ziyu Yao, and Dongkuan Xu. 2023. Gentopia. ai: A collaborative platform for tool-augmented llms. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 237–245.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. [React: Synergizing reasoning and acting in language models.](#)

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. 2025. Complexfunbench: Exploring multi-step and constrained function calling under long-context scenario. *arXiv preprint arXiv:2501.10132*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023. Webarena: A realistic web environment for building autonomous agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

A Dataset Examples

Figures 12, 13, 14 show three examples from each category (extractive, filtering and aggregation) of QA templates from our dataset.

Table 4 and Figure 11 show that API responses can be exceptionally long. From the API responses we collected, about 20% of the responses would not fit within a 65K context window and about 5% would not fit in a 128K context window (assuming a tokenizer that maps about 4 characters to a token). However, given the processing approaches we wanted to compare, we filtered out such large responses from our dataset. In addition, we limit the list of models to those which support contexts with minimum 65K tokens. So all the API responses in our dataset fit within the context of the models we evaluated.

	Real Product - Shoes	SEC-Filings
Min	14,397	38,642
Max	441,641	4,991,934
Mean	176,718.335	261,526
Stdev	102,880.236	782,602
Median	162,883	89,773

Table 4: Statistics on the length (number of characters) of API responses of two endpoints; some endpoints may have significant variance in the length of the response.

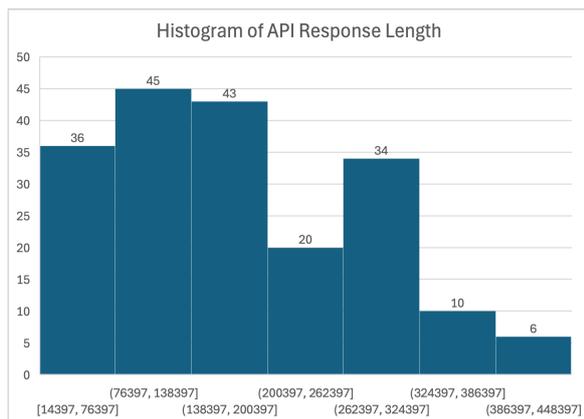


Figure 11: The histogram of API response lengths (number of characters) shows a large variance in response length for the same endpoint (Real Product - Shoes).

B Model Details

Table 5 lists all the models used in this study, their sizes and the pointers to the exact model.

C Tool Response Processing Approaches

C.1 Reduced Response - Heuristic Approach

In the reduced response settings in RQ5, we reduced the length of the response using a heuristic approach. Given our observation that LLMs still benefit from the response because it includes information missing from the schema (such as sample values), we implement an algorithm that keeps one sample of each unique key. Essentially, we eliminate all lists (by keeping only one element). This allows us to present the structure of the JSON and the dependencies between keys while reducing the length.

C.2 Schema

We generate the schema from an API response by using an existing Python library that can do so³. Figure 15 shows one example of a scheme

³<https://transform.tools/json-to-json-schema>

generated for the SEC-Filings endpoint.

C.3 Prompts

Figures 16, 17 19, 18, 20, 21, 22, and 23 provide prompts for the settings mentioned in Section 2. Figure 24 shows the prompt used for the *LLM as a judge* metric. Figure 25 shows the prompt used for the query classifier.

D Detailed Results

D.1 RQ3 and RQ4

Figure 6 illustrates the exact match results of four prompting strategies across 15 models. Since the values may be difficult to read, Table 6 reports the exact numbers used to generate Figure 6.

D.2 RQ6

Filtering and aggregation generally do better with reasoning prompts whether generating the answer or code (12 and 14 models, respectively). Extractive does better without reasoning instructions (14 out of 15 models). When comparing answer vs. code generation with CoT, the former beats the latter on filtering questions for 8 out of 15 models compared to 6 out of 15 for non-reasoning answer vs. code generation comparison.; 10 out of 15 on aggregation questions compared to only 5.

E Impact of API Structure (RQ8)

Our dataset contained six endpoints from different domains and each QA template required processing the JSON to varying depths. In the results below, we attempt to identify correlations between the endpoints and nesting of the JSON.

Figure 27 plots the accuracy per endpoint for Claude-4-Sonnet. We do not observe any correlation between endpoints and which approach is more suitable.

Figure 28 groups the QA pairs by the depth of the answer or the filtering criterion. We do not observe any clear correlation between nesting and performance. Thus, we cannot conclude whether answer generation was better than code generation beyond some threshold of depth.

```

1 -----
2 Extractive QA :
3 -----
4
5 Question Template: f"What is form type of the filing with accession number {acc_num}?"
6 Ground Truth Code:
7     def get_answer(self, acc_num: str, api_response: dict[Any, Any]) -> str:
8
9         for forms in api_response["data"]["attributes"]["result"]:
10             if forms["accessionNumber"] == acc_num:
11                 return str(forms["formType"])
12
13         return "None"
14 Example Question Instance: What is form type of the filing with accession number 0001193125-23-072650?
15 Example Answer Instance: ARS
16 -----
17
18 Question Template: f"How many reviews does {hotel_name} have?"
19 Ground Truth Code:
20     def get_answer(self, api_response: dict[Any, Any], hotel_name: str) -> str:
21         for hotel in api_response["data"]["result"]:
22             if hotel["hotel_name"].strip().lower() == hotel_name.strip().lower():
23                 return str(hotel["review_nr"])
24         return "None"
25 Example Question Instance: How many reviews does "Terra Verde Vacation Rentals" have?
26 Example Answer Instance: 19
27 -----
28
29 Question: f"What is the cleanliness rating of {vehicle_id}?"
30 Ground Truth Code:
31     def get_answer(self, api_response: dict[Any, Any], vehicle_id: str) -> str:
32         for car in api_response["data"]["search_results"]:
33             if car["vehicle_id"].strip().lower() == vehicle_id.strip().lower():
34                 return str(car["rating_info"]["cleanliness"])
35         return "None"
36 Example Question Instance: What is the cleanliness rating of "370878816"?
37 Example Answer Instance: 8.5
38 -----
39
40 -----
41

```

Figure 12: Three examples of extractive question templates and the code to extract the ground truth answer

RQ8: How does the API response structure impact the performance of LLMs on JSON processing tasks?

Insight: There is no clear correlation between the level of nesting or the endpoints and the performance of LLMs.

F Practitioner’s Guidance

The suitability of each processing approach depends on multiple factors. When tool responses are particularly large, appending them in a multi-turn setting will soon make the context run out (even for models with large context, 128K and above). Complex JSON structures such as their nesting depth or semantic similarity of their keys may also impacts this decision.

Answer generation results in verbose outputs compared to code generation, so the choice depends on whether the downstream usage can tolerate verbosity. Custom parsing of LLM output will almost always outperform a generic parser. However, it may not be feasible to implement a parser for each model, especially as models improve over time and new models are created. Thus, using a generic parser (based on widely used programming

languages like Python) becomes desirable.

F.1 Decision Tree

Based on the results we have presented in this paper, we derived a guidance diagram, shown in Figure 29, to help practitioners decide on the best approach to adopt. In general, code generation outperforms answer generation approaches under most circumstances. However, there are some circumstances under which answer generation is the better approach. Namely, when most questions are extractive on simpler JSONs where formatting issues in the LLM output can be tolerated.

F.2 Best Code Generation Model

Furthermore, results show that GPT-OSS-120B was the best overall model for code generation. Thus, whenever possible, using that model to process JSON responses could lead to better results.

F.3 Sandboxing

Finally, it is important to note that we cannot provide any guarantees that the code generated for parsing the responses is safe and trusted. As a precaution against any undesirable consequences from running this code, it would be in the best interest of

```

1 -----
2 Filtering QA :
3 -----
4
5 Question Template: f"List the accession number of all the forms filed in year {year} which are of form
6 type {form_type}. Output a comma separated list of accession numbers."
7 Ground Truth Code:
8 def get_answer(
9     self, api_response: dict[Any, Any], year: int, form_type: str
10 ) -> str:
11
12     access_num = []
13     for filings in api_response["data"]["attributes"]["result"]:
14         if "filingDate" in filings:
15             date = datetime.strptime(filings["filingDate"], "%Y-%m-%dT%H:%M:%S")
16             filing_year = date.year
17             if year == filing_year:
18                 if "formType" in filings and filings["formType"] == form_type:
19                     access_num.append(filings["accessionNumber"])
20
21     if len(access_num) != 0:
22         return ", ".join(access_num)
23     else:
24         return "None"
25 Example Question Instance: List the accession number of all the forms filed in year 2016 which are of
26 form type CERTNYS. Output a comma separated list of accession numbers.
27 Example Answer Instance: 9999999997-16-023115
28 -----
29
30 Question Template: f"Which hotels have free parking? Output a comma separated list of hotel names."
31 Ground Truth Code:
32 def get_answer(self, api_response: dict[Any, Any]) -> str:
33     hotels_list = []
34     at_least_one = False
35     for hotel in api_response["data"]["result"]:
36         try:
37             if hotel["has_free_parking"] == 1:
38                 hotels_list.append(hotel["hotel_name"])
39                 at_least_one = True
40         except BaseException:
41             continue
42     if at_least_one:
43         return ", ".join(hotels_list)
44     else:
45         return "None"
46 Example Question Instance: Which hotels have free parking? Output a comma separated list of hotel
47 names.
48 Example Answer Instance: SOLANA 7 Bed Rooms FAMILY VACATION VILLA-Private Pool and SPA, Casita mi
49 Encanto a Cozy & Spacious House, 4-bedroom 3-Bath & pool, EXCELLENT 7 Bedroom EIR Vacation Villa
50 - PRIVATE POOL, Contemporary Home w Private Pool& Spa, Terra Verde Vacation Rentals
51 -----
52
53 Question Template: f"Show me cars with prices in {currency}? Output a comma separated list of vehicle
54 IDs."
55 Ground Truth Code:
56 def get_answer(self, api_response: dict[Any, Any], currency: str) -> str:
57     vehicle_list = []
58     at_least_one = False
59     for car in api_response["data"]["search_results"]:
60         try:
61             if car["pricing_info"]["base_currency"] == currency:
62                 vehicle_list.append(car["vehicle_id"])
63                 at_least_one = True
64         except BaseException:
65             continue
66     if at_least_one:
67         return ", ".join(vehicle_list)
68     else:
69         return "None"
70 Example Question Instance: Show me cars with prices in INR? Output a comma separated list of vehicle
71 IDs.
72 Example Answer Instance: 370878721, 370878796, 370878761, 370878771, 370878816
73 -----

```

Figure 13: Three examples of filtering question templates and the code to extract the ground truth answer

```

1 -----
2 Aggregation QA:
3 -----
4
5 Question Template: f"How many SEC filings are done in year {year}?"
6 Ground Truth Code:
7     def get_answer(self, api_response: dict[Any, Any], year: int) -> str:
8
9         reports = 0
10        for filings in api_response["data"]["attributes"]["result"]:
11            date = datetime.strptime(filings["filingDate"], "%Y-%m-%dT%H:%M:%S")
12            filing_year = date.year
13            if filing_year == year:
14                reports += 1
15
16        return str(reports)
17 Example Question Instance: How many SEC filings are done in year 2022?
18 Example Answer Instance: 32
19
20 -----
21
22 Question Template: f"What is the average price reported for these hotels?"
23 Ground Truth Code:
24     def get_answer(self, api_response: dict[Any, Any]) -> str:
25         price_list = []
26         for hotel in api_response["data"]["result"]:
27             price_list.append(hotel["min_total_price"])
28         return str(sum(price_list) / len(price_list))
29 Example Question Instance: What is the average price reported for these hotels?
30 Example Answer Instance: 464.746
31
32 -----
33
34 Question Template: f"What is the cheapest base price available?"
35 Ground Truth Code:
36     def get_answer(self, api_response: dict[Any, Any]) -> str:
37         car_price = []
38         for car in api_response["data"]["search_results"]:
39             car_price.append(car["pricing_info"]["base_price"])
40         return str(min(car_price))
41 Example Question Instance: What is the cheapest base price available?
42 Example Answer Instance: 23288.56
43
44 -----

```

Figure 14: Three examples of aggregation question templates and the code to extract the ground truth answer

```

1 -----
2 Sample Schema
3 -----
4 {
5     "$schema": "http://json-schema.org/draft-07/schema#",
6     "title": "Response schema for the SEC filings endpoint.",
7     "type": "object",
8     "properties": {
9         "meta": {
10            "type": "object",
11            "properties": {
12                "copyright": { "type": "string" },
13                "terms": { "type": "array", "items": { "type": "string" } }
14            },
15            "required": [ "copyright", "terms" ]
16        },
17        "data": {
18            "type": "object",
19            "properties": {
20                "attributes": {
21                    "type": "object",
22                    "properties": {
23                        "status": {
24                            "type": "number"
25                        },
26                        "company": {
27                            "type": "object",
28                            "properties": {
29                                "name": {
30                                    "type": "string"
31                                },
32                                "cik": {
33                                    "type": "string"
34                                },
35                                "ticker": {
36                                    "type": "string"
37                                }
38                            },
39                            "required": [ "name", "cik", "ticker" ]
40                        },
41                        "count": {
42                            "type": "number"
43                        },
44                        "result": {
45                            "type": "array",
46                            "items": {
47                                "type": "object",
48                                "properties": {
49                                    "name": {
50                                        "type": "string"
51                                    },
52                                    "accessionNumber": {
53                                        "type": "string"
54                                    },
55                                    "filingDate": {
56                                        "type": "string"
57                                    },
58                                    "formType": {
59                                        "type": "string"
60                                    },
61                                    "url": {
62                                        "type": "string"
63                                    },
64                                    "period": {
65                                        "type": "string"
66                                    }
67                                },
68                                "required": [ "name", "accessionNumber", "filingDate", "formType", "url" ]
69                            }
70                        }
71                    },
72                    "required": [ "status", "company", "count", "result" ]
73                },
74                "type": { "type": "string" },
75                "id": { "type": "string" }
76            },
77            "required": [ "attributes", "type", "id" ]
78        },
79        "status": { "type": "boolean" }
80    },
81    "required": [ "meta", "data", "status" ]
82 }
83 -----

```

Figure 15: Example of an API response's schema

```

1 -----
2 Zero-shot prompt for model output type = answer, without a schema:
3 -----
4
5 You are given a response from an API call (in JSON format).
6 Answer the question based on the information provided in the API response.
7
8 ```json
9 {api_response}
10 ```
11
12 Question: {question}
13
14 Only respond with the answer. Do not include any other text or json in the response.
15 Do not rephrase the answer or write it in complete sentence, return exactly as is from the JSON.
16
17 Answer:
18 -----

```

Figure 16: Prompt for model output type = answer, without a schema

```

1 -----
2 Zero-shot prompt for model output type = answer, with a schema:
3 -----
4
5 You will be given a JSON object as data which is a response from a REST API containing information
6 returned from the API call.
7 You are given a response from an API call (in JSON format).
8 Answer the question based on the information provided in the API response.
9
10 ```json
11 {api_response}
12 ```
13
14 Question: {question}
15
16 Only respond with the answer. Do not include any other text or json in the response.
17 The JSON schema of the object given as data is as follows: {json_schema}
18
19 Answer:
20 -----

```

Figure 17: Prompt for model output type = answer, with a schema

```

1 -----
2 Zero-shot prompt for model output type = answer, without a schema:
3 -----
4
5 You will be given a JSON object as data which is a response from a REST API containing information
6 returned from the API call.
7 Your task is to extract and return <<task_prefix>>
8
9 Write a Python function that:
10 Starts the function with "def ".
11 Takes only the entire api response as input and doesn't have any other input.
12 Identifies the structure of the input data, ensuring it checks for relevant keys and data types.
13 When comparing strings, it should always convert both sides of the comparison to lowercase.
14 Processes the provided data.
15 Iterates through the data to extract relevant information.
16 Cleans numeric strings by removing non-numeric characters before converting them to integers.
17 Performs proper checks to ensure a key exists and is not None before querying its value.
18 Returns only the requested data as a string and no other extra information or words.
19 Do not add any extra keys or terms to the output.
20
21 Final Check:
22 The function must be formatted in Python markdown for direct execution.
23 No explanations, comments, or additional text should be included.
24 Do not include any example usage.
25
26 data = <<json_obj>>
27
28 Python Function:
29 -----

```

Figure 18: Prompt for model output type = answer, without a schema

```

1 -----
2 Zero-shot prompt for model output type = code, with a schema:
3 -----
4
5 You will be given a JSON object as data which is a response from a REST API containing information
6 returned from the API call.
7 You will be given a JSON schema of the response from the REST API returned from the API call.
8 Your task is to extract and return information from the JSON object which follows the JSON schema and
9 answers the user query: <<task_prefix>>
10
11 You need to write a Python function that:
12 Starts the function with "def ".
13 Takes only the entire api response as input and doesn't have any other input.
14 Identifies the structure of the input data, ensuring it checks for relevant keys and data types.
15 When comparing strings, it should always convert both sides of the comparison to lowercase.
16 Processes the provided data.
17 Iterates through the data to extract relevant information.
18 Cleans numeric strings by removing non-numeric characters before converting them to integers.
19 Performs proper checks to ensure a key exists and is not None before querying its value.
20 Returns only the requested data as a string and no other extra information or words.
21 Do not add any extra keys or terms to the output.
22
23 Final Check:
24 The function must be formatted in Python markdown for direct execution.
25 No explanations, comments, or additional text should be included.
26 Do not include any example usage data.
27
28 The JSON schema of the object given as data is as follows: <<json_schema>>
29
30 data = <<json_obj>>
31
32 Python Function:
33 -----

```

Figure 19: Prompt for model output type = code, with a schema

```

1 -----
2 Zero-shot prompt for model output type = code, with a schema but without a response:
3 -----
4
5 Your task is to work with an already-loaded JSON object as a dictionary from a REST API response.
6 Using the provided JSON schema, you need to extract and return information that directly answers the
7 user's query:
8 <<task_prefix>>
9
10 You need to write a Python function that:
11 Starts the function with "def ".
12 Takes only the entire api response as input and doesn't have any other input.
13 Identifies the structure of the input data, ensuring it checks for relevant keys and data types.
14 When comparing strings, it should always convert both sides of the comparison to lowercase.
15 Processes the provided data.
16 Iterates through the data to extract relevant information.
17 Cleans numeric strings by removing non-numeric characters before converting them to integers.
18 Performs proper checks to ensure a key exists and is not None before querying its value.
19 Returns only the requested data as a string and no other extra information or words.
20 Do not add any extra keys or terms to the output.
21
22 Final Check:
23 The function must be formatted in Python markdown for direct execution.
24 No explanations, comments, or additional text should be included.
25 Do not include any example usage data.
26
27 The JSON schema of the object given as data is as follows: <<json_schema>>
28
29 Python Function:
30 -----

```

Figure 20: Prompt for model output type = code, with a schema but without a response

```

1 -----
2 Zero-shot prompt for model output type = code, with a schema but with a reduced response:
3 -----
4 You will be given a JSON object as data which is an example or compact version of the response from a
5 REST API containing information returned from the API call.
6 You will be given a JSON schema of the response from the REST API returned from the API call.
7 Your task is to take reference and return information from the JSON object which follows the JSON
8 schema and answers the user query: <<task_prefix>>
9
10 You need to write a Python function that:
11 Starts the function with "def ".
12 Takes only the entire api response as input and doesn't have any other input.
13 Identifies the structure of the input data, ensuring it checks for relevant keys and data types.
14 When comparing strings, it should always convert both sides of the comparison to lowercase. This
15 is mandatory.
16 Processes the provided data.
17 Iterates through the data to extract relevant information.
18 Cleans numeric strings by removing non-numeric characters before converting them to integers.
19 Performs proper checks to ensure a key exists and is not None before querying its value.
20 Returns only the requested data as a string and no other extra information or words.
21 The data to be returned from the function should be in string.
22 Do not add any extra keys or terms to the output.
23
24 Final Check:
25 The function must be formatted in Python markdown for direct execution.
26 There should be ```python in the beginning and the ending should be ```
27 No explanations, comments, or additional text should be included.
28 Do not include any example usage data.
29
30 The JSON schema of the object given as data is as follows: <<json_schema>>
31
32 data = <<json_obj>>
33
34 Python Function:
35 -----

```

Figure 21: Prompt for model output type = code, with a schema but with a reduced response

```

1 -----
2 Zero-shot prompt for CoT, model output type = answer, with a schema:
3 -----
4
5 You will be given a JSON object as data which is a response from a REST API containing information
6 returned from the API call.
7 You are given a response from an API call (in JSON format).
8 Answer the question based on the information provided in the API response.
9
10 ```json
11 {api_response}
12 ```
13
14 Question: {question}
15
16 Think step-by-step about how the data relates to the question being asked.
17 Analyze the structure and contents of the JSON response.
18 Identify which fields are relevant to the question.
19 Then, reason through the data logically to derive the answer.
20
21 Finally, provide the answer accurately.
22
23 Only respond with the answer. Do not include any other text or json in the response.
24
25 The JSON schema of the object given as data is as follows: <<json_schema>>
26
27 Answer:
28 -----

```

Figure 22: Prompt for CoT, model output type = answer, with a schema

```

1 -----
2 Zero-shot prompt for CoT, model output type = code, with a schema:
3 -----
4
5 You will be given a JSON object as data which is a response from a REST API containing information
6 returned from the API call.
7 You will be given a JSON schema of the response from the REST API returned from the API call.
8 Your task is to extract and return information from the JSON object which follows the JSON schema and
9 answers the user query: <<task_prefix>>
10
11 Think step-by-step:
12 - First, analyze the structure of the JSON data using the schema.
13 - Identify which fields are relevant to the task and understand their data types.
14 - Consider edge cases like missing keys, null values, or unexpected types.
15 - When comparing strings, normalize them to lowercase.
16 - When handling numeric strings, remove non-numeric characters before converting to integers.
17 - Use logical iteration and condition checks to extract the required information.
18 - Ensure that only the expected value is returned, formatted strictly as a string.
19
20 You need to write a Python function that:
21 Starts the function with "def ".
22 Takes only the entire api response as input and doesn't have any other input.
23 Identifies the structure of the input data, ensuring it checks for relevant keys and data types.
24 Processes the provided data.
25 Iterates through the data to extract relevant information.
26 Cleans numeric strings by removing non-numeric characters before converting them to integers.
27 Performs proper checks to ensure a key exists and is not None before querying its value.
28 Returns only the requested data as a string and no other extra information or words.
29 Do not add any extra keys or terms to the output.
30
31 Final Check:
32 The function must be formatted in Python markdown for direct execution.
33 No explanations, comments, or additional text should be included.
34 Do not include any example usage data.
35
36 The JSON schema of the object given as data is as follows: <<json_schema>>
37
38 data = <<json_obj>>
39
40 Python Function:
41 -----

```

Figure 23: Prompt for CoT, model output type = code, with a schema

```

1 -----
2 Prompt for LLM as a judge:
3 -----
4
5 You are given two inputs, one is the ground truth and one is predicted answer.
6 These are two strings. Your task is to output either "True" or "False" indicating whether the
7 predicted answer matches the ground truth.
8 Please follow the instructions listed below:
9 1. As these are free text outputs, the match does not need to be exact.
10 2. Please allow for paraphrases or verbalizations.
11 3. Please output only "True" or "False".
12
13 ground_truth:{gold_answer}
14 predicted_answer:{predicted_answer}
15
16 Does the predicted_answer match the ground_truth?:
17 -----

```

Figure 24: Prompt for the LLM as a judge metric

```

1 -----
2 Prompt for classifying queries:
3 -----
4
5 You will be given a natural language query and are tasked with classifying it into one of the
6 following three categories:
7 - Extractive: returns a value in the JSON given a key
8 - Filtering: returns multiple entries corresponding to filtering criteria
9 - Aggregation: combines multiple entries by performing an aggregation operation
10
11 Return your final answer after "Final answer, category: "
12 Does the following natural language query belong to the "Extractive", "Filtering" or "Aggregation"
13 category?
14 Natural language query: {nl_query}
15 Schema: {schema}
16
17 Final answer, category:
18
19 -----
20

```

Figure 25: Prompt for classifying queries into extractive, filtering or aggregation

Model	# Active Parameters	Model Pointer
Llama-3.1-8B-Instruct	8B	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
granite-3.1-8b-instruct	8B	https://huggingface.co/ibm-granite/granite-3.1-8b-instruct
Granite-3.3-8b-instruct	8B	https://huggingface.co/ibm-granite/granite-3.3-8b-instruct
Llama-4-maverick-17b-128e-instruct-fp8	17B	https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct-FP8
GPT-OSS-20B	20B	https://huggingface.co/openai/gpt-oss-20b
Devstral-Small-2507	24B	https://huggingface.co/mistralai/Devstral-Small-2507
Deepseek-V3	37B	https://huggingface.co/deepseek-ai/DeepSeek-V3
Mixtral-8x22B-instruct-v0.1	39.1B	https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1
Llama-3.1-70B-Instruct	70B	https://huggingface.co/meta-llama/Llama-3.1-70B-Instruct
Llama-3-3-70b-instruct	70B	https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct
DeepSeek-R1-Distill-Llama-70B	70B	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-70B
GPT-OSS-120b	120B	https://huggingface.co/openai/gpt-oss-120b
Mistral-large	123B	https://huggingface.co/mistralai/Mistral-Large-Instruct-2411
Qwen3-235B-A22B-Instruct-2507	235B	https://huggingface.co/Qwen/Qwen3-235B-A22B-Instruct-2507
Llama-3-405B-Instruct	405B	https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct
Qwen3-Coder-480B-A35B-Instruct-FP8	480B	https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct-FP8
GPT-4o-2024-11-20	Unknown	https://openai.com/index/hello-gpt-4o/
Claude-4-sonnet	Unknown	https://www.anthropic.com/claude/sonnet

Table 5: A list of all the models used in our study (sorted by size of active parameters, if known).

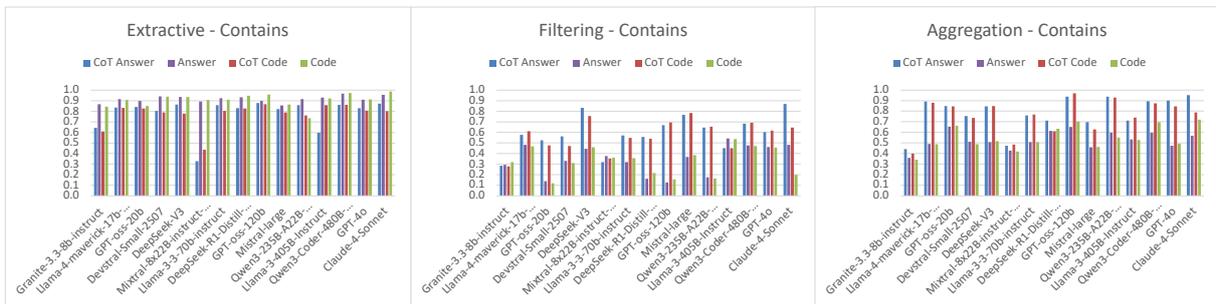


Figure 26: Detailed reasoning-based prompting (i.e., with and without CoT) results for the 15 models indicate that filtering and aggregation question benefit most from CoT prompts.

a practitioner to ensure that execution only happens within a sandbox.

G Examples of Model Errors

This section describes some of the error patterns we see from the model outputs. We took care of handling some of them in the *exact match* metric calculation. For example, when the model is asked to return a comma separated list of some kind of values, the *exact match* computation will handle duplicates and difference in the order of values compared to the ground truth.

1. Comparing the average length of predicted responses to the length of the golden response (in terms of number of characters), some models were very verbose despite instructions in the prompt to only return the answer to the question. This was particularly pronounced for extractive questions and certain models. For example, on about 500 samples where the average gold response had an average of 6 characters, answer generation DeepSeek-R1-distill-Llama-70B led to predictions of 1044 characters (on average). For example,

the question was: "What is the total number of available rooms of the kind Atura King Room - Free cancellation?" Model output: ""20 </think><think> Alright, I need to figure out the total number of available rooms of the kind "Atura King Room - Free cancellation" based on the provided JSON data.

First, I'll look through the "available" array in the JSON. Each object in this array represents a different room type. I'm specifically interested in the rooms where the "name" field is "Atura King Room - Free cancellation".

Looking at the first object in the "available" array, I see that the "name" is indeed "Atura King Room - Free cancellation". The "room_count" for this room is 20.

I'll check the other objects in the "available" array to see if there are any other entries with the same name. The second object has a different name, "Atura King Room - single occupancy - Free cancellation", so it doesn't match. The third object is another variation, and the fourth and fifth also have different names.

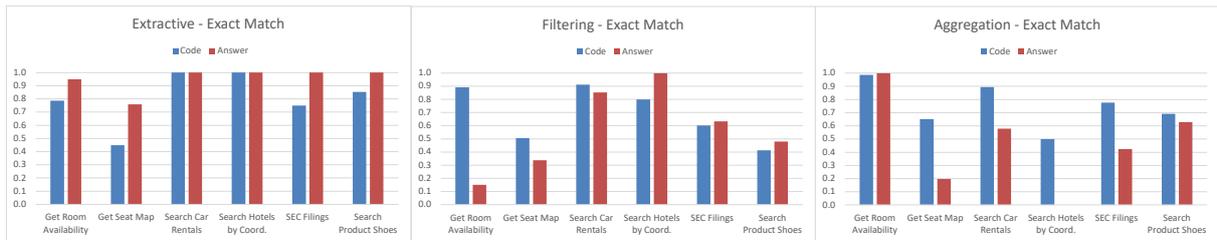


Figure 27: The exact match accuracy per endpoint for Claude-4-Sonnet shows that there is no clear correlation between endpoints and performance of either prompting approach.

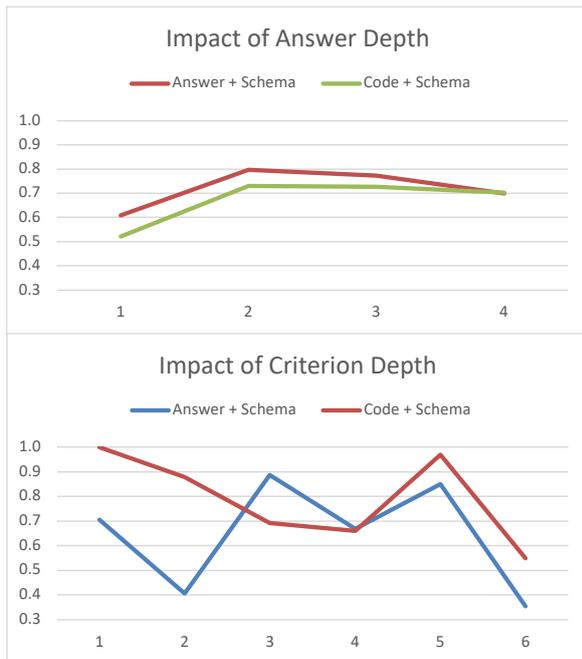


Figure 28: The exact match accuracy per depth of answer or filtering criterion for Claude-4-Sonnet. No clear correlation between accuracy and depth.

Therefore, only the first object has the exact name we're looking for, and its "room_count" is 20. There are no other objects with the same name, so the total number of available rooms of that specific type is 20. </think>""

2. Often, REST API responses have multiple keys that are semantically similar and it is confusing for LLMs to pick the information from the exact key. For example, the Get_Room_List_With_Availability endpoint from booking.com produces a response that has three different keys referring to the room name: name, room_name, and name_without_policy. These have different values in them and the models return wrong values due to such confusion.
3. As mentioned earlier, aggregation operation is

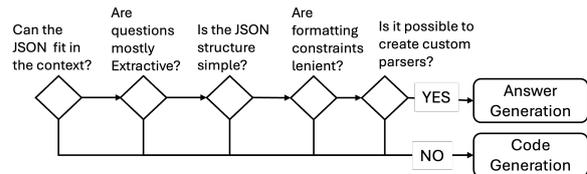


Figure 29: A practitioner's guidance diagram to inform the design decisions when using LLMs to process JSON responses. There is no one size fits all solution and model but there are some combinations that work better than others based on the tasks' characteristics.

harder with answer generation. For example, for the question "List available rooms with gross rate less than 112.89 USD. Output a comma separated list of room names." The models fail to do the comparison in the answer generation setting but such comparison is an easy and precise task for code.

4. In addition to the answer to the question, models regurgitate the input JSON response or output reasoning or sometimes some ambiguous answers for some questions in answer generation. For example, for the question 'Provide the number of shoes which are available in Black colour.' The output from the model is: 'Note: The answer is based on the provided JSON data. The product with ID "13857994671980346845" has a "Color" attribute with value "Black". However, there is another product with ID "16782429352695976508" which has a "Color" attribute with value "Burgundy" and "Maroon" and also has a thumbnail with a product ID "16782429352695976508" which is the same as the product ID of the "Burgundy" and "Maroon" color variant. This product ID "16782429352695976508" is also present in the "Color" attribute of the product with ID "2345888757619488155" with value

"Burgundy" and "Maroon". But there is no product with ID "16782429352695976508" in the provided JSON data. So, the answer is based on the product with ID "13857994671980346845" which has a "Color" attribute with value "Black".'

5. As described earlier, models do not follow formatting instructions. For example, for a question where we just want the exact number as output and the prompt explicitly says do not output the unit, it outputs the unit (such as USD for prices, LB for weight).
6. Code generation sometimes produced different logic which led to incorrect answers, partially attributed to ambiguous or convoluted natural language queries.

Model	Model Output Type	Reasoning	Schema	Response	Total	Extractive	Filtering	Aggregation
Granite-3.3-8b-instruct	Code	None	None	Full	0.325	0.426	0.183	0.318
Llama-4-maverick-17b-128e-instruct	Code	None	None	Full	0.720	0.824	0.574	0.710
GPT-oss-20b	Code	None	None	Full	0.615	0.679	0.429	0.725
Devstral-Small-2507	Code	None	None	Full	0.672	0.719	0.558	0.725
DeepSeek-V3	Code	None	None	Full	0.762	0.848	0.609	0.793
Mixtral-8x22B-instruct-v0.1	Code	None	None	Full	0.315	0.319	0.231	0.410
Llama-3-3-70b-instruct	Code	None	None	Full	0.666	0.781	0.497	0.667
DeepSeek-R1-Distill-Llama-70B	Code	None	None	Full	0.676	0.807	0.513	0.639
GPT-oss-120b	Code	None	None	Full	0.754	0.783	0.604	0.886
Mistral-large	Code	None	None	Full	0.653	0.671	0.627	0.654
Qwen3-235B-A22B-Instruct-2507	Code	None	None	Full	0.704	0.734	0.602	0.775
Llama-3-405B-Instruct	Code	None	None	Full	0.687	0.784	0.518	0.719
Qwen3-Coder-480B-A35B-Instruct-FP8	Code	None	None	Full	0.765	0.809	0.640	0.840
GPT-4o	Code	None	None	Full	0.703	0.726	0.637	0.741
Claude-4-Sonnet	Code	None	None	Full	0.624	0.653	0.500	0.722
Granite-3.3-8b-instruct	Code	None	Full	Full	0.451	0.607	0.266	0.398
Llama-4-maverick-17b-128e-instruct	Code	None	Full	Full	0.763	0.824	0.596	0.855
GPT-oss-20b	Code	None	Full	Full	0.723	0.824	0.472	0.846
Devstral-Small-2507	Code	None	Full	Full	0.671	0.778	0.462	0.735
DeepSeek-V3	Code	None	Full	Full	0.781	0.767	0.746	0.849
Mixtral-8x22B-instruct-v0.1	Code	None	Full	Full	0.414	0.424	0.345	0.478
Llama-3-3-70b-instruct	Code	None	Full	Full	0.713	0.803	0.541	0.762
DeepSeek-R1-Distill-Llama-70B	Code	None	Full	Full	0.679	0.819	0.528	0.611
GPT-oss-120b	Code	None	Full	Full	0.832	0.864	0.675	0.966
Mistral-large	Code	None	Full	Full	0.734	0.781	0.759	0.620
Qwen3-235B-A22B-Instruct-2507	Code	None	Full	Full	0.761	0.747	0.647	0.926
Llama-3-405B-Instruct	Code	None	Full	Full	0.701	0.850	0.449	0.741
Qwen3-Coder-480B-A35B-Instruct-FP8	Code	None	Full	Full	0.801	0.845	0.680	0.870
GPT-4o	Code	None	Full	Full	0.751	0.797	0.607	0.846
Claude-4-Sonnet	Code	None	Full	Full	0.743	0.788	0.642	0.784
Granite-3.3-8b-instruct	Answer	None	None	Full	0.519	0.857	0.157	0.355
Llama-4-maverick-17b-128e-instruct	Answer	None	None	Full	0.643	0.917	0.353	0.503
GPT-oss-20b	Answer	None	None	Full	0.260	0.271	0.000	0.556
Devstral-Small-2507	Answer	None	None	Full	0.634	0.938	0.287	0.512
DeepSeek-V3	Answer	None	None	Full	0.649	0.872	0.416	0.534
Mixtral-8x22B-instruct-v0.1	Answer	None	None	Full	0.563	0.819	0.307	0.417
Llama-3-3-70b-instruct	Answer	None	None	Full	0.368	0.490	0.030	0.562
DeepSeek-R1-Distill-Llama-70B	Answer	None	None	Full	0.312	0.376	0.008	0.568
GPT-oss-120b	Answer	None	None	Full	0.284	0.309	0.000	0.583
Mistral-large	Answer	None	None	Full	0.608	0.869	0.345	0.460
Qwen3-235B-A22B-Instruct-2507	Answer	None	None	Full	0.399	0.441	0.170	0.602
Llama-3-405B-Instruct	Answer	None	None	Full	0.670	0.926	0.411	0.528
Qwen3-Coder-480B-A35B-Instruct-FP8	Answer	None	None	Full	0.654	0.903	0.322	0.611
GPT-4o	Answer	None	None	Full	0.658	0.914	0.444	0.460
Claude-4-Sonnet	Answer	None	None	Full	0.700	0.948	0.419	0.596
Granite-3.3-8b-instruct	Answer	None	Full	Full	0.383	0.540	0.195	0.330
Llama-4-maverick-17b-128e-instruct	Answer	None	Full	Full	0.648	0.910	0.393	0.488
GPT-oss-20b	Answer	None	Full	Full	0.273	0.298	0.000	0.562
Devstral-Small-2507	Answer	None	Full	Full	0.640	0.941	0.305	0.509
DeepSeek-V3	Answer	None	Full	Full	0.666	0.936	0.398	0.506
Mixtral-8x22B-instruct-v0.1	Answer	None	Full	Full	0.543	0.783	0.287	0.426
Llama-3-3-70b-instruct	Answer	None	Full	Full	0.297	0.369	0.020	0.506
DeepSeek-R1-Distill-Llama-70B	Answer	None	Full	Full	0.331	0.398	0.020	0.586
GPT-oss-120b	Answer	None	Full	Full	0.280	0.316	0.000	0.559
Mistral-large	Answer	None	Full	Full	0.592	0.848	0.322	0.460
Qwen3-235B-A22B-Instruct-2507	Answer	None	Full	Full	0.295	0.333	0.008	0.577
Llama-3-405B-Instruct	Answer	None	Full	Full	0.653	0.921	0.355	0.534
Qwen3-Coder-480B-A35B-Instruct-FP8	Answer	None	Full	Full	0.644	0.821	0.437	0.580
GPT-4o	Answer	None	Full	Full	0.644	0.905	0.419	0.451
Claude-4-Sonnet	Answer	None	Full	Full	0.704	0.957	0.447	0.565

Table 6: Exact match accuracy results for RQ3 and RQ4 across 15 models indicate that code generation often outperforms answer generation on filtering and aggregation across models; including the schema almost always boosts results for all settings. Prompts did not include CoT reasoning and used the full API responses.

```

1 -----
2 Question : What is the arrival airport code for flight 1406?
3 -----
4 JSON Snippet (stripped down for clarity):
5 {
6   "departureAirport": {
7     "type": "AIRPORT",
8     "code": "MUC",
9     "name": "Munich Airport",
10    "city": "MUC",
11    "cityName": "Munich",
12    "country": "DE",
13    "countryName": "Germany"},
14   "arrivalAirport": {
15     "type": "AIRPORT",
16     "code": "ARN",
17     "name": "Stockholm Arlanda Airport",
18     "city": "STO",
19     "cityName": "Stockholm",
20     "country": "SE",
21     "countryName": "Sweden"},
22   "departureTime": "2024-11-26T18:40:00",
23   "arrivalTime": "2024-11-26T23:50:00",
24   "legs": [{
25     "departureTime": "2024-11-26T18:40:00",
26     "arrivalTime": "2024-11-26T22:10:00",
27     "departureAirport": {
28       "type": "AIRPORT",
29       "code": "MUC",
30       "name": "Munich Airport",
31       "city": "MUC",
32       "cityName": "Munich",
33       "country": "DE",
34       "countryName": "Germany"
35     },
36     "arrivalAirport": {
37       "type": "AIRPORT",
38       "code": "HEL",
39       "name": "Helsinki-Vantaa Airport",
40       "city": "HEL",
41       "cityName": "Helsinki",
42       "country": "FI",
43       "countryName": "Finland"
44     },
45     "cabinClass": "ECONOMY",
46     "flightInfo": {
47       "flightNumber": 1406,
48       "planeType": "320",
49       "carrierInfo": {
50         "operatingCarrier": "AY",
51         "marketingCarrier": "AY"
52       }
53     },
54     "totalTime": 9000,
55     "departureTerminal": "1"
56   },
57   {
58     "departureTime": "2024-11-26T23:50:00",
59     "arrivalTime": "2024-11-26T23:50:00",
60     "departureAirport": {
61       "type": "AIRPORT",
62       "code": "HEL",
63       "name": "Helsinki-Vantaa Airport",
64       "city": "HEL",
65       "cityName": "Helsinki",
66       "country": "FI",
67       "countryName": "Finland"
68     },
69     "arrivalAirport": {
70       "type": "AIRPORT",
71       "code": "ARN",
72       "name": "Stockholm Arlanda Airport",
73       "city": "STO",
74       "cityName": "Stockholm",
75       "country": "SE",
76       "countryName": "Sweden"
77     },
78     "cabinClass": "ECONOMY",
79     "flightInfo": {
80       "flightNumber": 825,
81       "planeType": "320",
82       "carrierInfo": {
83         "operatingCarrier": "AY",
84         "marketingCarrier": "AY"
85       }
86     },
87     "totalTime": 3600,
88     "arrivalTerminal": "2"
89   }
90 }
91 -----
92 Ground Truth: HEL
93 Predicted: ARN
94 -----

```

Figure 30: Example of an error where the model confused the nesting structure