Reducing hallucination in structured outputs via Retrieval-Augmented Generation

Patrice Béchard
ServiceNow
patrice.bechard@servicenow.com

Orlando Marquez Ayala
ServiceNow
orlando.marquez@servicenow.com

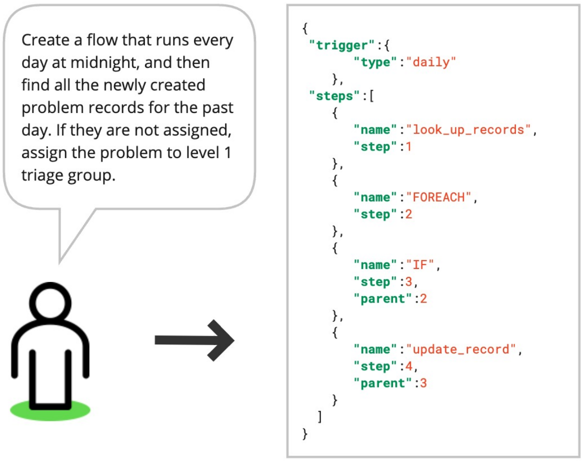
Abstract

A current limitation of Generative AI (GenAI) is its propensity to hallucinate. While Large Language Models (LLM) have taken the world by storm, without eliminating or at least reducing hallucination, real-world GenAI systems will likely continue to face challenges in user adoption. In the process of deploying an enterprise application that produces workflows from natural language requirements, we devised a system leveraging Retrieval-Augmented Generation (RAG) to improve the quality of the structured output that represents such workflows. Thanks to our implementation of RAG, our proposed system significantly reduces hallucination and allows the generalization of our LLM to out-of-domain settings. In addition, we show that using a small, well-trained retriever can reduce the size of the accompanying LLM at no loss in performance, thereby making deployments of LLM-based systems less resource-intensive.

1 Introduction

With the advent of Large Language Models (LLMs), structured output tasks such as converting natural language to code or to SQL have become commercially viable. A similar application is translating a natural language requirement to a *workflow*, a series of steps along with logic elements specifying their relationships. These workflows encapsulate processes that are executed automatically upon certain conditions, thereby increasing employee productivity. While enterprise systems offer such functionality to automate repetitive work and standardize processes, the barrier to entry is high, as building workflows requires specialized knowledge. Generative AI (GenAI) can lower this barrier since novice users can specify in natural language what they want their workflows to execute.

However, as with any GenAI application, using LLMs naively can produce ***untrustworthy*** outputs.



Create a flow that runs every day at midnight, and then find all the newly created problem records for the past day. If they are not assigned, assign the problem to level 1 triage group.

```
{
 "trigger":{
      "type":"daily"
   },
 "steps":[
    {
      "name":"look_up_records",
      "step":1
    },
    {
      "name":"FOREACH",
      "step":2
    },
    {
      "name":"IF",
      "step":3,
      "parent":2
    },
    {
      "name":"update_record",
      "step":4,
      "parent":3
    }
  ]
}
```

Figure 1: Sample structured output (JSON) to generate given a natural language requirement.

Such is the public concern for LLMs producing hallucinations that the Cambridge Dictionary chose *hallucinate* as its Word of the Year in 2023 (Cambridge, 2023). Retrieval-Augmented Generation (RAG) is a well-known method that can reduce hallucination and improve output quality, especially when generating the correct output requires access to external knowledge sources (Gao et al., 2024).

In this work, we describe how, in the process of building a commercial application that converts natural language to workflows, we employ RAG to improve the trustworthiness of the output by reducing hallucination. Workflows are represented as JSON documents where each step is a JSON object. Figure 1 shows an example of a text requirement and its associated JSON document. For simplicity, we include only the basic properties needed to identify a step along with properties indicating the relationship between steps. Besides the workflow steps, there may also be a *trigger* step that determines when the workflow should start, and sometimes this trigger requires a database table name. Hallucination in this task means generating properties such as steps or tables that do not exist.

While fine-tuning a sufficiently large LLM can

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track), pages 228–238
June 16-21, 2024 ©2024 Association for Computational Linguistics

produce reasonably good workflows, the model may hallucinate, particularly if the natural language input is out-of-distribution. As the nature of enterprise users requires them to customize their applications, in this case by adding their own type of workflow steps, a commercial GenAI application needs to minimize the out-of-distribution mismatch. While one could fine-tune the LLM per enterprise, this may be prohibitively expensive due to the high infrastructure costs of fine-tuning LLMs. Another consideration when deploying LLMs is their footprint, making it preferable to deploy the smallest LLM that can perform the task.

Our contributions are the following:

- We provide an application of RAG in workflow generation, a structured output task.
- We show that using RAG reduces hallucination and improves results.
- We demonstrate that RAG allows deploying a smaller LLM while using a very small retriever model, at no loss in performance.

2 Related Work

Retrieval-Augmented Generation is a common approach to limit generation of false or outdated information in classical NLP tasks such as question answering and summarization (Lewis et al., 2020; Izacard and Grave, 2021; Shuster et al., 2021). In the GenAI era, it refers to a process where relevant information from specific data sources is retrieved prior to generating text; the generation is then based on this retrieved information (Gao et al., 2024). Our work differs from standard RAG as we apply it to a structured output task. Instead of retrieving facts, we retrieve JSON objects that could be part of the JSON output document. Providing plausible JSON objects to the LLM before generation increases the likelihood that the output JSON properties exist and that the generated JSON can be executed.

A crucial ingredient of RAG is the retriever since its output will be part of the LLM input. Compared to classical methods such as TF-IDF or BM25 that use lexical information, **Dense Retrieval** has been shown to be more effective as it maps the semantics to a multidimensional space where both queries and documents are represented (Reimers and Gurevych, 2019; Gao et al., 2021; Karpukhin et al., 2020; Xiong et al., 2020). These retrievers are often used in open-domain question answering systems (Guu et al., 2020; Lee et al., 2019), where both queries and documents are unstructured data and thus share the same semantic space. In our case, the queries are unstructured (natural language) and the documents (JSON objects) are structured. Our retrieval training is similar to Structure Aware DeNse ReTrievAl (SANTA), which proposes a training method to align the semantics between code and text (Li et al., 2023b).

Generating structured data falls within the realm of **Structured Output** tasks, which consist of generating a valid structured output from natural language, such as text-to-code, text-to-SQL (Zhong et al., 2017; Yu et al., 2018; Wang et al., 2020) or if-then program synthesis (Quirk et al., 2015; Liu et al., 2016; Dalal and Galbraith, 2020). They are challenging as they not only require generating output that can be parsed, but also entities or field values that exist in a given lexicon; otherwise the resulting output cannot be interpreted or compiled. For simple database schemas or small lexicons, this extra information can be included in the prompt. However, in our task the available pool of steps that can be part of a workflow is potentially very large and customizable per deployment, thereby making in-context learning impractical.

With the arrival of LLMs, these tasks have become more accessible. In particular, **Code LLMs** enable developers to write code faster by providing instructions to the LLM to generate code snippets (Chen et al., 2021; Nijkamp et al., 2022; Li et al., 2023a; Roziere et al., 2023). These models, trained on large datasets of source code (Kocetkov et al., 2022), have acquired broad knowledge of many programming languages and have been shown to perform better at tasks that necessitate reasoning (Madaan et al., 2022). Since the JSON schema to represent workflows is domain-specific, we cannot use these models off-the-shelf. While fine-tuning them on a small dataset increases the quality of results, extra steps are required to reduce hallucination and support out-of-domain queries.

Lastly, an alternative and complementary technique to reduce hallucination with LLMs is **Guided Generation** using tools such as Outlines (Willard and Louf, 2023). A sufficiently expressive context-free grammar could ensure that the steps generated by the model exist, but it does not provide extra knowledge as to which steps the flow should include given the natural language query.

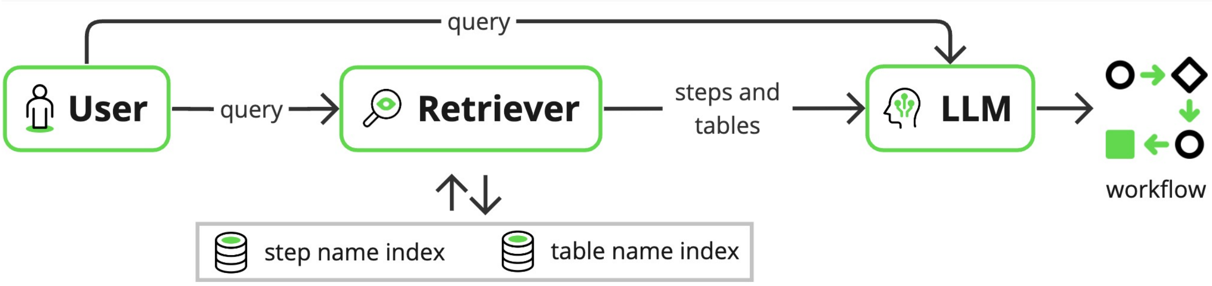


Figure 2: High-level architecture diagram showing how the user query is used by both the retriever and the LLM to generate the structured JSON output.

3 Methodology

Figure 2 depicts the high-level architecture of our RAG system. During initialization, indices of steps and tables are created using the retriever. When a user submits a request, the retriever is called to suggest steps and tables. The suggestions are then appended to the user query to form the LLM prompt. The LLM is then called to generate the workflow in the JSON format via greedy decoding.

To build our system, we first train a retriever encoder to align natural language with JSON objects. We then train an LLM in a RAG fashion by including the retriever's output in its prompt.

3.1 Retriever training

We expect the LLM to learn to construct JSON documents including the relationship between workflow steps, given sufficient examples. The risk of hallucination comes mainly from the step names since there are tens of thousands of possible steps and every customer can add their own steps if the default set does not meet their needs. In addition, as some trigger steps require database table names as a property, these names can also be hallucinated. We therefore require the retriever to map natural language to existing step and database table names.

We choose to fine-tune a retriever model for two reasons: to improve the mapping between text and JSON objects, and to create a better representation of the domain of our application. While there exist a myriad of open-source sentence encoders (Reimers and Gurevych, 2019; Ni et al., 2022), they have been trained in a setting where both queries and documents are in the same natural language semantic space. But in our case, the query or workflow requirement is unstructured while the JSON objects are structured data. Consistent with the results reported by Li et al. (2023b), who search code snippets based on text, fine-tuning improves the retrieval results greatly. Similarly, fine-tuning a model using our domain-specific data allows the retriever to learn the nuances and technicalities of the text and JSON that are particular to our setting.

We use a siamese transformer encoder with mean pooling similar to Reimers and Gurevych (2019) to encode both the user query and the step or table JSON object into fixed-length vectors. We include a normalization layer in our model so that the resulting embeddings have a norm of 1. We generate three embeddings $v_q \in \mathbb{R}^n$, $v_s \in \mathbb{R}^n$, $v_t \in \mathbb{R}^n$:

$$v_q = R(q) \qquad v_s = R(s) \qquad v_t = R(t) \tag{1}$$

where q, s, t are the user query, step, and table respectively. Retriever R can be decomposed as:

$$R(q) = \text{Norm}(\text{MeanPool}(\text{Enc}(q))) \tag{2}$$

The retriever model is trained on pairs of user queries and corresponding steps or tables. Since table names are used only in certain examples depending on the type of trigger, a query can be mapped to zero tables. For instance, the workflow in Figure 1 has four steps, forming four positive training pairs, each pair consisting of the same query and one of the steps in the flow. As the `daily` trigger step does not need a table name, the query is mapped to an empty list of tables.

We also construct negative training pairs by sampling steps or tables that are not relevant to the user query. We experiment with three different negative sampling strategies: random, BM25-based, and ANCE-based (Xiong et al., 2020).

The retriever is trained using a contrastive loss (Hadsell et al., 2006) to minimize the distance between positive pairs ($Y = 1$) and negative pairs ($Y = 0$). Given the cosine similarity between the query and step (or table) vectors, and cosine distance $D = 1 - \text{cossim}(v_q, v_s)$, we define contrastive loss $\mathcal{L}$ as:

$$\mathcal{L} = \frac{1}{2}\left(YD^2 + (1-Y)\cdot \max(0, \frac{1}{2} - D)^2\right) \tag{3}$$

During initialization, we build an index of steps and tables using FAISS (Douze et al., 2024). When a user submits a natural language query, we embed the incoming query using our retriever and use cosine similarity to retrieve the max K steps and tables associated with this requirement.

3.2 LLM training

Contrary to end-to-end RAG systems such as Lewis et al. (2020), we opted to train both the retriever and LLM separately, for simplicity. We use the trained retriever to augment our dataset with suggested step and table names for each example. We then proceed with standard LLM supervised fine-tuning.

```
<|system|>
Tables:[{"name":"table","value":"issue"},
{"name":"table","value":"problem"}]
Steps:[{"name":"log"},
{"name":"update_record"},
{"name":"create_record"}]<|end|>
<|user|>
When a new issue arrives, log the time and date and
create a copy in the problem table<|end|>
<|assistant|>
{"trigger":{"type":"row_create",
"inputs":{"name":"table","value":"issue"}},
"steps":[{"name":"log","step":1},
{"name":"create_record","step":2}]}<|end|>
```

Figure 3: Training example, where the last four lines are the expected output (in red). The underlined text comes from the retriever's output.

By inserting the retriever's output in JSON format into the LLM input, we effectively make this structured output task easier as the LLM can copy the relevant JSON objects during generation. Figure 3 shows an example of a training example. Every line except the last four make up the LLM prompt. The suggested tables and steps come before the user query and are underlined in the figure. We exclude the most frequent steps from these suggestions as we expect the LLM to memorize them. Also, in every LLM training example, we assume the retriever has 100% recall: the steps and table required to build the structured output are always in the suggestions, except for the most frequent steps.

As we are showing the LLM thousands of examples during training, we did not find it necessary to experiment with complicated or verbose prompts: we used a short and simple format, similar to Figure 3, to reduce the number of input tokens while making it clear that this is a structured output task. As shown in section 5.2, this approach yielded good performance.

4 Experiments

As the task we are interested in is part of a commercial enterprise system, we had to devise our own datasets as well as evaluation metrics.

4.1 Datasets

From internal deployments of our enterprise platform, we extracted around 4,000 examples of deployed workflows and asked annotators to write natural language requirements for them. In addition, using deterministic rules, we created around 1,000 samples having simple and few steps in order to teach the model to handle input where the user is incrementally building their workflow. To have an unbiased estimate of the quality of results once the system is deployed, we asked expert users to simulate interacting with the system through a simple user interface where they typed their requirement. We used these interactions and the expected JSON documents to create an additional dataset split, named "Human Eval." Our final metrics are based on this split instead of the "Test" split, due to its higher quality and more realistic input. Table 1 shows statistics for all of our in-domain splits. Not all samples require triggers, and a small subset require the model to generate tables.

Split	Size	# Triggers	# Tables
Train	2867	823	556
Dev	318	77	44
Test	798	247	163
Human Eval	157	99	60

Table 1: Data statistics for in-domain training and evaluation.

A drawback of our data labeling approach is that these internal datasets are mostly in the IT domain, whereas our RAG system can be deployed in diverse domains such as HR and finance. Without assessing the quality of the system in out-of-distribution settings, we cannot be confident that the system will behave as expected. We therefore asked annotators to label five other splits, which come from other deployments of our enterprise platform. These are real workflows that have been created by real users.

Table 2 includes statistics for these out-of-domain splits. A measure of how different they are from our training data is the % of steps that are not in the set of steps in the "Train" split. This discrepancy ranges from less than 10% to more than 70%, highlighting the need to use a retriever and to customize the indices per deployment.

Split	Size	# Triggers	# Tables	% Steps not in Train
OOD1	146	133	47	49%
OOD2	162	111	21	76%
OOD3	429	226	114	34%
OOD4	42	25	11	33%
OOD5	353	271	26	7%

Table 2: Data statistics for out-of-domain evaluation.

To train the retriever encoder, we create pair examples out of the 4,000 extracted and 1,000 deterministically generated samples, resulting in around 15,000 pairs in the step names dataset and 1,500 in the table names dataset. The quality of this encoder is evaluated on the "Human Eval" split described above.

4.2 Metrics

We evaluate the entire RAG system using three metrics, which can all range from 0 to 1:

- **Trigger Exact Match (EM)** verifies whether the generated JSON trigger is exactly the same as the ground-truth, including the table name if this trigger requires it.
- **Bag of Steps (BofS)** measures the overlap between the generated JSON steps and the ground-truth steps in an order-agnostic fashion, akin to a bag-of-words approach.
- **Hallucinated Tables (HT)** and **Hallucinated Steps (HS)** measure the % of generated tables/steps that do not exist per workflow, indicating that they were invented by the LLM. This is the only metric where lower is better.

To evaluate the retriever, we use **Recall@15** for steps and **Recall@10** for tables. That is, given a natural language requirement, we retrieve the top K steps/tables from their respective indices and verify whether they cover the set of steps and the table, if required, included in the JSON document representing the workflow.

4.3 Models

As this is a production system, we have a trade-off between model size and performance for both the LLM and the retriever encoder.

We fine-tune models of different sizes to measure the impact of model size on the final metrics. As StarCoderBase (Li et al., 2023a) has been pre-trained on JSON in addition to many programming languages and comes in different sizes, we fine-tune its 1B, 3B, 7B and 15.5B variants. Given our infrastructure constraints, we could deploy an LLM of at most 7B parameters. Thus we also fine-tune other pretrained LLMs of this size: CodeLlama-7B (Roziere et al., 2023) and Mistral-7B-v0.1 (Jiang et al., 2023). All the LLMs were fine-tuned using the same datasets and hyperparameters.

We use all-mpnet-base-v2[1] as the base retriever model. As it has only 110M parameters, it is suitable for deployment. We compare our fine-tuned model against different sizes of off-the-shelf GTR-T5 models (Ni et al., 2022) to see whether larger encoders impact the performance.

Please see Appendix A for training details for both the LLM and the retriever encoder.

5 Results

5.1 Retriever encoder

Table 3 shows the results of retrieval on the "Human Eval" split for both steps and tables. Scaling the size of the off-the-shelf encoders, as we did with GTR-T5, does not yield significant improvements on both retrieval metrics. A similar observation was made by Neelakantan et al. (2022) for code retrieval. What was crucial to significantly improve the performance was fine-tuning the encoder.

Model (# Params)	Step Recall@15	Table Recall@10
gtr-t5-base (110M)	0.505	0.489
gtr-t5-large (355M)	0.575	0.511
gtr-t5-xl (1.24B)	0.579	0.489
gtr-t5-xxl (4.8B)	0.561	0.489
all-mpnet-base-v2 (110M)	0.425	0.170
+ Random	0.640	0.752
+ BM25	0.537	0.586
+ ANCE	0.556	0.699
+ All	**0.743**	**0.766**

Table 3: Evaluation of different encoders on step and table retrieval. The last four rows represent encoders fine-tuned using different negative sampling strategies.

Due to deployment considerations, we fine-tune the smallest encoders (110M parameters), and found that all-mpnet-base-v2 yielded the best performance after fine-tuning with all negative sampling strategies.

[1] https://huggingface.co/sentence-transformers/all-mpnet-base-v2

Model	Trigger EM	Bag of Steps	Hallucinated Steps	Hallucinated Tables
No Retriever				
StarCoderBase-1B	0.580	0.645	0.157	0.192
StarCoderBase-3B	0.551	0.648	0.140	0.214
StarCoderBase-7B	0.547	**0.669**	0.137	0.206
StarCoderBase (15.5B)	0.632	0.662	0.160	0.194
With Retriever				
StarCoderBase-1B	0.591	0.619	0.072	0.044
StarCoderBase-3B	0.615	0.641	**0.017**	0.030
StarCoderBase-7B	**0.664**	**0.672**	**0.019**	0.042
StarCoderBase (15.5B)	**0.667**	**0.667**	0.040	**0.016**
CodeLlama-7B	0.623	0.617	0.039	0.108
Mistral-7B-v0.1	0.596	0.617	0.049	0.045

Table 4: Performance of various model types and sizes on the "Human Eval" split. Lower is better for the hallucination metrics. Results within 0.005 of the best score are highlighted in **bold**.

5.2 Retrieval-Augmented Generation

Our main objective is to reduce hallucination while keeping the overall performance high given our infrastructure constraints. Table 4 shows that without a retriever (only LLM fine-tuning), the % of hallucinated steps and tables can be as high as 21% on the "Human Eval" split. Using a retriever, this decreases to less than 7.5% for steps and less than 4.5% for tables with all StarCoderBase LLMs. All models produce valid JSON documents following the expected schema, thanks to fine-tuning.

Without a retriever, scaling the size of the StarCoderBase models improves the Bag of Steps and Trigger Exact Match metrics, albeit unevenly. Scaling also helps with RAG, but we observe more consistent improvements. This suggests that larger LLMs can better copy and paste retrieved steps and tables during generation.

The smallest RAG fine-tuned model (1B) hallucinates significantly more than its larger counterparts. Among the other three variants, the 7B version gives us the best trade-off, as the performance difference between 7B and 15.5B is marginal. Another observation is that the 3B version trained with RAG is competitive even with the 15.5B version without RAG on the Trigger EM and Bag of Steps metrics, while keeping hallucination low. This is a key lesson as we could deploy a 3B RAG fine-tuned model if we had more limited infrastructure.

Lastly, we compare the RAG fine-tuned StarCoderBase-7B to fine-tuning more recent LLMs of the same size. Despite also fine-tuning them with RAG, CodeLlama-7B and Mistral-7B-v0.1 produce worse results across all metrics, even compared to the smaller StarCoderBase-3B. We suspect that pre-training on large amounts of natural language data may be detrimental to our task.

5.3 OOD evaluation

We want our approach to perform well on OOD scenarios without further fine-tuning the retriever or the LLM. Table 5 assesses the performance of our chosen RAG fine-tuned StarCoderBase-7B model on the five OOD splits described by Table 2.

Split	Trigger EM	BofS	HS	HT
OOD1	0.662	0.619	0.063	0.051
OOD2	0.645	0.612	0.020	0.151
OOD3	0.562	0.743	0.014	0.033
OOD4	0.400	0.671	0.011	0.154
OOD5	0.774	0.770	0.005	0.063
Avg.	0.647	0.714	0.018	0.066
No RAG Avg.	0.544	0.629	0.020	0.428
Human Eval	0.664	0.672	0.019	0.042

Table 5: Performance of RAG fine-tuned StarCoderBase-7B on OOD splits.

We observe that on average, thanks to the retriever, all the OOD metrics are similar to the in-domain results represented by the "Human Eval" split. We use a weighted average based on the number of samples per split.

To quantify the effect of suggesting step and table names, we evaluate the RAG fine-tuned StarCoderBase-7B model without suggestions in row "No RAG Avg.". All metrics worsen significantly while the "Hallucinated Steps" remains roughly the same. Upon inspection, we see that the RAG fine-tuned model has learned to be conservative in generating steps when it does not receive suggestions, relying only on steps that it has seen during training. On the other hand, the "Hallucinated Tables" metric is significantly worse as the model is more creative when it comes to tables. Please see Appendix B for supplementary detail.

5.4 Error Analysis

When investigating error patterns found in the generated workflows, we observe issues arising from failures both on the retriever and the LLM.

For complex flows where steps that are used less frequently need to be retrieved, if a crucial component is not in the retriever's suggestions, it becomes difficult for the LLM to generate a valid workflow in line with the user query. To improve the retriever's recall, we can decompose the query into shorter texts to make the retrieval step more precise for each step. This would mean performing several retrieval calls, potentially one per step, instead of making one single retrieval call as we are doing now.

In some cases, the LLM did not produce the desired structure. This is more often seen when using steps that determine the logic of the workflow, such as `IF`, `TRY`, or `FOREACH`. These are important errors that can be addressed by synthetic data generation after analyzing which steps are being missed. For examples of perfect output and when the retriever and LLM fail, please refer to Appendix C.

5.5 Impact on Engineering

The obtained results led us to make several decisions that impacted the scalability and modularity of the system. Since the best overall performance was given by a 7B-parameter model, we could have a larger batch size for incoming user requests, thereby increasing the system throughput given a single GPU. This implies a trade-off in latency as larger queries (in number of tokens) result in larger number of generated tokens, sometimes causing large queries to become a bottleneck if they are included in a batch with many shorter queries. Our stress tests and user research reveal that the current system overall response time is acceptable.

Obtaining good results after fine-tuning a very small encoder for the retriever (110M parameters), allowed us to deploy it on the same GPU with negligible effect on the larger LLM. But we could even deploy the retriever on CPU due to its small size. A benefit of not performing joint training between the retriever and the LLM is that the retriever can be reused for other use cases involving similar data sources. Moreover, decoupling them allows clearer separation of concerns and independent optimization by separate team members. Nevertheless, for scientific purposes, it is still worthwhile to experiment with joint training.

We have several ideas to reduce the system response time: changing the structured output format from JSON to YAML to reduce the number of tokens, leveraging speculative decoding (Leviathan et al., 2023; Chen et al., 2023; Joao Gante, 2023), and streaming one step at a time back to the user instead of the entire generated workflow.

6 Conclusion

We propose an approach to deploy a Retrieval-Augmented LLM to reduce hallucination and allow generalization in a structured output task. Reducing hallucination is a sine qua non for users to adopt real-world GenAI systems. We show that RAG allows deploying a system in limited-resource settings as a very small retriever can be coupled with a small LLM. Future work includes improving the synergy between the retriever and the LLM, through joint training or a model architecture that allows them to work better together.

Ethical Considerations

While our work proposes an approach to reduce hallucination in structure output tasks, we do not claim that the risk of harm due to hallucination is eliminated. Our deployed system includes a layer of post-processing to clearly indicate to users the generated steps that do not exist and urge them to fix the output before continuing their work.

Acknowledgements

We thank our ServiceNow colleagues who worked hard in building the aforementioned system, from project managers to quality engineers. We also thank the several colleagues who reviewed an earlier version of this paper: Lindsay Brin, Hessam Amini, Erfan Hosseini, and Gabrielle Gauthier-Melançon, as well as the NAACL reviewers, for their valuable feedback.

References

Cambridge. 2023. Why hallucinate? https://dictionary.cambridge.org/editorial/woty.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dhairya Dalal and Byron V Galbraith. 2020. Evaluating sequence-to-sequence learning models for if-then program synthesis. *arXiv preprint arXiv:2002.03485*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. In *2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pages 6894–6910. Association for Computational Linguistics (ACL).

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2024. Retrieval-augmented generation for large language models: A survey.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3929–3938.

Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2006*, pages 1735–1742.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Gautier Izacard and Edouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering. In *EACL 2021-16th Conference of the European Chapter of the Association for Computational Linguistics*, pages 874–880. Association for Computational Linguistics.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Joao Gante. 2023. Assisted generation: a new direction toward low-latency text generation.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics.

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6086–6096.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pages 9459–9474.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xinze Li, Zhenghao Liu, Chenyan Xiong, Shi Yu, Yu Gu, Zhiyuan Liu, and Ge Yu. 2023b. Structure-aware language model pretraining improves dense retrieval on structured data. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11560–11574, Toronto, Canada. Association for Computational Linguistics.

Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. *Advances in Neural Information Processing Systems*, 29.

Ilya Loshchilov and Frank Hutter. 2018. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pretraining. *arXiv preprint arXiv:2201.10005*.

Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernandez Abrego, Ji Ma, Vincent Zhao, Yi Luan, Keith Hall, Ming-Wei Chang, et al. 2022. Large dual encoders are generalizable retrievers. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9844–9855.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702*.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *International Conference on Learning Representations*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. 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*, pages 3911–3921.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Training details for LLM and retriever

All LLMs were fine-tuned using the same set of hyperparameters. We use the AdamW optimizer with a learning rate of $5e-4$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and weight decay of 0.01. Models were trained for 5,000 steps with a cosine learning rate scheduler with 100 warmup steps. We use an effective batch size of 32 for all models, using gradient accumulation when the batch size would not fit on a single GPU. We trained all models using LoRA (Hu et al., 2021) with $r = 16$, $\alpha = 16$ and a dropout rate of 0.05. All models were trained with flash-attention (Dao et al., 2022) on a single A100 80GB GPU.

We fine-tuned the retriever model using the SentenceTransformers framework (Reimers and Gurevych, 2019). We use the AdamW optimizer (Loshchilov and Hutter, 2018) and a learning rate of $2e-5$. We use a batch size of 128 and train the model for 10 epochs.

B Differences in generation with and without suggestions

To understand the impact of suggesting step and table names during generation, for each OOD split, we inspect the % of unique steps and % of unique table names that are hallucinated with and without suggestions.

Table 6 shows that without suggestions, the RAG fine-tuned StarCoderBase-7B tends to generate significantly fewer unique step names. Receiving suggestions allows the model to copy the suggestions, thereby increasing the diversity of what it generates. In addition, without suggestions a greater percentage of the unique step names it generates are invented.

	No suggestions		With suggestions	
Split	# unique steps	% H	# unique steps	% H
OOD1	52	40%	100	13%
OOD2	38	34%	96	13%
OOD3	122	37%	269	9%
OOD4	20	5%	32	9%
OOD5	88	17%	151	3%

Table 6: Statistics of generated step names in terms of uniqueness and hallucination. H refers to unique hallucinated step names.

We also see that even with suggestions, there is still an important gap in the percentage of unique step names that are hallucinated, as in some splits more than 10% of unique steps are invented. While the overall hallucination rate is less than 2%, as shown in Table 5, there are cases where the retriever does not suggest what is expected or the LLM does not take into account the suggestions.

	No suggestions		With suggestions	
Split	# unique tables	% H	# unique tables	% H
OOD1	40	70%	22	14%
OOD2	31	71%	19	21%
OOD3	61	64%	44	9%
OOD4	11	54%	9	22%
OOD5	38	68%	29	17%

Table 7: Statistics of generated table names in terms of uniqueness and hallucination. H refers to unique hallucinated table names

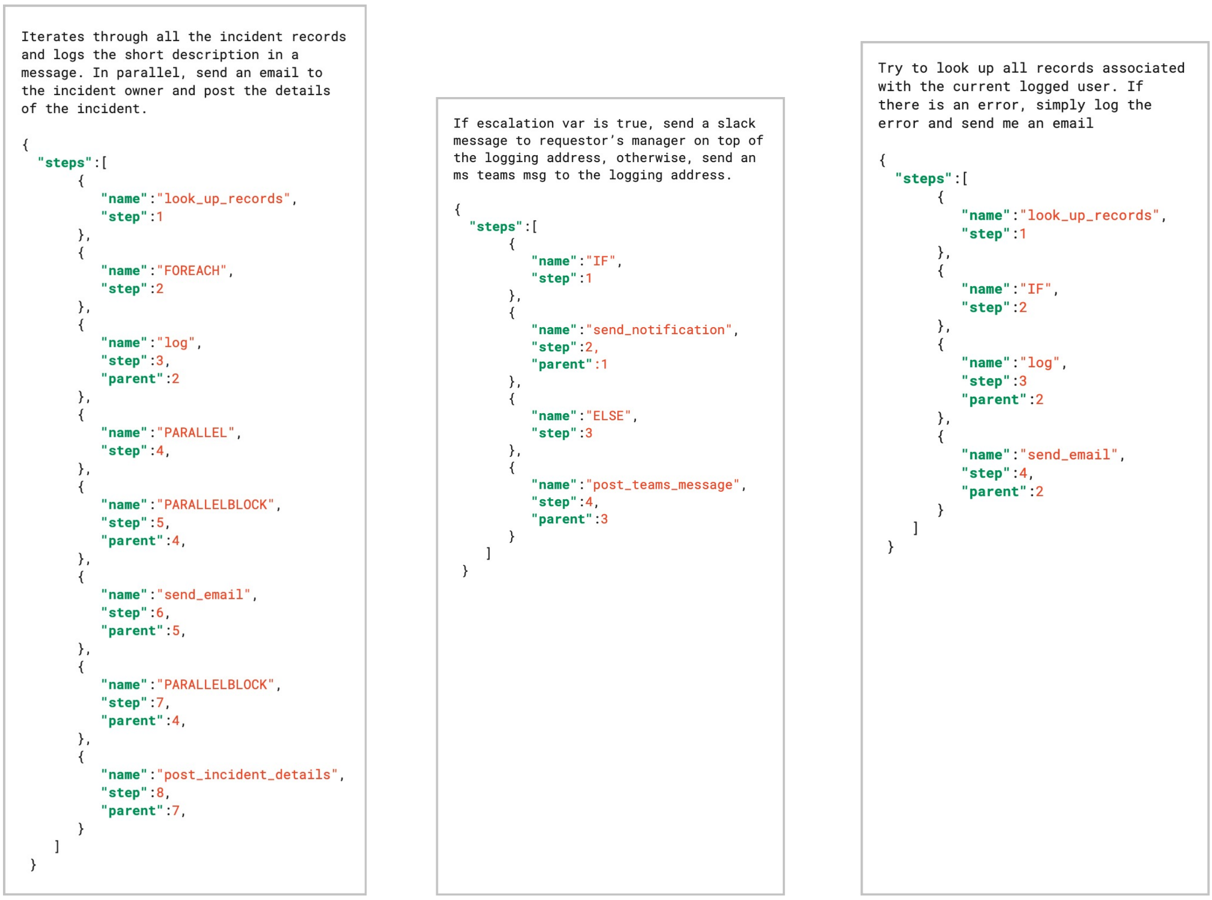
When it comes to table names, there are similar and different observations, as shown in Table 7. As in the case of step names, without suggestions a greater percentage of unique table names are invented. However, when provided with suggestions, the model is more conservative as it generates fewer unique table names. This may be an artifact of the data, where there is less diversity of tables used compared to step names.

C Sample perfect output and errors

Figure 4 shows three user queries along with their generated workflows. The first one is a complicated workflow where the LLM is able to follow exactly the structure described in the user query, and is able to use the steps that the user expected. In this case, the retriever suggests only the step `post_incident_details`, as the rest are considered common steps.

In the second example, the retriever fails to suggest the `send_slack_message` step. The resulting workflow is not entirely wrong but it is of lesser quality as the LLM uses the common step `send_notification`, which is not what the user expected.

In the last example, the LLM shows that it does not sufficiently understand the semantics of the task. The word *Try* in the user query should have made it use the `TRY` and `CATCH` flow logic, but the LLM seems to ignore this word, resulting in a workflow that does not reflect what the user asked for.



```
Iterates through all the incident records
and logs the short description in a
message. In parallel, send an email to
the incident owner and post the details
of the incident.

{
  "steps":[
       {
          "name":"look_up_records",
          "step":1
       },
       {
          "name":"FOREACH",
          "step":2
       },
       {
          "name":"log",
          "step":3,
          "parent":2
       },
       {
          "name":"PARALLEL",
          "step":4,
       },
       {
          "name":"PARALLELBLOCK",
          "step":5,
          "parent":4,
       },
       {
          "name":"send_email",
          "step":6,
          "parent":5,
       },
       {
          "name":"PARALLELBLOCK",
          "step":7,
          "parent":4,
       },
       {
          "name":"post_incident_details",
          "step":8,
          "parent":7,
       }
    ]
 }
```

(a) Perfect output

```
If escalation var is true, send a slack
message to requestor's manager on top of
the logging address, otherwise, send an
ms teams msg to the logging address.

{
  "steps":[
       {
          "name":"IF",
          "step":1
       },
       {
          "name":"send_notification",
          "step":2,
          "parent":1
       },
       {
          "name":"ELSE",
          "step":3
       },
       {
          "name":"post_teams_message",
          "step":4,
          "parent":3
       }
    ]
 }
```

(b) Retrieval error

```
Try to look up all records associated
with the current logged user. If
there is an error, simply log the
error and send me an email

{
  "steps":[
       {
          "name":"look_up_records",
          "step":1
       },
       {
          "name":"IF",
          "step":2
       },
       {
          "name":"log",
          "step":3
          "parent":2
       },
       {
          "name":"send_email",
          "step":4,
          "parent":2
       }
    ]
 }
```

(c) LLM error

Figure 4: Examples where both the retriever and the LLM worked perfectly and where each of them failed: (a) All expected step names were suggested and used by the LLM. (b) The retriever did not suggest the step send_slack_message and therefore the LLM used the common step send_notification instead. (c) The LLM should have used the TRY step as the parent to all the steps, but it did not fully understand the user query.