A Library of LLM Intrinsics for Retrieval-Augmented Generation

Marina Danilevsky, Kristjan Greenewald, Chulaka Gunasekara, Maeda Hanafi, Lihong He
Yannis Katsis, Krishnateja Killamsetty, Yatin Nandwani, Lucian Popa, Dinesh Raghu,
Frederick Reiss, Vraj Shah, Khoi-Nguyen Tran, Huaiyu Zhu, Luis Lastras
IBM Research

Abstract

In the developer community for large language models (LLMs), there is not yet a clean pattern analogous to a software library, to support very large scale collaboration. Even for the commonplace use case of Retrieval-Augmented Generation (RAG), it is not currently possible to write a RAG application against a well-defined set of APIs that are agreed upon by different LLM providers. Inspired by the idea of compiler intrinsics, we propose some elements of such a concept through introducing a library of *LLM Intrinsics* for RAG. An LLM intrinsic is defined as a capability that can be invoked through a well-defined API that is reasonably stable and independent of how the LLM intrinsic itself is implemented. The intrinsics in our library are released as LoRA adapters on HuggingFace, and through a software interface with clear structured input/output characteristics on top of vLLM as an inference platform, accompanied in both places with documentation and code. This article describes the intended usage, training details, and evaluations for each intrinsic, as well as compositions of multiple intrinsics.

1 Introduction

One of the most important software design patterns is the concept of a software library: generally reusable code with a well documented interface that enables very large scale collaboration between developers with different expertise. In large language models (LLMs), no such equivalent such pattern appears to have emerged as of yet. For example, prompt libraries tend to be useful only for a specific model. Even for the commonplace use case of Retrieval-Augmented Generation (RAG), it is not currently possible to write a RAG application against a well-defined set of APIs that are agreed upon by different LLM providers. Analogies to previous groundbreaking technologies abound; for example, different instruction set architectures used to be commonplace in microprocessor design, making code incompatible across such processors, and different operative systems offered different abstractions for applications that wanted to use system resources.

History suggests that the emergence of interfaces at key parts of system design are inevitable, to allow different specializations to flourish and support the creation of more complex systems. The purpose of this article is to introduce the elements of a proposal in the context of RAG. We take inspiration from the idea of compiler intrinsics, which are functions that occur often enough to warrant inclusion in a programming language. The compiler is responsible for producing instructions that implement such functions in the specific computer architecture where software is expected to run, but it may take any leeway in optimizing such an implementation.

In a loosely analogous concept, we define an LLM intrinsic to be a capability that can be invoked through a well-defined API that is reasonably stable and independent of how the LLM intrinsic itself is implemented. Metrics of performance, including accuracy, latency and throughput, may vary significantly across such implementations. We believe that LLM intrinsics are best implemented as a combination of a model and a co-optimized software layer that offers a familiar interface to the model developer. This pattern is already partly being followed in the LLM community; for example, LLM models in Huggingface are commonly packaged with configuration files for tokenizers, which transform structured representations of inputs (e.g., tool descriptions, sequences of messages) to raw tokens that are passed as actual inputs to the LLM.

We present a library of RAG LLM intrinsics that are implemented both as LoRA adapters, and through a software interface with clear structured input/output characteristics on top of vLLM as an inference platform. For illustrative purposes, these intrinsics are implemented using IBM Granite language models, with extension to other model families possible in the future. We remark that nothing in the definition of an LLM intrinsic demands that it be built as an adapter; it could be implemented in a number of ways, including simply as part of the training data of the underlying model. This article is a sister article to Greenewald et al. (2025), which introduces the concept of activated LoRAs as a mechanism that can be used to implement LLM intrinsics in a highly inference-efficient way.

1.1 OVERVIEW OF THE RAG LLM INTRINSICS LIBRARY

The LLM Intrinsics RAG Library currently comprises five intrinsics, each of which expects as input a (single-turn or multi-turn) conversation between a user and an AI assistant. Three of the intrinsics also expect a set of grounding passages. The functionality of each intrinsic is described below, and Table 1 summarizes the inputs and outputs of each one.

Query Rewrite (QR). Given a conversation ending with a user query, QR will decontextualize that last user query by rewriting it (whenever necessary) into an equivalent version that is standalone and can be understood by itself. While this adapter is general purpose for any multi-turn conversation, it is especially effective in RAG settings where its ability to rewrite a user query into a standalone version directly improves the retriever performance, which in turn improves the answer generation performance. This is a *pre-generation* intrinsic since its suggested use is before invoking retrieval.

Uncertainty Quantification (UQ). Given a conversation ending with an assistant response, UQ calculates a certainty percentage to reflect how certain it is about the answer generated to the previous user query. UQ can also take as input a conversation ending with an user query and predicting the certainty score based solely on the query, prior to generating an answer. UQ is also calibrated on document-based question answering datasets, and hence it can be applied to giving certainty scores for RAG responses created using grounding passages. This intrinsic could be used in a *post-generation* or *pre-generation* step.

Hallucination Detection (HD). Given a conversation ending with an assistant response, and a set of passages, HD outputs a hallucination risk range for each sentence in the last assistant response, with respect to the set of passages. It could be used in concert with sampling techniques that yield multiple generated responses, some of which could then be filtered according to their HD scores. This is a *post-generation* intrinsic since its expected use is after invoking the LLM to create the response.

Answerability Determination (AD). Given a conversation ending with a user query, and a set of passages, AD classifies whether that final user query is answerable or unanswerable based on the available information in the passages. It is valuable for restraining over-eager models by identifying unanswerable queries and prevent the generation of hallucinated responses. It can also be used to indicate that the system should re-query the retriever with alternate formulations, to fetch more relevant passages. This is a *pre-generation* intrinsic.

Citation Generation (CG). Given a conversation ending with an assistant response, and a set of passages, CG generates citations for that last assistant response from the provided passages. Citations are generated for each sentence in the response (when available), where each citation consists of a set of sentences from the supporting passages. This is a post-generation intrinsic since its expected use is after invoking the LLM, and therefore can be used to create citations for responses generated by any model.

1.2 RAG LLM INTRINSICS IMPLEMENTATION

Each of these intrinsics has been implemented by training a LoRA adapter for ibm-granite/granite-3.2-8b-instruction fine-tuned for a particular task. Each of these LoRA models has been released on HuggingFace:

- QR: `https://huggingface.co/ibm-granite/granite-3.2-8b-lora-rag-query-rewrite`
- UQ: `https://huggingface.co/ibm-granite/granite-3.2-8b-lora-uncertainty`
- HD: `https://huggingface.co/ibm-granite/granite-3.2-8b-lora-rag-hallucination-detection`
- AD: `https://huggingface.co/ibm-granite/granite-3.2-8b-lora-rag-answerability-prediction`
- CG: `https://huggingface.co/ibm-granite/granite-3.2-8b-lora-rag-citation-generation`

However, the recommended use is via a second release mechanism: through `Granite IO Processing`,[1] a framework which enables transforming how a user calls or infers an IBM Granite model and how the output

[1] Granite IO can be found at: `https://github.com/ibm-granite/granite-io`

Intrinsic	Input: Passages	Input: End Query	Input: End Resp.	Output	Pre/Post Gen
Query Rewrite (QR)		×		Standalone version of last query	pre
Uncertainty Quantification (UQ)	× (optional)	×		Certainty score for last assistant response (before generation)	pre
	× (optional)		×	Certainty score for last assistant response (after generation)	post
Hallucination Detection (HD)	×		×	Hallucination score for last assistant response	post
Answerability Determination (AD)	×	×		Flag denoting if last query is answerable from passages	pre
Citation Generation (CG)	×		×	Citations for last assistant response based on passages	post

Table 1: RAG LLM Intrinsics with their expected inputs and outputs. *End Query* and *End Resp.* refer to conversations ending with a user query and assistant response, respectively. *Pre/Post Gen* denotes if an intrinsic is called before or after generation, respectively.

from the model is returned to the user. In other words, the framework allows extended functionality of calling the model. This is particularly valuable as the downstream use of intrinsics relies on correctly structured output. Although we have made the individual LoRAs available, we strongly suggest that everyone uses the implementations in `Granite IO` and we have made example notebooks available.

In the rest of this paper we describe the specific implementation of each intrinsic in the library and evaluate their performance. We also discuss composing multiple intrinsics, and present particular implementations of composite flows accompanied by evaluations.

2 Query Rewrite

Granite 3.2 8b Instruct - Query Rewrite is a LoRA adapter for ibm-granite/granite-3.2-8b-instruct fine-tuned for the following task:

> Given a multi-turn conversation between a user and an AI assistant, decontextualize the last user utterance (query) by rewriting it (whenever necessary) into an equivalent version that is standalone and can be understood by itself.

2.1 Intended Use

The query rewrite adapter is generally applicable for multi-turn conversational use cases. It is particularly useful in RAG settings where its ability to rewrite a user query into a standalone version directly improves the retriever performance, which in turn improves the answer generation performance.

The rewrite is typically an expansion that in-lines, into the query, any implicit references that are made to entities, concepts, or even parts of the conversation that occur in the previous turns (either by the user or the AI assistant). Such expansion can include coreference resolution (i.e., replacement of pronouns with the actual entities), handling of ellipsis, which is the common linguistic phenomenon where parts of a sentence or phrase are omitted by the user, but can be understood from the context (i.e., for whom, of what, with respect to something discussed above, etc.).

As a result of the expansion, the query becomes a standalone query, still equivalent in meaning with what the user asked in the last turn. The rewritten query can be sent to downstream tasks (e.g., to a retriever in a RAG setting) as a better replacement for the original user query, and without the need for (a potentially very long) context.

Input: The input to the model is a list of conversational turns converted to a string using `apply_chat_template` function. These turns can alternate between the user and assistant roles, and the last turn is assumed to be from the user.

To prompt the LoRA adapter to rewrite the last user turn, a special rewrite role is used to trigger the rewrite capability of the model. The role includes the keyword "rewrite" followed by a short description of the query rewrite task.

```
<|start_of_role|>rewrite: Reword the final utterance from the USER into a single
     utterance that doesn’t need the prior conversation history to understand
    the user’s intent. If the final utterance is a clear and standalone question
    , please DO NOT attempt to rewrite it, rather output the last utterance as
    is. Your output format should be in JSON: { \"rewritten_question\": <REWRITE
    > }"<|end_of_role|>
```

Output: When prompted with the above special rewrite role, the model generates a json object, which contains a field with the actual rewritten question.

Note: Even though one main application for query rewrite is in RAG settings, this LoRA adapter can be used to rewrite user questions for other conversational use cases (e.g., to access a database, or other APIs, or tools). As such, the adapter does not need any RAG documents (that may be present in the context, in a RAG setting) and uses only the dialog turns with what is being said between the user and assistant.

See Section A.1 for an example describing how to use the Query Rewrite intrinsic.

2.2 Evaluation

2.2.1 Evaluation of the retriever

We evaluate `Recall@k` on the MT-RAG benchmark Katsis et al. (2025), under various query rewrite strategies for the retriever. All retrieved passages are obtained using the Elser retriever with the same settings as in the above paper. In addition to the LoRA adapter, we include several other baselines, including no-rewrite (where we send the last user turn to the retriever as-is), Mixtral rewrites, as well as gold rewrites (human-created). We evaluate on three different testsets: a) full MT-RAG dataset (842 data points with last user turns); b) the non-standalone subset of MT-RAG dataset, which is a subset of 260 (out of 842) last user turns that were annotated by humans as non-standalone (i.e., they are dependent on the prior context); c) the standalone subset of MT-RAG dataset, which is the complementary subset, with all the last user turns that were annotated by humans as standalone.

Retrieval recall evaluation (Recall@k) with different query rewrite strategies, evaluated on full, non-standalone and standalone subsets of MT-RAG dataset are shown in Tables 2, 3, and 4 respectively.

Rewrite Strategy	Recall@5	Recall@10	Recall@20
No rewrite	0.49	0.59	0.67
Mixtral 8x7b	0.52	0.64	0.72
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.56	0.68	0.76
Gold rewrite	0.56	0.67	0.75

Table 2: Comparison of query rewrite strategies on the retrieval task of full MT-RAG dataset

Rewrite Strategy	Recall@5	Recall@10	Recall@20
No rewrite	0.26	0.39	0.44
Mixtral 8x7b	0.36	0.49	0.57
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.44	0.57	0.66
Gold rewrite	0.48	0.58	0.66

Table 3: Comparison of query rewrite strategies on the retrieval task of non-standalone subset of MT-RAG

If we focus on Recall@20 numbers, as one instance of the metric, there is an overall 9 percentage points jump when using query rewrite with the Granite 3.2-8b LoRA adapter versus when using the no rewrite strategy.

Rewrite Strategy	Recall@5	Recall@10	Recall@20
No rewrite	0.61	0.72	0.79
Mixtral 8x7b	0.61	0.73	0.81
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.63	0.75	0.83
Gold rewrite	0.61	0.72	0.79

Table 4: Comparison of query rewrite strategies on the retrieval task of standalone subset of MT-RAG

This jump is more pronounced on the non-standalone fragment, where query rewrite with the Granite 3.2-8b LoRA adapter leads to 22 percentage points improvement over the no-rewrite strategy. Also, we can observe that the numbers with the LoRA rewrites are very close to what can be obtained with the gold rewrites on non-standalones (and slightly better on standalones for LoRA – human annotators were instructed to leave the query unchanged when classifying it as standalone, however, the LoRA adapter may still perform some rewriting which turns out to further improve the recall).

2.2.2 Evaluation of answer generation

We evaluate answer generation quality, with top-k passages retrieved under the various query rewrite strategies for the retriever. We choose here k = 20, but similar trends take place for other values of k. We used Granite-3.2-8b instruct as the answer generator, and RAGAS Faithfulness (RAGAS-F) and RAD-Bench score as metrics for answer quality. We use the same three testsets as above.

The answer quality evaluation using RAGAS-F and RAD-Bench on full, non-standalone and standalone subsets of MT-RAG dataset are shown in Tables 5, 6, and 7 respectively.

Rewrite Strategy	RAGAS-F	RAD-Bench
No rewrite	0.73	0.66
Mixtral 8x7b	0.80	0.68
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.81	0.70
Gold rewrite	0.79	0.69

Table 5: Comparison of query rewrite strategies on the answer quality on full MT-RAG dataset

Rewrite Strategy	RAGAS-F	RAD-Bench
No rewrite	0.61	0.62
Mixtral 8x7b	0.76	0.65
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.79	0.69
Gold rewrite	0.80	0.69

Table 6: Comparison of query rewrite strategies on the answer quality on non-standalone subset of MT-RAG

Rewrite Strategy	RAGAS-F	RAD-Bench
No rewrite	0.79	0.68
Mixtral 8x7b	0.82	0.70
Granite 3.2-8b-instruct-query-rewrite-LoRA	0.83	0.71
Gold rewrite	0.79	0.69

Table 7: Comparison of query rewrite strategies on the answer quality on standalone subset of MT-RAG

As with Recall, similar observations can be made here as well. Specifically, we see an 8 percentage points jump in RAGAS Faithfulness and 4 percentage points jump in RAD-Bench score when using query rewrite with the Granite 3.2-8b LoRA adapter versus when using the no rewrite strategy. This improvement is more pronounced on the non-standalone fragment, where query rewrite with the Granite 3.2-8b LoRA adapter leads to a 18 percentage points jump in RAGAS Faithfulness and 7 percentage points jump in RAD-Bench score.

2.3 Training Details

The training data contains both: 1) standalone examples, which teach the adapter to refrain from rewriting user questions that are already standalone, and 2) non-standalone examples containing a diversity of patterns that are used to teach the adapter to expand the user turn so that it becomes standalone.

The training data uses the publicly available Cloud corpus of technical documentation pages from MT-RAG.[2] Based on this corpus of documents, we constructed a dataset consisting of high-quality, human-created conversations, where the last turn of the conversation comes into versions: non-standalone version, and corresponding standalone version. The training dataset is proprietary and was obtained in combination with a third-party company who contracted the human annotators.

The LoRA adapter was fine-tuned using PEFT under the following regime: rank = 32, learning rate = $3e-6$, number of epochs = 25, with early stopping based on validation set, and $90/10$ split between training and validation.

3 Uncertainty Quantification

Granite 3.2 8b Instruct - Uncertainty Quantification is a LoRA adapter for ibm-granite/granite-3.2-8b-instruct, adding the capability to provide calibrated certainty scores when answering questions when prompted, in addition to retaining the full abilities of the ibm-granite/granite-3.2-8b-instruct model. The model is a LoRA adapter finetuned to provide certainty scores mimicking the output of a calibrator trained via the method in Shen et al. (2024).

3.1 Intended Use

Certainty score definition. The model will respond with a certainty percentage, quantized to 10 possible values (i.e. 5%, 15%, 25%,...95%). This percentage is calibrated in the following sense: given a set of answers assigned a certainty score of X%, approximately X% of these answers should be correct. See the eval experiment below for out-of-distribution verification of this behavior.

Certainty score interpretation. Certainty scores calibrated as defined above may at times seem biased towards moderate certainty scores for the following reasons. Firstly, as humans we tend to be overconfident in our evaluation of what we know and don't know - in contrast, a calibrated model is less likely to output very high or very low confidence scores, as these imply certainty of correctness or incorrectness. Examples where you might see very low confidence scores might be on answers where the model's response was something to the effect of "I don't know", which is easy to evaluate as not being the correct answer to the question (though it is the appropriate one). Secondly, remember that the model is evaluating itself - correctness/incorrectness that may be obvious to us or to larger models may be less obvious to an 8b model. Finally, teaching a model every fact it knows and doesn't know is not possible, hence it must generalize to questions of wildly varying difficulty (some of which may be trick questions!) and to settings where it has not had its outputs judged. Intuitively, it does this by extrapolating based on related questions it has been evaluated on in training - this is an inherently inexact process and leads to some hedging.

Important note: Certainty is inherently an intrinsic property of a model and its abilities. Granite-3.2-8b-Uncertainty-Quantification is not intended to predict the certainty of responses generated by any other models besides itself or ibm-granite/granite-3.2-8b-instruct. Additionally, certainty scores are distributional quantities, and so will do well on realistic questions in aggregate, but in principle may have surprising scores on individual red-teamed examples.

3.1.1 Usage steps

There are two supported usage scenarios.

Scenario 1. Answering a question and obtaining a certainty score proceeds as follows. Given a user query written in the user role:

1. Use the base model to generate a response as normal (via the assistant role).

[2] https://github.com/IBM/mt-rag-benchmark

2. Prompt the model to generate a certainty score by generating in the certainty role (use "certainty" as the role in the chat template, or simply append `<|start_of_role|>certainty<|end_of_role|>` and continue generating).
3. The model will respond with a certainty percentage, quantized with steps of 10% (i.e. 05%, 15%, 25%,...95%). Note, any additional text after the score and % can be ignored. You can curb additional generation by setting "max token length" = 3 when using this role.

Scenario 2. Predicting the certainty score from the question (optionally plus documents) only, prior to generating an answer. Given a user query written in the user role:

1. Prompt the model to generate a certainty score by generating in the certainty role (use "certainty" as the role in the chat template, or simply append `<|start_of_role|>certainty<|end_of_role|>` and continue generating).
2. The model will respond with a certainty percentage, quantized with steps of 10% (i.e. 05%, 15%, 25%,...95%). Note, any additional text after the score and % can be ignored. You can curb additional generation by setting "max token length" = 3 when using this role.
3. Remove the generated certainty string, and if desired, use the base model to generate a response as normal (via the assistant role).

See Section A.2 for an example describing how to use the Uncertainty Quantification intrinsic to answer questions and obtain intrinsic calibrated certainty scores.

3.1.2 Possible downstream use cases (not implemented)

- Human usage: Certainty scores give human users an indication of when to trust answers from the model (which should be augmented by their own knowledge).
- Model routing/guards: If the model has low certainty (below a chosen threshold), it may be worth sending the request to a larger, more capable model or simply choosing not to show the response to the user.
- RAG: Granite-3.2-8b-Uncertainty-Quantification is calibrated on document-based question answering datasets, hence it can be applied to giving certainty scores for answers created using RAG. This certainty will be a prediction of overall correctness based on both the documents given and the model's own knowledge (e.g. if the model is correct but the answer is not in the documents, the certainty can still be high).

3.2 Evaluation

The model was evaluated on the MMLU[3] datasets (not used in training). Shown are the Expected Calibration Error (ECE)[4] for each task, for the base model (Granite-3.2-8b-instruct) and Granite-3.2-8b-Uncertainty-Quantification. The average ECE across tasks for our method is 0.064 (out of 1) and is consistently low across tasks (maximum task ECE 0.10), compared to the base model average ECE of 0.20 and maximum task ECE of 0.60. Note that our ECE of 0.064 is smaller than the gap between the quantized certainty outputs (10% quantization steps). Additionally, the zero-shot performance on the MMLU tasks does not degrade, averaging at 89%.

3.3 Training Details

The model is a LoRA adapter finetuned to provide certainty scores mimicking the output of a calibrator trained via the method in Shen et al. (2024).

The following datasets were used for calibration and/or finetuning:

- BigBench (`https://huggingface.co/datasets/tasksource/bigbench`)
- MRQA (`https://huggingface.co/datasets/mrqa-workshop/mrqa`)

[3]https://huggingface.co/datasets/cais/mmlu

[4]https://towardsdatascience.com/expected-calibration-error-ece-a-step-by-step-visual-explanation-with-python-code-c3e9aa12937d

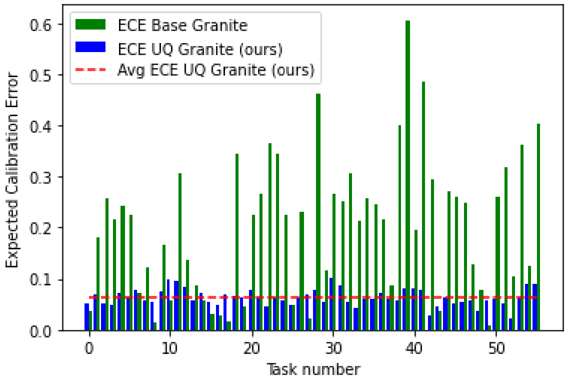

Figure 1: Evaluation of UQ Intrinsic

- newsqa (`https://huggingface.co/datasets/lucadiliello/newsqa`)
- trivia_qa (`https://huggingface.co/datasets/mandarjoshi/trivia_qa`)
- search_qa (`https://huggingface.co/datasets/lucadiliello/searchqa`)
- openbookqa (`https://huggingface.co/datasets/allenai/openbookqa`)
- web_questions (`https://huggingface.co/datasets/Stanford/web_questions`)
- smiles-qa (`https://huggingface.co/datasets/alxfgh/ChEMBL_Drug_Instruction_Tuning`)
- orca-math (`https://huggingface.co/datasets/microsoft/orca-math-word-problems-200k`)
- ARC-Easy (`https://huggingface.co/datasets/allenai/ai2_arc`)
- commonsense_qa (`https://huggingface.co/datasets/tau/commonsense_qa`)
- social_iq_a (`https://huggingface.co/datasets/allenai/social_i_qa`)
- super_glue (`https://huggingface.co/datasets/aps/super_glue`)
- figqa (`https://huggingface.co/datasets/nightingal3/fig-qa`)
- riddle_sense (`https://huggingface.co/datasets/INK-USC/riddle_sense`)
- ag_news (`https://huggingface.co/datasets/fancyzhx/ag_news`)
- medmcqa (`https://huggingface.co/datasets/openlifescienceai/medmcqa`)
- dream (`https://huggingface.co/datasets/dataset-org/dream`)
- codah (`https://huggingface.co/datasets/jaredfern/codah`)
- piqa (`https://huggingface.co/datasets/ybisk/piqa`)

4 Hallucination Detection

Granite 3.2 8b Instruct - Hallucination Detection is a LoRA adapter for ibm-granite/granite-3.2-8b-instruct fine-tuned for the hallucination detection task of model outputs. Given a multi-turn conversation between a user and an AI assistant ending with an assistant response and a set of documents/passages on which the last assistant response is supposed to be based, the adapter outputs a hallucination risk range for each sentence in the assistant response.

4.1 Intended Use

This is a LoRA adapter that gives the ability to identify hallucination risks for the sentences in the last assistant response in a multi-turn RAG conversation based on a set of provided documents/passages.

While you can invoke the LoRA adapter directly, we highly recommend calling it through `Granite IO`, as described in Section 1.2. `Granite IO` wraps the hallucination detection adapter with a tailored I/O processor. The I/O processor provides a friendlier development interface, as it takes care of various data transformations and validation tasks. This includes splitting the assistant response into sentences before calling the adapter, as well as validating the adapter's output and transforming the sentence IDs returned by the adapter into appropriate spans over the response.

However, if you prefer to invoke the LoRA adapter directly, its expected input/output is described below.

Model input: The input to the model is conceptually a list of conversational turns ending with an assistant response and a list documents converted to a string using `apply_chat_template` function. For the adapter to work, the last assistant response should be pre-split into sentences and sentence indices need to be prepended. In more detail, the primary inputs are the following three items, each represented in JSON:

- **conversation**: A list of conversational turns between the user and the assistant, where each item in the list is a dictionary with fields `role` and `content`. The `role` equals to either `user` or `assistant`, denoting user and assistant turns, respectively, while the `content` field contains the corresponding user/assistant utterance. The conversation should end with an assistant turn and the `text` field of that turn should contain the assistant utterance with each sentence prefixed with a response sentence id of the form `<rI>`, where `I` is an integer. The numbering should start from 0 (for the first sentence) and be incremented by one for each subsequent sentence in the last assistant turn.
- **instruction**: A task instruction, which is encoded as a dictionary with fields `role` and `content`, where `role` equals to `system` and `content` equals to the following string describing the hallucination detection task: "Split the last assistant response into individual sentences. For each sentence in the last assistant response, identify the faithfulness score range. Ensure that your output includes all response sentence IDs, and for each response sentence ID, provide the corresponding faithfulness score range. The output must be a json structure."
- **documents**: A list of documents, where each item in the list is a dictionary with fields `doc_id` and `text`. The `text` field contains the text of the corresponding document.

To prompt the LoRA adapter, we combine the above components as follows: We first append the `instruction` to the end of the `conversation` to generate an `augmented_conversation` list. Then we invoke the `apply_chat_template` function with parameters: `conversation = augmented_conversation` and `documents = documents`.

Model output: When prompted with the above input, the model generates a range for faithfulness score (hallucination risk) for each sentence of the last assistant response in the form of a JSON dictionary. The dictionary is of the form `{"<r0>": "value_0", "<r1>": "value_1", ...}`, where each field `<rI>`, where `I` an integer, corresponds to the ID of a sentence in the last assistant response and its corresponding value is the range for faithfulness score (hallucination risk) of the sentence. The output values can show numeric ranges between 0-1 with increments of 0.1, where the higher values correponds to high faithfulness (low hallucination risk), and lower values corresponds to low faithfulness (high hallucination risk). Additionally, the model is trained to output unanswerable when the response sentence indicate that the question is not answerable, and to output NA when the faithfulness cannot be determined (ex: very short sentences).

See Section A.3 for an example describing how to use the Hallucination Detection intrinsic.

4.2 Evaluation

The LoRA adapter was evaluated on the QA portion of the RAGTruth benchmark Niu et al. (2024). We compare the response-level hallucination detection performance between the LoRA adapter and the methods reported in the RAGTruth paper. The responses that obtain a faithfulness score less than 0.1 for at least one sentence are considered as hallucinated responses.

The evaluation results are shown in the Table 8. The results for the baselines are extracted from the RAGTruth paper Niu et al. (2024).

4.3 Training Details

The process of generating the training data consisted of two main steps:

- **Multi-turn RAG conversation generation:** Starting from publicly available document corpora, we generated a set of multi-turn RAG data, consisting of multi-turn conversations grounded on passages

Model	Precision	Recall	F1
gpt-3.5-turbo (prompted)	18.8	84.4	30.8
gpt-4-turbo (prompted)	33.2	90.6	45.6
SelfCheckGPT Manakul et al. (2023)	35.0	58	43.7
LMvLM Cohen et al. (2023)	18.7	76.9	30.1
Finetuned Llama-2-13B	61.6	76.3	68.2
Hallucination Detection LoRA	67.6	77.4	72.2

Table 8: Hallucination detection results

retrieved from the corpora. For details on the RAG conversation generation process please refer to the Granite Technical Report[5] as well as Lee et al. (2024).

- **Faithfulness label generation:** For creating the faithfulness labels for responses, we used the NLI-based technique available at Achintalwar et al. (2024).

The following public datasets were used as seed datasets for the multi-turn RAG conversation generation process:

- CoQA Wikipedia Passages (`https://stanfordnlp.github.io/coqa/`)
- MultiDoc2Dial (`https://huggingface.co/datasets/IBM/multidoc2dial`)
- QuAC (`https://huggingface.co/datasets/allenai/quac`)

The LoRA adapter was fine-tuned using PEFT under the following regime: rank = 8, learning rate = 1e-5, and 90/10 split between training and validation.

5 Answerability Determination

Granite 3.2 8b Instruct - Answerability Determination is a LoRA adapter for ibm-granite/granite-3.2-8b-instruct fine-tuned for binary answerability classification task. The model takes as input a multi-turn conversation and a set of documents, and classifies whether the user's final query is answerable or unanswerable based on the available information in the set of input documents.

5.1 Intended Use

This is a LoRA adapter that enables answerability classification for the final user query in a multi-turn conversation, with respect to a set of provided documents. The model is trained to determine whether the last user query is answerable or unanswerable, based solely on the information present in the input documents. This makes it suitable for applications involving RAG and document-grounded chatbots, where knowing whether sufficient information exists to answer a query is crucial. The classification output from the answerability model can be used in several downstream applications, including but not limited to:

- Filter out unanswerable questions before sending them to generation in RAG setting. By classifying a query as unanswerable upfront, the system can prevent hallucinated or misleading responses.
- Re-query the retriever to get more relevant documents. If a query is initially deemed unanswerable, the retriever can be re-invoked with alternate formulations to fetch more relevant documents.

Model input: The input to the model is a list of conversational turns and a list of documents converted to a string using `apply_chat_template` function. These turns can alternate between the user and assistant roles. The last turn is from the user. The list of documents is a dictionary with text field, which contains the text of the corresponding document.

To prompt the LoRA adapter to determine answerability, a special answerability role is used to trigger this capability of the model. The role includes the keyword `"answerability": <|start_of_role|>answerability<|end_of_role|>`

[5]https://github.com/ibm-granite/granite-3.0-language-models/blob/main/paper.pdf

Model	Unans. Precision	Unans. Recall	Unans. F1	Ans. Precision	Ans. Recall	Ans. F1	Weighted F1
BigBird w/ MLP	49.2	68.5	57.3	48.0	29.2	36.3	46.8
LLaMA 2-7B	72.2	71.0	71.6	71.4	72.6	72.0	71.8
Granite 3.2-8b LoRA	84.2	68.0	75.2	73.1	87.2	79.5	77.4

Table 9: Comparison of classification performance across models on SQUADRUN Dev set. Metrics are broken down by class (Answerable vs. Unanswerable) and include precision, recall, and F1 score.

Model	Unans. Precision	Unans. Recall	Unans. F1	Ans. Precision	Ans. Recall	Ans. F1	Weighted F1
BigBird w/ MLP	69.6	77.6	73.4	70.1	60.8	65.2	69.6
LLaMA 2-7B	86.9	89.4	88.2	87.3	84.5	85.9	87.1
Granite 3.2-8b LoRA	85.4	89.3	87.3	87.0	82.4	84.6	86.1

Table 10: Comparison of classification performance across models on MT-RAG Benchmark. Metrics are broken down by class (Answerable vs. Unanswerable) and include precision, recall, and F1 score.

Model output: When prompted with the above input, the model generates the answerable or unanswerable output.

See Section A.4 for an example describing how to use the Answerability Determination intrinsic.

5.2 Evaluation

5.2.1 Answerability Classification

We evaluated the model against baselines on binary answerability classification using two separate benchmarks:

- Single-turn Setting (SQUADRun Benchmark Rajpurkar et al. (2018)): In this setting, the user query and the supporting documents are provided. Our model was evaluated against standard baselines to measure its ability to determine whether a standalone question is answerable based on the document set. Table 9 shows the classification results.
- Multi-turn Setting (MT-RAG Benchmark Katsis et al. (2025)): In this setting, the model is given the full multi-turn conversation history along with the supporting documents. This benchmark evaluates the model's ability to assess answerability when the final user query can also depend on prior turns for context. Table 10 shows the results.

5.2.2 Comparing LoRA Adapter vs. Vanilla Granite for Answer Quality

We compare the performance of Granite 3.2-8b Instruct vs. Granite 3.2-8b LoRA adapter on a subset of MT-RAG Benchmark in Table 11. In this setup, each query is paired with only 5 retrieved passages as context. The true answerability label for each query indicates whether the query is answerable with respect to the retrieved context.

- Answerability Classification Performance: The LoRA adapter outperforms the vanilla model in overall F1 on both answerables and unanswerables. The LoRA adapter achieves higher recall on unanswerable queries, making it better at identifying questions that should not be answered. However, this comes at the cost of lower recall on answerable queries.
- The RAGAS Faithfulness (`RF`) score (on truly answerable queries): This drops slightly with the LoRA adapter. However, this is not due to degraded generation quality, but rather because the model labels more truly answerable queries as unanswerable and abstains from answering.
- Joint Answerability-Faithfulness Score (`JAFS`):

$$\texttt{JAFS} = \begin{cases} 1 & \text{if prediction = IDK/unanswerable \& truth = unanswerable} \\ \texttt{RF} & \text{if prediction = non-IDK/answerable \& truth = answerable} \\ 0 & \text{otherwise} \end{cases}$$

This score rewards the model for correctly abstaining on unanswerable queries (full credit) and for providing faithful answers on answerable queries (partial credit based on RAGAS Faithfulness). No credit is given for incorrect or unfaithful predictions.

The LoRA adapter achieves a 7% lift on this metric - rewarding the model for correctly abstaining on unanswerable queries and for being faithful when it chooses to answer.

Model	Unans. F1	Ans. F1	Unans. Recall	Ans. Recall	RF (on Truly Answerable)	JAFS
Granite 3.2-8b Instruct	14	76	8	97	75	50
Granite 3.2-8b LoRA	47	77	37	88	70	57

Table 11: Comparison of Granite 3.2-8B Instruct vs. LoRA Adapter on Answerability and Faithfulness metrics using MT-RAG Benchmark.

5.3 Training Details

The training data uses the publicly available Government corpus from MT-RAGKatsis et al. (2025) as the source of documents. Based on this corpus, we constructed a dataset consisting of a mix of human-created and synthetically generated multi-turn conversations. It includes two types of examples: (1) Answerable queries, where the final user question can be answered based on the provided documents. These examples teach the adapter to recognize when sufficient information is present to support an answer. (2) Unanswerable queries, where the documents lack the necessary information to answer the final user query. We used Mixtral as an automatic judge to validate the answerability labels and filter out noisy samples.

The LoRA adapter was fine-tuned using PEFT under the following regime: rank = 32, learning rate = 5e-6, number of epochs = 25, with early stopping based on validation set, and 90/10 split between training and validation.

6 Citation Generation

Granite 3.2 8b Instruct - Citation Generation is a RAG-specific LoRA adapter for ibm-granite/granite-3.2-8b-instruct fine-tuned for the citation generation task. Given a multi-turn conversation between a user and an AI assistant ending with an assistant response and a set of documents/passages on which the last assistant response is supposed to be based, the adapter generates citations for the last assistant response from the provided documents/passages. The LoRA adapter has the following features:

- **Fine-grained citations:** The adapter generates citations for each sentence in the assistant response (when available). Moreover, each citation consists of a set of sentences from the documents/passages that support the corresponding sentence in the assistant response.
- **Post-hoc citation generation:** Since the adapter takes the assistant response as input, it can generate citations for responses generated by any LLM. Pick your favorite LLM and use the adapter to generate post-hoc citations!

6.1 Intended Use

This is a LoRA adapter that gives the ability to generate citations for the last assistant response in a multi-turn RAG conversation based on a set of provided documents/passages. It can be used to generate post-hoc citations for assistant responses generated by any LLM in a RAG setting.

While you can invoke the LoRA adapter directly, we highly recommend calling it through `Granite IO`, as described in Section 1.2. `Granite IO` wraps the adapter with a tailored I/O processor. The I/O processor provides a friendlier development interface, as it takes care of various data transformations and validation tasks. This includes, among others, splitting the input documents and assistant response into sentences before calling the adapter, as well as validating the adapter's output and transforming the sentence IDs returned by the adapter into appropriate spans over the documents and the response.

However, if you prefer to invoke the LoRA adapter directly, the expected input/output is described below.

Model input: The input to the model is conceptually a list of conversational turns ending with an assistant response and a list of documents converted to a string using the `apply_chat_template` function. For the adapter to work, the last assistant response as well as the documents should be pre-split into sentences. In more detail, the primary inputs are the following three items, each represented in JSON:

- **conversation**: A list of conversational turns between the user and the assistant, where each item in the list is a dictionary with fields `role` and `content`. The `role` equals to either `user` or `assistant`, denoting user and assistant turns, respectively, while the `content` field contains the corresponding user/assistant utterance. The conversation should end with an assistant turn and the `text` field of that turn should contain the assistant utterance with each sentence prefixed with a response sentence ID of the form `<rI>`, where `I` is an integer. The numbering should start from 0 (for the first sentence) and be incremented by one for each subsequent sentence in the last assistant turn. Note that only the last assistant turn should be split into sentences as described above; earlier assistant turns (as well as all user turns) should be maintained in their original form.
- **instruction**: A task instruction, which is encoded as a dictionary with fields `role` and `content`, where `role` equals to `system` and `content` equals to the following string describing the citation generation task: "Split the last assistant response into individual sentences. For each sentence in the response, identify the statement IDs from the documents that it references. Ensure that your output includes all response sentence IDs, and for each response sentence ID, provide the corresponding referring document sentence IDs."
- **documents**: A list of documents, where each item in the list is a dictionary with fields `doc_id` and `text`. The `text` field contains the text of the corresponding document with each sentence prefixed with a context sentence ID of the form `<cI>`, where `I` is an integer. The context sentence ID numbers should start from 0 (for the first sentence of the first document) and be incremented by one for each subsequent sentence. The numbers should continue to be incremented across documents to ensure that each context sentence ID appears once across the entire list of documents. For instance, if the last sentence of the 1st document has context sentence ID `<cn>`, then the first sentence of the 2nd document is expected to have ID `<cn+1>`.

To prompt the LoRA adapter, we combine the above components as follows: We first append the `instruction` to the end of the `conversation` to generate an `augmented_conversation` list. Then we invoke the `apply_chat_template` function with parameters: `conversation = augmented_conversation` and `documents = documents`.

Model output: When prompted with the above input, the model generates the citations for each sentence of the last assistant response in the form of a JSON dictionary. The dictionary is of the form `{"<r0>": ..., "<r1>": ..., ...}`, where each field `<rI>`, with `I` an integer, corresponds to the ID of the corresponding sentence in the last assistant response and its value is a list of context sentence IDs corresponding to the sentence(s) in the input documents that support the particular response sentence.

See Section A.5 for an example describing how to use the Citation Generation intrinsic to generate citations for a given assistant response.

6.2 EVALUATION

We evaluate the LoRA adapter on two citation benchmarks:

- **ALCE** Gao et al. (2023): Evaluates the ability of models to produce *document/passage-level* citations (i.e., identify the documents/passages that support a statement in the response).

- **LongBench-Cite** Zhang et al. (2024): Evaluates the ability of models to produce fine-grained *span-level* citations (i.e., identify the spans within the input documents/passages that support a statement in the response) with a focus on long contexts.

Since the LoRA adapter is a post-hoc citation generation approach, its performance on the two benchmarks depends on the assistant responses for which it is asked to generate citations. To facilitate an apples-to-apples comparison, for each experiment, we keep the assistant responses the same and change the model that is used to generate the citations. In particular, we prompt an LLM to create an assistant response together with citations and evaluate the generated citations on the corresponding benchmark. Then, we compute and evaluate the citations generated for the same LLM response by the LoRA adapter.

6.2.1 EVALUATION ON ALCE

For the ALCE evaluation, we prompt Llama-3.1-70B-Instruct and Mixtral-8x22B-Instruct to generate both the assistant response and corresponding passage-level citations. We first calculate the performance of the citations generated by these models on ALCE. Subsequently, we feed the responses of these models (leaving out the citations) to the LoRA adapter and evaluate its generated citations. The results are shown in Table 12.

Model generating response	Model generating citations	Recall	Precision	F1
Llama-3.1-70B-Instruct	Llama-3.1-70B-Instruct	61.4	58.1	59.7
Llama-3.1-70B-Instruct	Granite-3.2-8B LoRA citations	54.8	65.9	59.8
Mixtral-8x22B-Instruct	Mixtral-8x22B-Instruct	62.2	62.5	62.3
Mixtral-8x22B-Instruct	Granite-3.2-8B LoRA citations	54.3	69.5	61.0

Table 12: Citation generation evaluation on ALCE

We observe that the LoRA adapter performs on par with much bigger models when those are prompted to create passage-level citations. It is interesting to note that while the adapter's F1 performance is similar to the baselines, it exhibits a different precision-recall trade-off, trading lower recall for higher precision.

Notes:

- All results are reported on the ELI5 dataset using the ORACLE (5-psg) setting.
- To prompt Llama and Mixtral, we employ a setting similar to the one proposed in the ALCE paper; in particular we use a two-shot prompt comprised of two of the ICL examples from ALCE as well as a slightly modified version of the instruction from the paper Gao et al. (2023).
- Sentence splitting of context/response is performed using NLTK.
- Finally, since ALCE expects passage-level citations, we elevate the finer-grained citations produced by the LoRA adapter to the passage level before running the ALCE evaluation.

6.2.2 EVALUATION ON LONGBENCH-CITE

For the LonBench-Cite evaluation, we prompt Llama-3.1-70B-Instruct to generate both the assistant response and corresponding citations. Then we evaluate the citations generated by Llama as well as the post-hoc citations generated by the LoRA adapter when invoked on the Llama responses. The results are shown in Table 13.

Model generating response	Model generating citations	Longbench-Chat (en)			MultifieldQA (en)			HotpotQA			GovReport		
		R	P	F1	R	P	F1	R	P	F1	R	P	F1
Llama-3.1-70B-Instruct	Llama-3.1-70B-Instruct	27.0	34.4	26.1	46.1	63.3	49.7	34.0	39.4	30.2	55.0	77.5	62.0
Llama-3.1-70B-Instruct	Granite-3.2-8B LoRA citations	61.9	68.6	62.0	71.2	84.1	74.3	66.8	73.3	65.4	70.3	83.6	75.4

Table 13: Citation generation evaluation on LongBench-Cite

We observe that the LoRA adapter performs across the board significantly better than Llama-3.1-70B-Instruct when prompted to create span-level citations. This demonstrates the value of the adapter to create post-hoc citations even for assistant responses generated by much bigger LLMs.

Notes:

- The evaluation results are reported on the English subset of LongBench-Cite (i.e., restricted to instances whose `language` field equals to `en`).
- The results for the LoRA adapter do not include the performance for 4/585 tasks, which encountered out of memory errors.

- To prompt Llama to generate a response with citations, we use the one-shot prompt described in the LongBench-Cite paper Zhang et al. (2024).
- For the LoRA adapter, sentence splitting of the context is performed using NLTK. For the response, we reuse the splitting in Llama's output (since the LongBench-Cite prompt instructs the model to output a response split into sentences/statements).

6.3 Training Details

The LoRA adapter was trained on synthetically-generated citation datasets. The process of generating the training data consisted of two main steps:

- **Multi-turn RAG conversation generation:** Starting from publicly available document corpora, we generated a set of multi-turn RAG data, consisting of multi-turn conversations grounded on passages retrieved from the corpora. For details on the RAG conversation generation process please refer to the Granite Technical Report[6] as well as Lee et al. (2024).
- **Citation generation:** For each turn of the multi-turn RAG conversations from the previous step, we used a multi-step synthetic citation generation pipeline to generate citations for the assistant response.

The following public datasets were used as seed datasets for the multi-turn RAG conversation generation process:

- CoQA Wikipedia Passages (`https://stanfordnlp.github.io/coqa/`)
- MultiDoc2Dial (`https://huggingface.co/datasets/IBM/multidoc2dial`)
- QuAC (`https://huggingface.co/datasets/allenai/quac`)

Leveraging the generated training data, the LoRA adapter was fine-tuned using PEFT under the following regime: rank = 8, learning rate = 1e-5, and 90/10 split between training and validation.

7 Composite Intrinsics

Individual intrinsics are created and trained to focus on particular tasks. In reality, we would certainly like to simultaneously improve retriever performance, reduce hallucinations, produce more accurate citations, and so on. Since the intrinsics' implementations are abstracted, it is simple to add one or more to a "flow" for a particular application.

For example, since using Query Rewrite improves recall performance, it is also likely to positively impact Citations, by providing more relevant contexts from which citations can be drawn. Or, intrinsics such as Uncertainty Quantification or Hallucination Detection could be combined with a sampling approach to response generation (such a sampling approach is incidentally available through `Granite IO`) in order to easily filter out low quality candidates.

On the other hand, there are some composite flows that have a good chance of producing puzzling outcomes. For example, what might it mean if the same input yields a high score from Uncertainty Quantification (meaning, the model is quite certain about its answer) and yet low scores for Hallucination Detection (meaning, the model believes the answer to be mostly unfaithful)? Or, what if a query is unanswerable according to Answerability Determination, and yet a subsequently generated answer is richly cited by Citation Generation? With every additional intrinsic added to an application flow, the complexity of testing and interpreting the resulting behavior significantly increases. Therefore, although many combinations may be technically possible, we recommend caution, and spend the rest of this section going through the process of creating and evaluating a composite intrinsic flow.

In particular, we will consider a flow which uses both the Query Rewrite (QR) and Answerability Determination (AD) intrinsics. These intrinsics are beneficial when the conversation with a RAG system is expected to frequently be multi-turn, and it is important to limit responses to only those which can be successfully supported (many customer-facing chat agents would fall under this use case). Although on the surface it may seem like neither of these intrinsics would affect each other's performance, we will see that the truth is a little more complicated.

[6]https://github.com/ibm-granite/granite-3.0-language-models/blob/main/paper.pdf

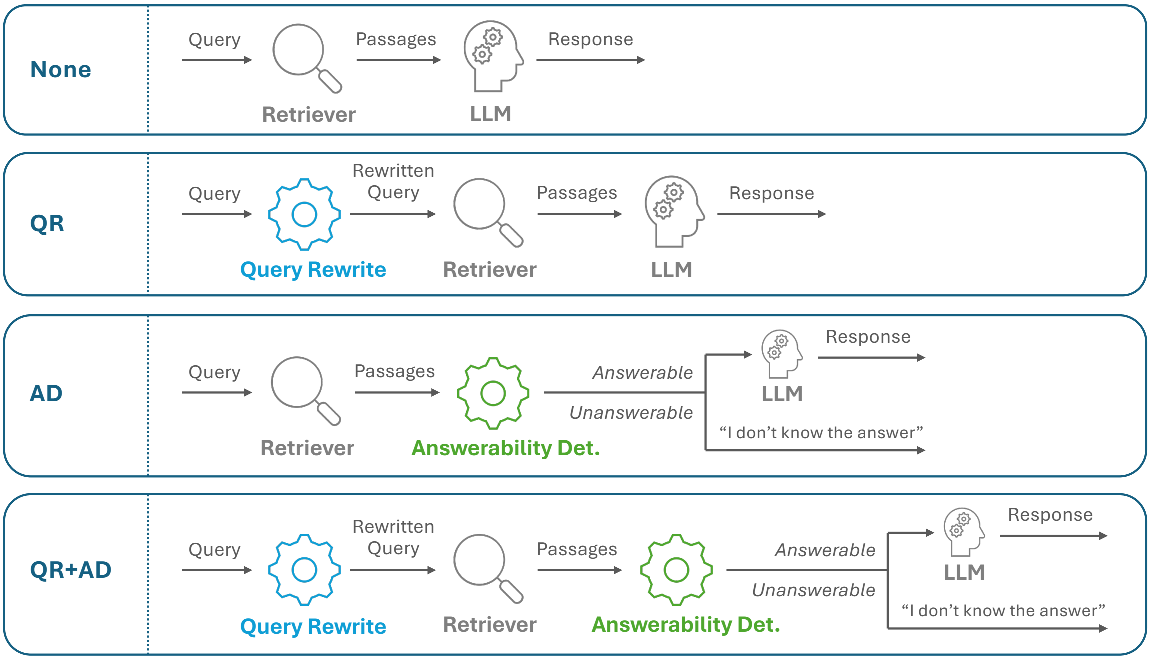

Figure 2: RAG flows considered in this work

7.1 Query Rewrite plus Answerability Determination

We briefly introduce four RAG flows which use these two intrinsics in various ways:[7]

- **None**: The given user query is used to retrieve the top k passages; both are input to the generator model to create the response.
- **Query Rewrite (QR)**: The given user query is transformed using the QR instrinsic. The resulting query is used to retrieve the top k passages. The original query and retrieved passages are input to the generator model to create the response.
- **Answerability Determination (AD)**: The given user query is used to retrieve the top k passages. The query and passages are input to the AD intrinsic. If AD returns *yes*, the query and passages are input to the generator model to create the response; if AD returns *no*, this step is skipped and a pre-determined response of "I don't know the answer" is output.
- **Query Rewrite and Answerability Determination (QR+AD)**: Both intrinsics are used: QR to affect which top k passages are retrieved and AD to determine whether to circumvent the generator model.

Figure 2 offers a visual representation of these four flows. We will now examine the benefits and tradeoffs to using both of these intrinsics in the four flows. As previously mentioned, the value of QR is in improving the performance of the retriever, increasing the relevance of the top k passages. This increases the ability to generate both a more faithful response and more accurate citations, since there is more relevant context provided. On the other hand, the value of AD is in restraining an overeager model when a grounded response is impossible, greatly increasing the likelihood of correctly refusing to answer (though at the cost of occasionally being too conservative). Therefore, we must look at the quantitative effect of the individual and composite intrinsic flows on a) correctly classifying the query as answerable or unanswerable; b) the faithfulness of those responses which the generative model creates; and c) the aggregate score of faithfulness weighted by answerability classification. We will take these one at a time.

Experimental Setup. To benchmark the above flows, we use MT-RAG conversations and Elser for retriever. QR and AD is performed with QR LoRA adapter and AD LoRA adapter with Granite 3.2-8B Instruct, respectively. We set the number of retrieved passages to 5 across different retrieval strategies. For generation,

[7]These are not the only possible applications of either intrinsic; rather these serve as reasonable examples that allow us to investigate the effects of composing them.

Flow	$F1_{Unanswerable}$	$F1_{Answerable}$
None	14	76
QR	12	82
AD	47	77
QR+AD	42	82

Table 14: Performance on the task of Answerability Classification

Flow	#Responses	RAGAS-F
None	505	75
QR	578	78
AD	455	70
QR+AD	530	73

Table 15: Faithfulness of generated responses (RAGAS-F) for queries determined to be answerable by each flow; the size of that set is denoted by #Responses (Number of Generated Responses)

we send the Granite 3.2-8B Instruct model the following information: the entire conversation, top-5 retrieved passages, and the model's default RAG instruction prompt.

7.1.1 Evaluation: Answerability Classification

Table 14 shows the $F1$ scores on the task of answerability classification of the 4 RAG flows described above. Using the AD intrinsic significantly improves performance on unanswerable queries ($F1$ score increases from 14 to 47). The QR intrinsic also has an effect (though it's much smaller): increasing the Recall@5 performance of the retriever makes some questions more likely to be answerable ($F1$ score increases from 76 to 82).

7.1.2 Evaluation: Answer Faithfulness

The QR intrinsic improves the faithfulness of the answer created by the generative model. Table 15 shows the number of responses in each of the 4 flows (meaning, the question was correctly identified as answerable and the generative model wrote a response) along with the RAGAS-F score (see Section 2) on those responses. To identify whether a flow considers a query to have been answerable, a simple "I don't know" judge is used on the final output response, which determined whether the response contains content, or is in essence equivalent to saying, "I don't know the answer" (see Katsis et al. (2025) for details on the IDK judge). It is important to note that this does not provide a comprehensive view, as it does not reflect the performance of the RAG system on the rest of the cases not captured by this table (thus the inclusion of the number of responses that was able to be scored in each flow).

7.1.3 Evaluation: Joint Answerability-Faithfulness

Considering the RAGAS-F score from Table 15 in isolation, it would appear that using the AD intrinsic harms performance, and that the best approach is to only make use of the QR intrinsic. Therefore it is clear that we should not only rely on this evaluation. Therefore, we return to the Joint Answerability-Faithfulness Score (JAFS), introduced in Section 5. This score rewards the model for correctly abstaining on unanswerable queries (full credit) and for providing faithful answers on answerable queries (partial credit based on RAGAS Faithfulness). No credit is given for responding to an unanswerable query, nor for refusing to respond to an answerable query. Table 16 presents the JAFS score for each of the four flows.

Flow	JAFS
None	50
QR	57
AD	57
QR+AD	**61**

Table 16: Joint Answerability-Faithfulness Score for each flow

By going through this careful evaluation process, we are able to understand the benefits and trade-offs of this composite flow. The important aspects were a) creating appropriate non-composite flows for comparison and b) analyzing the effect of each flow using metrics which reflect the value brought by each intrinsic. We have now been able to demonstrate that for the use case where the conversations with our RAG system are expected to frequently be multi-turn, and it is important to limit responses to only those which can be successfully supported, making use of both the QR and AD intrinsic in the flow as described will yield better overall performance.

8 Conclusion

In this paper we introduce a library of LLM intrinsics for RAG. The intrinsics currently implemented are Query Rewrite, Uncertainty Quantification, Hallucination Detection, Answerability Determination, and Citation Generation. They are released as LoRA adapters for `ibm-granite/granite-3.2-8b-instruct` on HuggingFace, as well as through the recommended implementations in `Granite IO`, accompanied in both places with documentation and code. All the models are publicly released under an Apache 2.0 license for both research and commercial use. We describe the intended usage, training details, and evaluations for each intrinsic. We also introduce the notion of Composite Intrinsics, and describe one particular composition in detail, including in-depth evaluation of the created flow.

9 Acknowledgments

Thanks to internal and external annotators.

References

Swapnaja Achintalwar, Adriana Alvarado Garcia, Ateret Anaby-Tavor, Ioana Baldini, Sara E. Berger, Bishwaranjan Bhattacharjee, Djallel Bouneffouf, Subhajit Chaudhury, Pin-Yu Chen, Lamogha Chiazor, Elizabeth M. Daly, Kirushikesh DB, Rogério Abreu de Paula, Pierre Dognin, Eitan Farchi, Soumya Ghosh, Michael Hind, Raya Horesh, George Kour, Ja Young Lee, Nishtha Madaan, Sameep Mehta, Erik Miehling, Keerthiram Murugesan, Manish Nagireddy, Inkit Padhi, David Piorkowski, Ambrish Rawat, Orna Raz, Prasanna Sattigeri, Hendrik Strobelt, Sarathkrishna Swaminathan, Christoph Tillmann, Aashka Trivedi, Kush R. Varshney, Dennis Wei, Shalisha Witherspooon, and Marcel Zalmanovici. Detectors for safe and reliable LLMs: Implementations, uses, and limitations, 2024. URL `https://arxiv.org/abs/2403.06009`.

Roi Cohen, May Hamri, Mor Geva, and Amir Globerson. LM vs LM: Detecting factual errors via cross examination. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12621–12640, 2023.

Tianyu Gao, Howard Yen, Jiatong Yu, and Danqi Chen. Enabling large language models to generate text with citations. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 6465–6488, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.398. URL `https://aclanthology.org/2023.emnlp-main.398/`.

Kristjan Greenewald, Luis Lastras, Thomas Parnell, Lucian Popa Vraj Shah, Giulio Zizzo, Chulaka Gunasekara, Ambrish Rawat, and David Cox. Activated LoRA: Fine-tuned LLMs for intrinsics, 2025.

Yannis Katsis, Sara Rosenthal, Kshitij Fadnis, Chulaka Gunasekara, Young-Suk Lee, Lucian Popa, Vraj Shah, Huaiyu Zhu, Danish Contractor, and Marina Danilevsky. MTRAG: A multi-turn conversational benchmark for evaluating retrieval-augmented generation systems, 2025. URL `https://arxiv.org/abs/2501.03468`.

Young-Suk Lee, Chulaka Gunasekara, Danish Contractor, Ramón Fernandez Astudillo, and Radu Florian. Multi-document grounded multi-turn synthetic dialog generation, 2024. URL `https://arxiv.org/abs/2409.11500`.

Potsawee Manakul, Adian Liusie, and Mark Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9004–9017, 2023.

Cheng Niu, Yuanhao Wu, Juno Zhu, Siliang Xu, Kashun Shum, Randy Zhong, Juntong Song, and Tong Zhang. Ragtruth: A hallucination corpus for developing trustworthy retrieval-augmented language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10862–10878, 2024.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don’t know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 784–789, 2018.

Maohao Shen, Subhro Das, Kristjan Greenewald, Prasanna Sattigeri, Gregory Wornell, and Soumya Ghosh. Thermometer: Towards universal calibration for large language models, 2024. URL `https://arxiv.org/abs/2403.08819`.

Jiajie Zhang, Yushi Bai, Xin Lv, Wanjun Gu, Danqing Liu, Minhao Zou, Shulin Cao, Lei Hou, Yuxiao Dong, Ling Feng, and Juanzi Li. LongCite: Enabling LLMs to generate fine-grained citations in long-context QA, 2024. URL `https://arxiv.org/abs/2409.02897`.

A Technical Appendices and Supplementary Material

A.1 Query Rewrite Quickstart Example

The following code describes how to use the Query Rewrite model.

```

import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import PeftModel
import json, re

INSTRUCTION_TEXT = "Reword the final utterance from the USER into a single utterance that doesn’t need the prior conversation history to understand the user’s intent. If the final utterance is a clear and standalone question, please DO NOT attempt to rewrite it, rather output the last user utterance as is. "
JSON = "Your output format should be in JSON: { \"rewritten_question\": <REWRITE> }"
REWRITE_PROMPT = "<|start_of_role|>rewrite: " + INSTRUCTION_TEXT + JSON + "<|end_of_role|>"

device=torch.device(’cuda’ if torch.cuda.is_available() else ’cpu’)

BASE_NAME = "ibm-granite/granite-3.2-8b-instruct"
LORA_NAME = "ibm-granite/granite-3.2-8b-lora-rag-query-rewrite"

tokenizer = AutoTokenizer.from_pretrained(BASE_NAME, padding_side=’left’, trust_remote_code=True)
model_base = AutoModelForCausalLM.from_pretrained(BASE_NAME, device_map =’auto’)
model_rewrite = PeftModel.from_pretrained(model_base, LORA_NAME)

# Input conversation
conv = [
    {
        "role":"user",
        "content":"Tim Cook is the CEO of Apple Inc."
    },
    {
        "role":"assistant",
        "content":"Yes, Tim Cook is the Chief Executive Officer of Apple Inc."
    },
    {
        "role":"user",
        "content":"and for Microsoft?"
    }
```

```
]

# Generate the query rewrite for the last turn in the above conversation
conv = [{"role":"system", "content":""}] + conv
input_text = tokenizer.apply_chat_template(conv, tokenize=False) +
    REWRITE_PROMPT
inputs = tokenizer(input_text, return_tensors="pt")
output = model_rewrite.generate(inputs["input_ids"].to(device), attention_mask=
    inputs["attention_mask"].to(device), max_new_tokens=80)
output_text = tokenizer.decode(output[0])

# Regex pattern to extract the JSON with the rewrite from the output of the
    model
pattern = r'\{\s*"[^"]+"\s*:\s*"[^"]*"\s*\}'
match_js = re.findall(pattern, output_text)[0]
try:
    #Parse the JSON and extract the rewrite
    rewrite = json.loads (match_js) ['rewritten_question']
except Exception as e:
    rewrite = match_js.split ("\"rewritten_question\": ", 1)[1]

print(f"Rewrite: {rewrite}\n")
# Rewrite: Who is the CEO of Microsoft?
```

A.2 Uncertainty Quantification Quickstart Example

The following code describes how to use the Uncrtainty Quantification model to answer questions and obtain intrinsic calibrated certainty scores. Note that a generic system prompt is included, this is not necessary and can be modified as needed.

```

import torch,os
from transformers import AutoTokenizer,  AutoModelForCausalLM
from peft import PeftModel, PeftConfig

token = os.getenv("HF_MISTRAL_TOKEN")
BASE_NAME = "ibm-granite/granite-3.2-8b-instruct"
LORA_NAME = "ibm-granite/granite-3.2-8b-lora-uncertainty"
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load model
tokenizer = AutoTokenizer.from_pretrained(BASE_NAME,padding_side='left',
    trust_remote_code=True, token=token)
model_base = AutoModelForCausalLM.from_pretrained(BASE_NAME,device_map="auto")
model_UQ = PeftModel.from_pretrained(model_base, LORA_NAME)

question = "What is IBM Research?"
print("Question:" + question)
question_chat = [
  {
    "role": "user",
    "content": question
  },
]

# Generate answer with base model
input_text = tokenizer.apply_chat_template(question_chat,tokenize=False,
    add_generation_prompt=True)


#tokenize
inputs = tokenizer(input_text, return_tensors="pt")
output = model_base.generate(inputs["input_ids"].to(device), attention_mask=
    inputs["attention_mask"].to(device), max_new_tokens=600)
```

```
output_text = tokenizer.decode(output[0])
answer = output_text.split("assistant<|end_of_role|>")[1]
print("Answer: " + answer)

# Generate certainty score
uq_generation_prompt = "<|start_of_role|>certainty<|end_of_role|>"
uq_chat = [
    {
    "role": "system",
    "content": ""
  },
  {
    "role": "user",
    "content": question
  },
    {
        "role": "assistant",
        "content": answer
    },
]

uq_text = tokenizer.apply_chat_template(uq_chat,tokenize=False) + uq_generation_prompt
# remove automatic system prompt
string_to_remove = tokenizer.apply_chat_template(uq_chat[0:1], tokenize=False, add_generation_prompt=False)
input_text = input_text[len(string_to_remove):]
uq_text = uq_text[len(string_to_remove):]

# tokenize and generate
inputs = tokenizer(uq_text, return_tensors="pt")
output = model_UQ.generate(inputs["input_ids"].to(device), attention_mask=inputs["attention_mask"].to(device), max_new_tokens=1)
output_text = tokenizer.decode(output[0])
uq_score = int(output_text[-1])
print("Certainty: " + str(5 + uq_score * 10) + "%")
```

A.3 Hallucination Detection Quickstart Example

As explained in Section 4, it is highly recommended to use the Hallucination Detection model through `Granite IO`. However, if you prefer to invoke the model directly, you can use the following code.

```

import torch
from transformers import AutoTokenizer,  AutoModelForCausalLM
from peft import PeftModel, PeftConfig
from nltk import tokenize
import json

BASE_NAME = "ibm-granite/granite-3.2-8b-instruct"
LORA_NAME = "ibm-granite/granite-3.2-8b-lora-rag-hallucination-detection"
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')

tokenizer = AutoTokenizer.from_pretrained(BASE_NAME, padding_side='left', trust_remote_code=True)
model_base = AutoModelForCausalLM.from_pretrained(BASE_NAME, device_map="auto")
model_hallucination = PeftModel.from_pretrained(model_base, LORA_NAME)

hallucination_sys_prompt = "Split the last assistant response into individual sentences. For each sentence in the last assistant response, identify the faithfulness score range. Ensure that your output includes all response sentence IDs, and for each response sentence ID, provide the corresponding faithfulness score range. The output must be a json structure."

```

```
def format_conversation(conversation):
    response_sents = tokenize.sent_tokenize(conversation[-1]["content"])
    response_sents_with_ids = []
    for ind, sent in enumerate(response_sents):
        response_sents_with_ids.append(f"<r{ind}> {sent}")
    conversation[-1]["content"] = ’ ’.join(response_sents_with_ids)
    conversation.append({
        "role": "system",
        "content": hallucination_sys_prompt
    })
    return conversation


conversation = [
    {
        "role": "user",
        "content": "What happened to Dennis Wilson of the Beach Boys in 1983?"
    },
    {
        "role": "assistant",
        "content": "Dennis Wilson of the Beach Boys drowned in Marina del Rey on December 28, 1983, while diving from a friend’s boat trying to recover items that he had previously thrown overboard in fits of rage. Forensic pathologists believed that Dennis experienced shallow-water blackout just before his death"
    }
]
input_conversation = format_conversation(conversation=conversation)

documents = [
    {
        "doc_id": 1,
        "text": "The Beach Boys are an American rock band formed in Hawthorne, California, in 1961. The group’s original lineup consisted of brothers Brian, Dennis, and Carl Wilson; their cousin Mike Love; and their friend Al Jardine. Distinguished by their vocal harmonies and early surf songs, they are one of the most influential acts of the rock era. The band drew on the music of jazz-based vocal groups, 1950s rock and roll, and black R&B to create their unique sound, and with Brian as composer, arranger, producer, and de facto leader, often incorporated classical or jazz elements and unconventional recording techniques in innovative ways. In 1983, tensions between Dennis and Love escalated so high that each obtained a restraining order against each other. With the rest of the band fearing that he would end up like Brian, Dennis was given an ultimatum after his last performance in November 1983 to check into rehab for his alcohol problems or be banned from performing live with them. Dennis checked into rehab for his chance to get sober, but on December 28, 1983, he fatally drowned in Marina del Rey while diving from a friend’s boat trying to recover items that he had previously thrown overboard in fits of rage."
    },
    {
        "doc_id": 2,
        "text": "A cigarette smoker since the age of 13, Carl was diagnosed with lung cancer after becoming ill at his vacation home in Hawaii, in early 1997. Despite his illness, Carl continued to perform while undergoing chemotherapy. He played and sang throughout the Beach Boys’ entire summer tour which ended in the fall of 1997. During the performances, he sat on a stool, but he stood while singing \"God Only Knows\".  Carl died of lung cancer in Los Angeles, surrounded by his family, on February 6, 1998, just two months after the death of his mother, Audree Wilson. He was interred at Westwood Village Memorial Park Cemetery in Los Angeles."
    },
    {
        "doc_id": 3,
```

```
        "text": "Carl Dean Wilson (December 21, 1946 - February 6, 1998) was an American musician, singer, and songwriter who co-founded the Beach Boys. He is best remembered as their lead guitarist, as the youngest brother of bandmates Brian and Dennis Wilson, and as the group's de facto leader in the early 1970s. He was also the band's musical director on stage from 1965 until his death. Influenced by the guitar playing of Chuck Berry and the Ventures, Carl's initial role in the group was that of lead guitarist and backing vocals, but he performed lead vocals on several of their later hits, including \"God Only Knows\" (1966), \"Good Vibrations\" (1966), and \"Kokomo\" (1988). By the early 1980s the Beach Boys were in disarray; the band had split into several camps. Frustrated with the band's sluggishness to record new material and reluctance to rehearse, Wilson took a leave of absence in 1981.  He quickly recorded and released a solo album, Carl Wilson, composed largely of rock n' roll songs co-written with Myrna Smith-Schilling, a former backing vocalist for Elvis Presley and Aretha Franklin, and wife of Wilson's then-manager Jerry Schilling. The album briefly charted, and its second single, \"Heaven\", reached the top 20 on Billboard's Adult Contemporary chart."
    }
]

# Generate answer
input_text = tokenizer.apply_chat_template(conversation=input_conversation, documents=documents, tokenize=False)

inputs = tokenizer(input_text, return_tensors="pt")
output = model_hallucination.generate(inputs["input_ids"].to(device), attention_mask=inputs["attention_mask"].to(device), max_new_tokens=500)
output_text = tokenizer.decode(output[0][inputs["input_ids"].shape[1]:], skip_special_tokens=True)
print("Output: " + json.loads(output_text))
```

A.4 Answerability Determination Quickstart Example

The following code describes how to use the Answerability Determination model.

```

import  torch
from  transformers  import  AutoTokenizer,  AutoModelForCausalLM
from  peft  import  PeftModel
from peft import PeftModelForCausalLM as lora_model

device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')

ANSWERABILITY_PROMPT = "<|start_of_role|>answerability<|end_of_role|>"
BASE_NAME = "ibm-granite/granite-3.2-8b-instruct"
LORA_NAME = "ibm-granite/granite-3.2-8b-lora-rag-answerability-prediction"

tokenizer = AutoTokenizer.from_pretrained(BASE_NAME, padding_side='left', trust_remote_code=True)
model_base = AutoModelForCausalLM.from_pretrained(BASE_NAME,device_map="auto")
model_answerability = PeftModel.from_pretrained(model_base, LORA_NAME)

convo = [
    {"role": "user", "content": "name the viceroy at the time of quit india movement"},
    {"role": "assistant", "content": "The Viceroy at the time of the Quit India Movement was Lord Linlithgow."},
    {"role": "user", "content": "opposition leader for the movement"},
    {"role": "assistant", "content": "A movement was launched at the Bombay session of the All - India Congress Committee by Mahatma Gandhi on 8 August 1942.  Many students paid attention to Subhas Chandra Bose as a leader,  who was in exile. He organized the Indian Legion in Germany , reorganized the
```

```
    Indian National Army with Japanese assistance , and soliciting help from the
     Axis Powers , conducted a guerrilla war against the British authorities."},
    {"role": "user", "content": "who is Patel?"},
    {"role": "assistant", "content": "Vallabhbhai Patel, popularly known as
    Sardar Patel , was the first Deputy Prime Minister of India . He was an
    Indian barrister and statesman , a senior leader of the Indian National
    Congress and a founding father of the Republic of India who played a leading
     role in the country 's struggle for independence and guided its integration
     into a united , independent nation . He acted as de facto Supreme Commander
     - in - chief of Indian army during the political integration of India and
    the Indo - Pakistani War of 1947 . He is credited by historians with
    ensuring the success of the rebellion across India. He was arrested and was
    imprisoned with the entire Congress Working Committee from 1942 to 1945"},
    {"role": "user", "content": "how do you pronounce Vallabhbhai?"},
 ]


documents = [
    {'text': "Vallabhbhai Patel\nAmong Patel 's surviving family , Maniben Patel
     lived in a flat in Mumbai for the rest of her life following her father 's
    death ; she often led the work of the Sardar Patel Memorial Trust , which
    organises the prestigious annual Sardar Patel Memorial Lectures , and other
    charitable organisations . Dahyabhai Patel was a businessman who was elected
     to serve in the Lok Sabha ( the lower house of the Indian Parliament ) as
    an MP in the 1960s ."},
    {'text': "Vallabhbhai Patel\nPatel 's date of birth was never officially
    recorded ; Patel entered it as 31 October on his matriculation examination
    papers . He belonged to the Leuva Patel Patidar community of Central Gujarat
     , although the Leuva Patels and Kadava Patels have also claimed him as one
    of their own ."},
    {'text': "Vallabhbhai Patel\nIn April 2015 the Government of India
    declassified surveillance reports suggesting that Patel , while Home
    Minister , and Nehru were among officials involved in alleged government -
    authorised spying on the family of Subhas Chandra Bose ."}
]

convo = [{"role":"system", "content": ""}] +convo

string = tokenizer.apply_chat_template(convo,documents=documents, tokenize=False
    ,add_generation_prompt=False)
string_to_remove = tokenizer.apply_chat_template(convo[0:1], tokenize=False,
    add_generation_prompt=False)
string = string[len(string_to_remove):]
inputs = string + ANSWERABILITY_PROMPT

inputT = tokenizer(inputs, return_tensors="pt")

output = model_answerability.generate(inputT["input_ids"].to(device),
    attention_mask=inputT["attention_mask"].to(device), max_new_tokens=3)
output_text = tokenizer.decode(output[0])
answer = output_text.split(ANSWERABILITY_PROMPT)[1]
print(answer)
```

A.5 CITATION GENERATION QUICKSTART EXAMPLE

As explained in Section 6, it is highly recommended to use the Citation Generation model through `Granite IO`. However, if you prefer to invoke the model directly, you can use the following code. Note that the code assumes that the documents and the last assistant response have been already split into sentences.

```
import torch
from transformers import AutoTokenizer,  AutoModelForCausalLM
from peft import PeftModel, PeftConfig
import json

```

```
BASE_NAME = "ibm-granite/granite-3.2-8b-instruct"
LORA_NAME = "ibm-granite/granite-3.2-8b-lora-rag-citation-generation"
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')

tokenizer = AutoTokenizer.from_pretrained(BASE_NAME, padding_side='left',
    trust_remote_code=True)
model_base = AutoModelForCausalLM.from_pretrained(BASE_NAME, device_map="auto")
model_citation = PeftModel.from_pretrained(model_base, LORA_NAME)

conversation = [
    {"role": "user", "content": "What is the visibility level of Git Repos and
    Issue Tracking projects?"},
    {"role": "assistant", "content": "<r0> Git Repos and Issue Tracking projects
     can have one of the following visibility levels: private, internal, or
    public. <r1> Private projects are visible only to project members, internal
    projects are visible to all users that are logged in to IBM Cloud, and
    public projects are visible to anyone. <r2> By default, new projects are set
     to private visibility level, which is the most secure for your data."}]

documents = [
    {"doc_id": 0, "text": "<c0> Git Repos and Issue Tracking is an IBM-hosted
    component of the Continuous Delivery service. <c1> All of the data that you
    provide to Git Repos and Issue Tracking, including but not limited to source
     files, issues, pull requests, and project configuration properties, is
    managed securely within Continuous Delivery. <c2> However, Git Repos and
    Issue Tracking supports various mechanisms for exporting, sending, or
    otherwise sharing data to users and third parties. <c3> The ability of Git
    Repos and Issue Tracking to share information is typical of many social
    coding platforms. <c4> However, such sharing might conflict with regulatory
    controls that apply to your business. <c5> After you create a project in Git
     Repos and Issue Tracking, but before you entrust any files, issues, records
    , or other data with the project, review the project settings and change any
     settings that you deem necessary to protect your data. <c6> Settings to
    review include visibility levels, email notifications, integrations, web
    hooks, access tokens, deploy tokens, and deploy keys. <c7> Project
    visibility levels \n\nGit Repos and Issue Tracking projects can have one of
    the following visibility levels: private, internal, or public. <c8> *
    Private projects are visible only to project members. <c9> This setting is
    the default visibility level for new projects, and is the most secure
    visibility level for your data. <c10> * Internal projects are visible to all
     users that are logged in to IBM Cloud. <c11> * Public projects are visible
    to anyone. <c12> To limit project access to only project members, complete
    the following steps:\n\n\n\n1. <c13> From the project sidebar, click
    Settings > General. <c14> 2. <c15> On the General Settings page, click
    Visibility > project features > permissions. <c16> 3. <c17> Locate the
    Project visibility setting. <c18> 4. <c19> Select Private, if it is not
    already selected. <c20> 5. <c21> Click Save changes. <c22> Project
    membership \n\nGit Repos and Issue Tracking is a cloud hosted social coding
    environment that is available to all Continuous Delivery users. <c23> If you
     are a Git Repos and Issue Tracking project Maintainer or Owner, you can
    invite any user and group members to the project. <c24> IBM Cloud places no
    restrictions on who you can invite to a project."},
    {"doc_id": 1, "text": "<c25> After you create a project in Git Repos and
    Issue Tracking, but before you entrust any files, issues, records, or other
    data with the project, review the project settings and change any settings
    that are necessary to protect your data. <c26> Settings to review include
    visibility levels, email notifications, integrations, web hooks, access
    tokens, deploy tokens, and deploy keys. <c27> Project visibility levels \n\
    nGit Repos and Issue Tracking projects can have one of the following
    visibility levels: private, internal, or public. <c28> * Private projects
    are visible only to project members. <c29> This setting is the default
    visibility level for new projects, and is the most secure visibility level
    for your data. <c30> * Internal projects are visible to all users that are
    logged in to IBM Cloud. <c31> * Public projects are visible to anyone. <c32>
     To limit project access to only project members, complete the following
```

```
    steps:\n\n\n\n1. <c33> From the project sidebar, click Settings > General. <
    c34> 2. <c35> On the General Settings page, click Visibility > project
    features > permissions. <c36> 3. <c37> Locate the Project visibility setting
    . <c38> 4. <c39> Select Private, if it is not already selected. <c40> 5. <
    c41> Click Save changes. <c42> Project email settings \n\nBy default, Git
    Repos and Issue Tracking notifies project members by way of email about
    project activities. <c43> These emails typically include customer-owned data
     that was provided to Git Repos and Issue Tracking by users. <c44> For
    example, if a user posts a comment to an issue, Git Repos and Issue Tracking
     sends an email to all subscribers. <c45> The email includes information
    such as a copy of the comment, the user who posted it, and when the comment
    was posted. <c46> To turn off all email notifications for your project,
    complete the following steps:\n\n\n\n1. <c47> From the project sidebar,
    click Settings > General. <c48> 2. <c49> On the **General Settings **page,
    click Visibility > project features > permissions. <c50> 3. <c51> Select the
     Disable email notifications checkbox. <c52> 4. <c53> Click Save changes. <
    c54> Project integrations and webhooks"}]

# Add system prompt
citation_sys_prompt = "Split the last assistant response into individual
    sentences. For each sentence in the response, identify the statement IDs
    from the documents that it references. Ensure that your output includes all
    response sentence IDs, and for each response sentence ID, provide the
    corresponding referring document sentence IDs."
conversation.append({"role": "system", "content": citation_sys_prompt})

# Generate answer
input_text = tokenizer.apply_chat_template(conversation=conversation, documents=
    documents, tokenize=False)
inputs = tokenizer(input_text, return_tensors="pt")
output = model_citation.generate(inputs["input_ids"].to(device), attention_mask=
    inputs["attention_mask"].to(device), max_new_tokens=500)
output_text = tokenizer.decode(output[0][inputs["input_ids"].shape[1]:],
    skip_special_tokens=True)
print("Output: ")
print(json.loads(output_text))
```