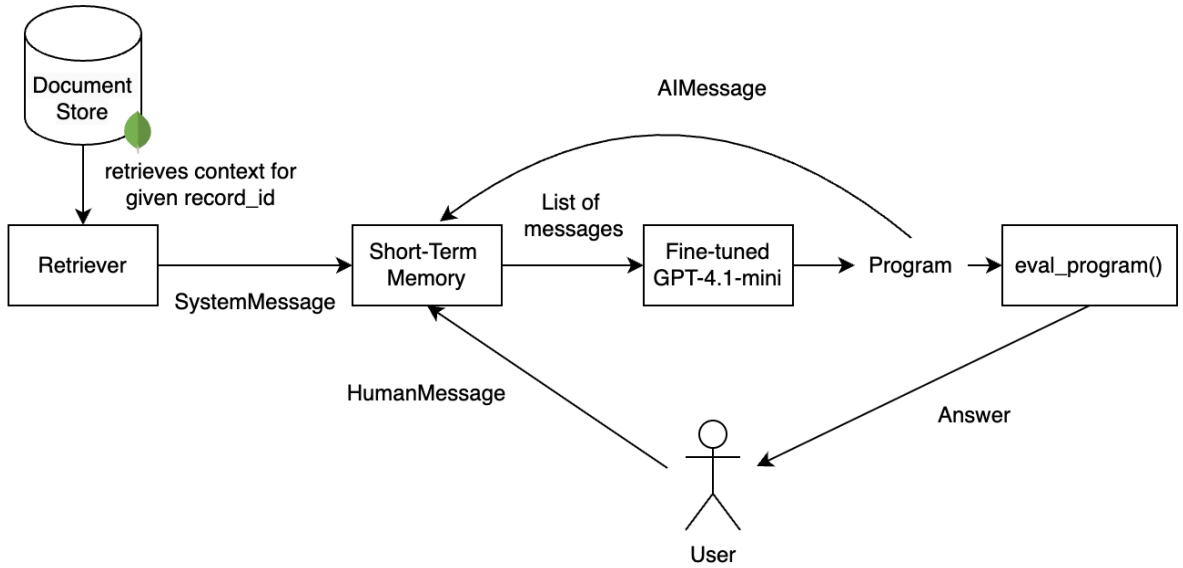
ConvFinQA Report

System Design



After the user submits a record_id (either through the CLI or streamlit app), this starts a connection to MongoDB, after which the retriever queries the database to retrieve the pre-text, table and post-text for the record_id. This is then formatted into a SystemMessage, which is added to the Short-Term Memory module, a list.

Subsequently, an interactive chat session is started with the user. When the user asks a question, it gets formatted into a HumanMessage and appended to ST memory. The list of messages is then passed into a Fine-tuned GPT-4.1 mini which will output a program. The program is formatted as an AIMessage and added to ST memory for subsequent turns. The program is also passed into an eval_program() function, which computes the answer for the given program. Lastly, the answer is returned to the user. This describes the process for a single-turn.

For subsequent turns, the process repeats, with each question getting added as a HumanMessage to ST memory, and each program output added as an AIMessage. This ensures that the model has context of the entire multi-turn conversation when generating programs.

Design Considerations:

- **Database**: Given the unstructured nature of the pre-text, table and post-text, I opted to store the documents in a NoSQL database such as MongoDB. Additionally, with native support for vector and BM25 indexing, MongoDB allows us to explore semantic and keyword search without relying on a different database provider.
- **Retrieval**: The entire context, comprising pre-text, table and post-text, is retrieved from MongoDB for a given record_id. This is decided over retrieval of only relevant chunks for a given question because of a few reasons:

- **Passing in the entire context is not costly**: For the ConvFinQA dataset, the context length (text and table) is not very long, with the average number of tokens being 675.61 and the max being 2338.

Conversations	3,892
Questions	14,115
Report pages	2,066
Vocabulary	20k
Avg. # questions in one conversation	3.67
Avg. question length	10.59
Avg. # sentences in input text	23.65
Avg. # rows in input table	6.39
Avg. # tokens in all inputs (text & table)	675.61
Max. # tokens in all inputs (text & table)	2338.00

Table 2: Statistics of **CONVFINQA**.

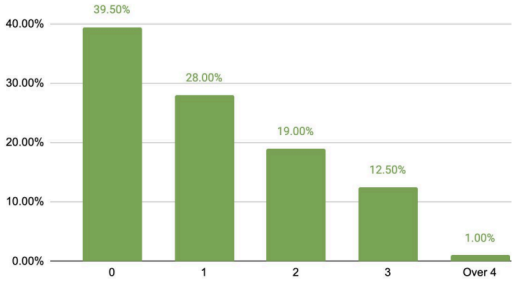


Figure 4: Distribution of the longest dependency distances of the questions in **CONVFINQA**. Over 60% of the questions have longer dependencies with previous questions.

- **Retrieval of only relevant chunks is unlikely to significantly improve accuracy**: Modern LLMs such as GPT-4.1 mini are able to process long context lengths (up to 1 million tokens for GPT-4.1). They have also been trained to notice relevant text and ignore distractors across long and short context lengths. Thus, passing in the entire context is less likely to confuse the model, especially in our case where the context length is relatively short. In fact, from error analysis, very few errors are attributed to the LLM extracting the wrong information from the context, confirming that changing the retrieval mechanism to retrieve only relevant chunks is unlikely to significantly improve accuracy.
- **Retrieval of only relevant chunks involves a more complicated design which might hurt performance:** For retrieving only the relevant chunks for a given question, sophisticated chunking, retrieval and query rewriting strategies are required. Effective chunking must preserve the semantic integrity of the content such as keeping tables intact and ensuring that meaningful units such as "Operating expenses in 2020 are 5 million" are not split across chunks.

  For effective retrieval, experimentation will be needed to discover the best approach. This includes experimenting with a pretrained embedding model trained on financial documents for semantic search, combining semantic search and BM25 search to take into account keywords, as well as using a re-ranker to improve retrieval performance.

In addition, because ConFinQA involves multi-turn conversations, query rewriting becomes necessary to resolve context. For example, if the first question is “what are the total operating expenses in 2009?” and the follow-up is “what about in 2008?”, the second query should be rewritten to “what are the total operating expenses in 2008?” to ensure accurate retrieval.

As a result, this introduces a lot more complexity into the system. If one of the components is not working correctly, the relevant chunk might not be retrieved. This results in an incorrect program and answer being generated for a given question, which will impact the accuracy for subsequent turns as well.

Nevertheless, if we had a different dataset with very long documents, passing in the entire context will be costly and it might be worth experimenting retrieval of only relevant chunks.

- **Program and Answer generation**: A fine-tuned LLM is used to generate the program, while eval_program() function computes the final answer based on the program.
    - **Generating a program allows us to evaluate the mathematical reasoning ability of the LLM in a rigorous way:** Generating only the final answer provides limited insight into the LLM’s reasoning ability. One approach to evaluate the reasoning ability is to ask the LLM to provide an explanation when generating the answer, then use LLM-as-a-judge to evaluate the explanation. However, given that the explanation is qualitative, the evaluation will not be as consistent and rigorous as evaluating a program. Furthermore, the dataset includes annotated ground truth programs, which allows us to objectively evaluate the generated programs by comparing them to the ground truth.
    - **We outsource the computation to an external function as LLMs are prone to making mistakes:** For complex mathematical calculations, LLMs are often error-prone. To address this, I experimented with approaches that delegate computation to external functions or tools.

      One such approach involved augmenting the LLM with a suite of tools to handle calculations. However, performance was suboptimal—likely because the LLM had to handle too many responsibilities: understanding the context, determining whether the task was extractive or computational, selecting the appropriate tool, and supplying the correct parameters.

      A more effective approach was to have the LLM focus solely on generating the correct program, and then rely on an external function to compute the final answer. This separation of concerns led to better performance and reduced errors.
    - **Without fine-tuning, an LLM struggles to adhere to the program format:** A program follows a domain-specific language with strict formatting rules—such as returning only a value for extraction questions, referencing

results from previous steps using special tokens like #n, and correctly chaining operations. Even with detailed step-by-step instructions, few-shot examples or agentic frameworks, the LLM often fails to consistently follow the required structure. Based on my experiments, fine-tuning the model on examples from the dataset significantly improved format adherence and consequently program accuracy.

I chose to fine-tune GPT-4.1 mini because the GPT-4.1 series is recommended by OpenAI for supervised fine-tuning and generating output in a specific format. I opted for GPT-4.1 mini over GPT-4.1 due to its significantly lower training cost.

Evaluation Metrics

Inspired by the FinQA and ConvFinQA papers, we measure the accuracy of the system using **execution accuracy** and **program accuracy**.

- Execution accuracy measures answer correctness and answers the question “Did the system produce the correct final answer?”. It checks for an exact match between the generated answer and the ground truth answer, after applying normalization and rounding to 5 decimal places for numeric answers. Execution accuracy is important for a question-answering system since an incorrect answer will negatively impact user experience. However, the limitation of this metric is that a logically incorrect program can happen to produce the right final answer. This is especially so for a program that has a greater() operation: both greater(50, 1) and greater(45, 1) produce the same result, even though the numbers extracted in each program are different.
- Thus, program accuracy is needed to evaluate the model’s logic behind the generated answer, and it answers the question “Was the model’s reasoning mathematically correct?”. It compares the generated program with the ground truth program, and evaluates if they are mathematically equivalent. For instance, add(a,b) and add(b,a) should be considered correct, since addition is commutative.

One key consideration in evaluating the system is determining the right granularity for measuring execution accuracy and program accuracy. Evaluating execution accuracy at the conversation level—that is, only considering the correctness of the final answer— can miss errors for intermediate answers. In a multi-turn setting, this is especially important since every incorrect answer will negatively impact user experience. To capture the correctness of intermediate answers, turn-level execution accuracy will be a better metric, since it answers the question “For what proportion of questions did the system produce the correct answer?”.

As for program accuracy, the domain-specific language specifies that programs referencing previous steps should be expressed as a concatenation of those earlier programs. As such, measuring program accuracy at the conversation level—that is, evaluating only the correctness of the final program—provides a meaningful assessment of the model’s ability to reason across multiple turns. In contrast, turn-level program accuracy is less representative of overall reasoning performance, as it tends to over-penalize the system. Errors in earlier turns can propagate to later ones due to program concatenation. For example, an incorrect

program in turn 3 will likely cause the program in turn 4 to be incorrect as well. This cascading effect makes turn-level evaluation harsher and potentially less fair than conversation-level evaluation.

Therefore, the key metrics for evaluating our system's performance are **conversation-level program accuracy**—to assess the system's overall reasoning ability—and **turn-level execution accuracy**—to evaluate the correctness of each individual answer provided to the user. Nevertheless, we will still report turn-level program accuracy and conversation-level execution accuracy metrics to aid in diagnostics. For example, a large gap between conversation-level and turn-level execution accuracies may indicate that while the system often produces correct intermediate answers, it struggles with final answers. In addition, if turn-level execution accuracy is lower than turn-level program accuracy, this could signal an issue with the evaluation script, since execution accuracy is generally expected to be higher—multiple different programs can yield the same correct answer.

Performance comparison

Establishing a baseline

To quickly iterate and identify a promising approach at low cost, I created a small evaluation set of 50 conversations and experimented with several methods:

- Augmenting the LLM with external tools for calculations: As discussed in the earlier section, this approach performed poorly. The LLM struggled with managing multiple responsibilities such as choosing the correct tool, leading to unreliable outputs.
- Zero-shot prompting for program generation: Despite being given detailed step-by-step instructions, the LLM still frequently made mistakes, often resulting in invalid programs. This behavior was observed across both fast-thinking and reasoning-oriented models.
- Few-shot prompting for program generation: This approach yielded a significant improvement over zero-shot prompting. While some issues remained, such as incorrect argument ordering, the generated programs were generally valid. As a result, this method was chosen as our baseline.

Providing several examples helped the LLM better adhere to the domain-specific language. This led to the hypothesis that fine-tuning could further enhance the model's ability to generate correctly formatted programs, by learning from a larger set of structured examples. After fine-tuning GPT-4.1 mini on 80 conversations, I observed an improvement of at least 20% in both conversation-level program accuracy and turn-level execution accuracy on the evaluation set. These results demonstrate the promise of the approach and suggest that scaling up the fine-tuning with a larger dataset could lead to even more conclusive improvements.

Train-test split

To scale up the experiment, I set aside 1,000 conversations for supervised finetuning and 200 conversations for evaluation.

To ensure that both datasets capture important characteristics of the data, I applied a multi-stage stratified split based on three key variables that are essential for robust model performance:

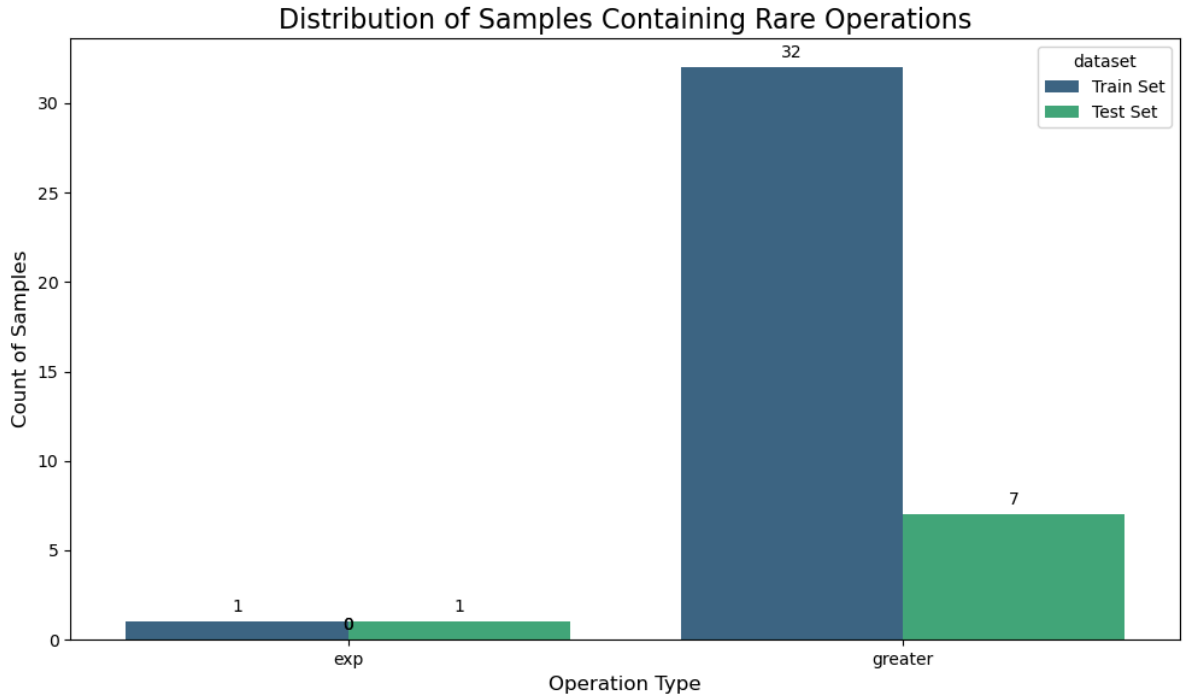
- Rare operations such as exp() and greater(): to expose the model to uncommon operations and ensure it can generate correct programs involving them
- has_type2_question: to ensure the model is trained and evaluated on more complex type 2 questions
- num_dialogue_turns: to ensure the model can reason effectively across varying number of turns

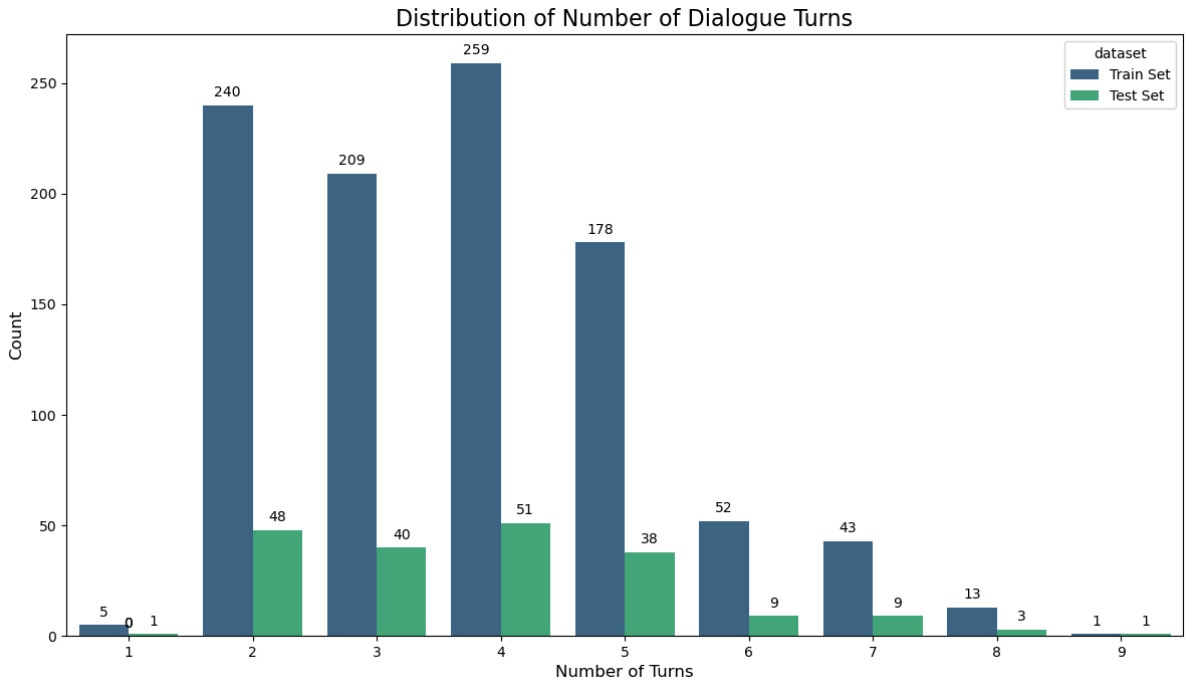
The splitting process was carried out as follows:

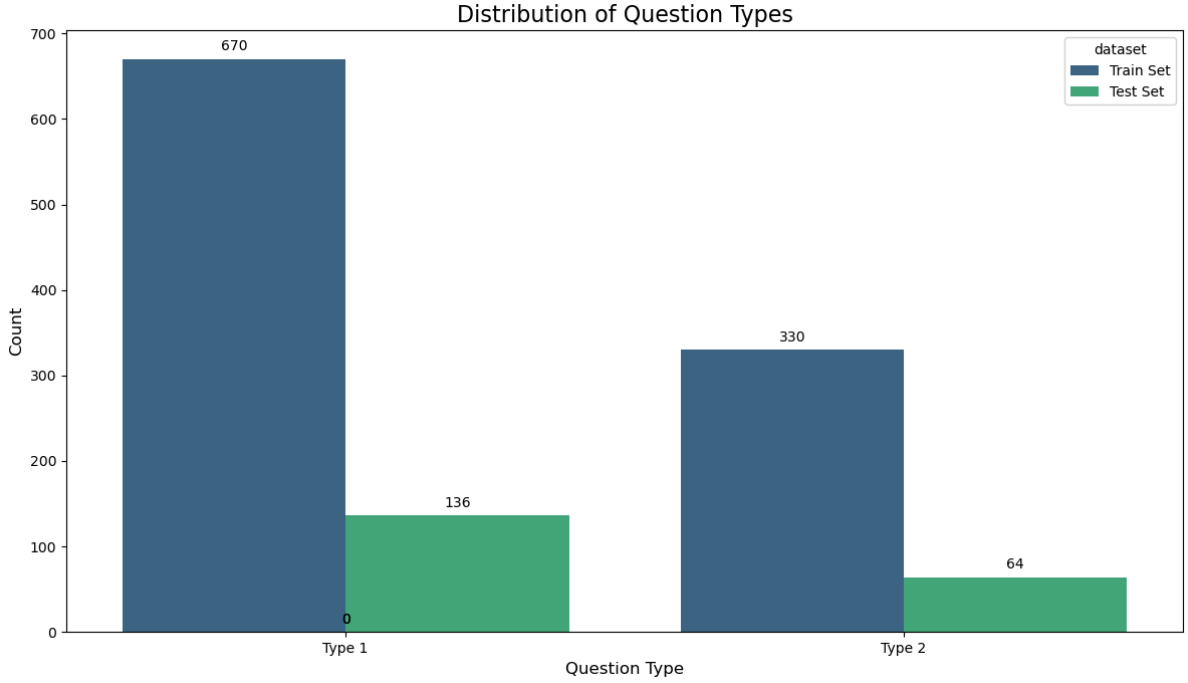
1. Rare categories were handled first:
    - For extremely rare categories like exp() and 9-turn conversations (only 2 samples each), I allocated 1 sample to training and 1 to test.
    - For categories with slightly more data—greater() and conversations with 1, 7, or 8 turns—I applied a proportional split (~5:1 ratio between training and evaluation sets).

2. For the remaining conversations, I performed **stratified sampling** based on:
    - num_dialogue_turns
    - has_type2_question

The resulting distributions across the three key variables for both the training and test sets can be seen below:







Evaluation on test set of 200 samples

	Few-shot prompting (GPT-4.1)	Few-shot prompting (Gemini 2.5 pro)	Fine-tuned model (GPT-4.1 mini)
Conversation-level Execution Accuracy	64.00%	61.00%	86.00%

Conversation-level Program Accuracy	48.50%	48.00%	**83.50%**
Turn-level Execution Accuracy	66.23%	65.31%	**86.86%**
Turn-level Program Accuracy	57.03%	57.42%	85.81%

The results show that the fine-tuned GPT-4.1 mini significantly outperforms few-shot prompting, despite being a smaller model. It achieves a 35% higher conversation-level program accuracy compared to GPT-4.1 with few-shot prompting, demonstrating strong reasoning capabilities and a better grasp of the domain-specific language. In addition, it attains a 20.63% higher turn-level execution accuracy, indicating that a much larger proportion of its answers to users are correct.

Interestingly, for the few-shot prompting approaches, conversation-level program accuracy is noticeably lower than turn-level program accuracy. This discrepancy likely arises because the model performs reasonably well on intermediate extraction tasks, which often involve simply returning a value. In contrast, generating the final program requires more precise adherence to the domain-specific syntax, which few-shot prompting tends to struggle with.

Error Analysis

When analyzing the erroneous samples, a large proportion of the errors is due to data quality issues:

- **Missing values in the context**: For the sample with id “Double_FRT/2006/page_133.pdf”, the first question was "what was the ratio of the additions in 2005 to the ones in 2004?" and the ground truth program was “divide(83656, 82551)”. However, these values are not present in the provided context. There appears to be some parsing issues during the preparation of the ConvFinQA dataset.
- **Incorrect labeling of ground truth:** for the sample with id “Double_IP/2013/page_61.pdf”, the fifth question asked for net sales in 2012 in billions. The ground truth answer was 11600.0, however the correct answer should be 11.6, which was correctly captured by our system.
- **Inconsistent units:** for the sample with id “Single_HII/2011/page_60.pdf-2”, one question asked for the weighted average discount rate for post-retirement benefits in 2011. The context explicitly stated the value as 4.94%, and the ground truth answer was 4.94. However, in another sample “Double_AWK/2015/page_81.pdf”, a similar question asked for percentage of capital structure, and the ground truth answer was given as 50.6%. This inconsistency—whether the % symbol is included in the answer or implicitly dropped—can confuse the model during training, especially when attempting to match numerical answers to the expected format.
- **Inconsistent signs**: For the sample with id “Single_ABMD/2015/page_93.pdf-2”, one question asked about the foreign currency translation impact, referencing the

following table. The ground truth answer was 3789.0 (a positive value), whereas our system's answer was -3789.0.


```
"march 31 2015 ( in $ 000 2019s )": {
    "beginning balance": 2014.0,
    "additions": 18500.0,
    "foreign currency translation impact": -3789.0,
    "ending balance": 14711.0
}
```


In contrast, for id "Single_PM/2017/page_117.pdf-4", a similar extraction question involved a negative value in the table, and the ground truth answer correctly retained the negative sign. This highlights an inconsistency in the dataset, since the ground truth sometimes treats negative values as positive and sometimes retains the sign, even for similar types of questions.

There were a few tricky cases that the model failed to answer correctly:

- For sample "Double_AMT/2012/page_121.pdf", one question asked for the total of customer-related and network location intangibles from american tower corporation. Since customer-related and network location intangibles were mentioned twice in the context, the model was confused and extracted the wrong values, instead of the values from american tower corporation.
- For sample "Single_MRO/2007/page_79.pdf-1", one question asked for the quarterly dividends per share for 2007. However, the context only provided the annual dividends per share, which was 0.92, with no explicit mention of quarterly figures. As a result, our system extracted 0.92 directly from the context, whereas the ground truth answer was 0.23, obtained by dividing 0.92 by 4.
- For sample "Double_ETR/2015/page_24.pdf", the conversation featured a complex type 2 question. It began by asking about net revenue in 2014, followed by several questions focusing on 2013, culminating in a final question that referenced 2013's net revenue. However, instead of correctly referencing the 2013 figure, the model mistakenly reused the 2014 net revenue, highlighting a limitation in its ability to track context and reason across multiple turns.

Future Work

Since most of the errors stemmed from data quality issues, the most impactful next step is to inspect and clean these problematic samples. The effectiveness of fine-tuning is heavily dependent on the quality of the underlying data, making this a critical area for improvement.

For dealing with tricky cases, there are several approaches we can explore:

- We can include more of such examples in the train set so that the model will learn how to handle such cases.
- For the incorrect extraction and type 2 question cases, a more powerful model with stronger extraction and reasoning ability might be able to handle such cases correctly. We can try fine-tuning models such as GPT-4.1 and compare performance with the fine-tuned GPT-4.1 mini.
- At times, the question may be ambiguous or unclear. Depending on the business context, it may be more appropriate to ask the user for clarification rather than attempting to generate a program based on incomplete understanding.