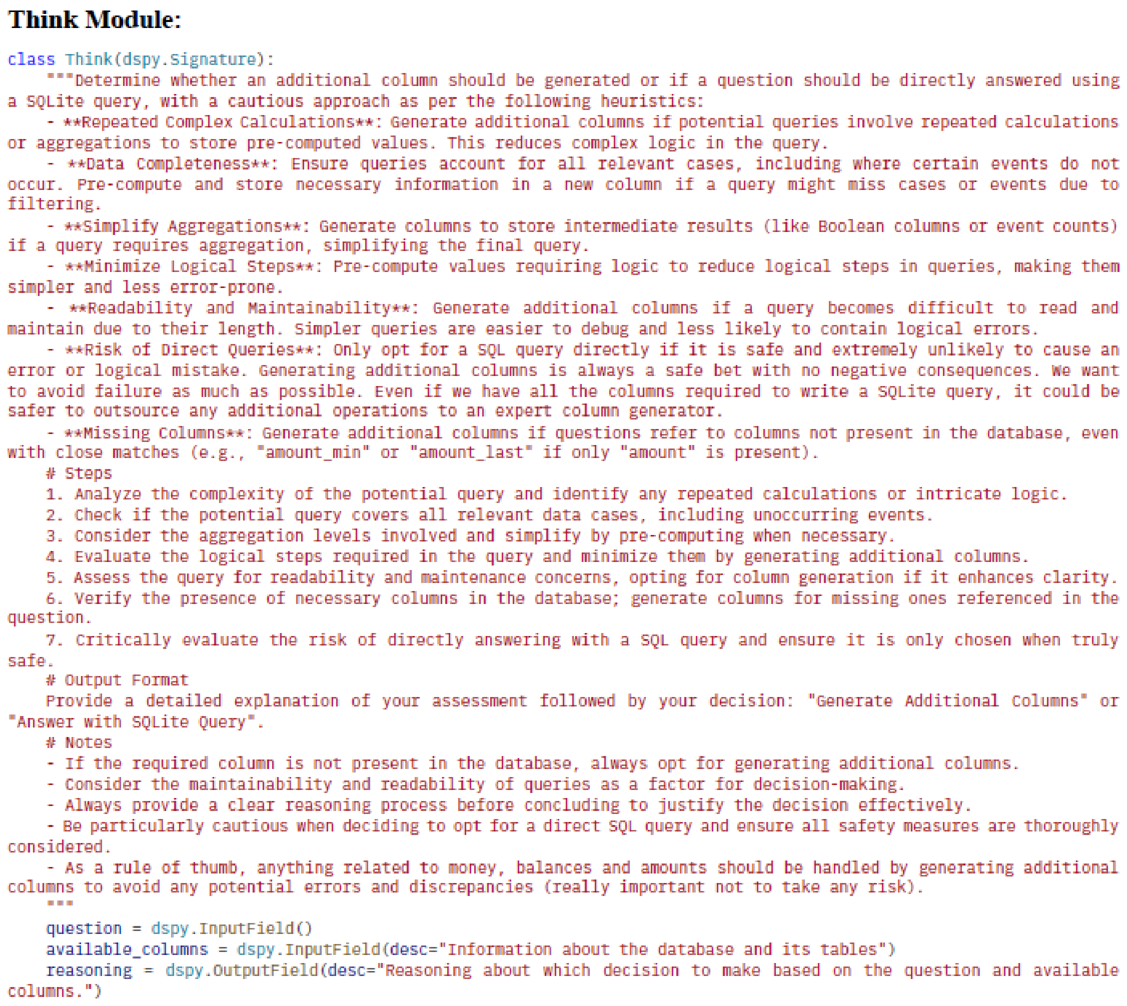
Appendix

Supplementary Material for the Submission *An Agentic LLM-based System for Exploratory Process Mining*

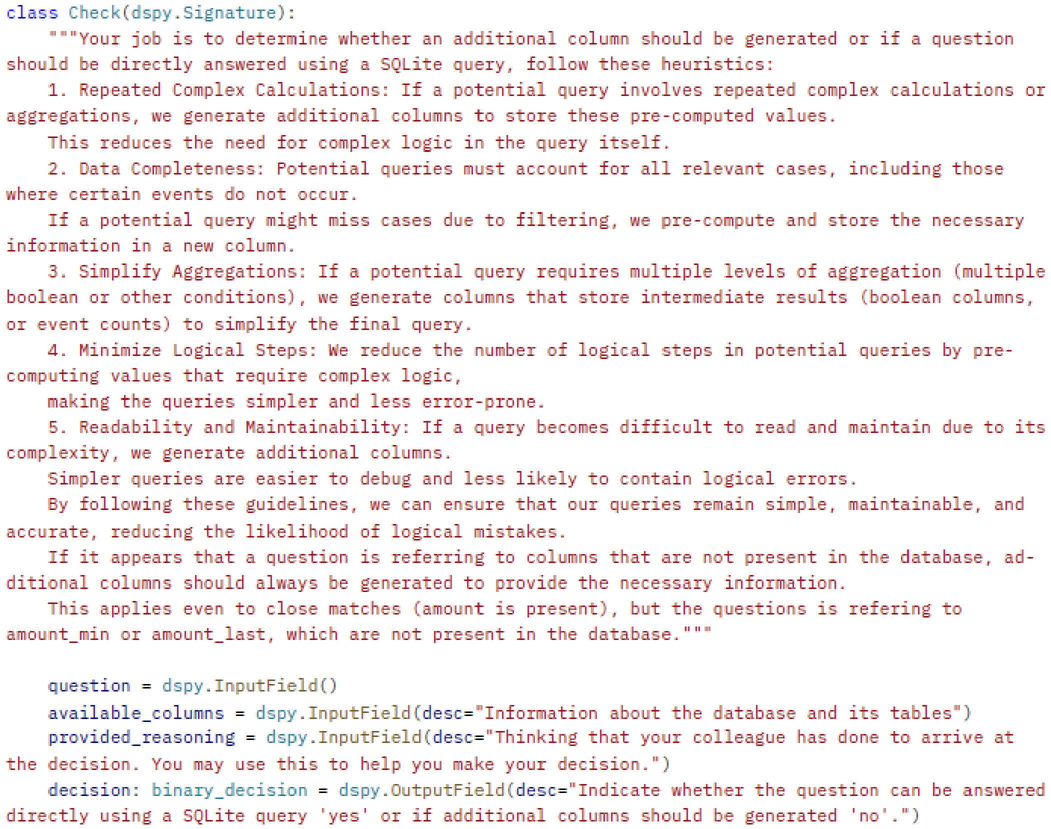


Think Module:

```
class Think(dspy.Signature):
    """Determine whether an additional column should be generated or if a question should be directly answered using
a SQLite query, with a cautious approach as per the following heuristics:
    - **Repeated Complex Calculations**: Generate additional columns if potential queries involve repeated calculations
or aggregations to store pre-computed values. This reduces complex logic in the query.
    - **Data Completeness**: Ensure queries account for all relevant cases, including where certain events do not
occur. Pre-compute and store necessary information in a new column if a query might miss cases or events due to
filtering.
    - **Simplify Aggregations**: Generate columns to store intermediate results (like Boolean columns or event counts)
if a query requires aggregation, simplifying the final query.
    - **Minimize Logical Steps**: Pre-compute values requiring logic to reduce logical steps in queries, making them
simpler and less error-prone.
    - **Readability and Maintainability**: Generate additional columns if a query becomes difficult to read and
maintain due to their length. Simpler queries are easier to debug and less likely to contain logical errors.
    - **Risk of Direct Queries**: Only opt for a SQL query directly if it is safe and extremely unlikely to cause an
error or logical mistake. Generating additional columns is always a safe bet with no negative consequences. We want
to avoid failure as much as possible. Even if we have all the columns required to write a SQLite query, it could be
safer to outsource any additional operations to an expert column generator.
    - **Missing Columns**: Generate additional columns if questions refer to columns not present in the database, even
with close matches (e.g., "amount_min" or "amount_last" if only "amount" is present).
    # Steps
    1. Analyze the complexity of the potential query and identify any repeated calculations or intricate logic.
    2. Check if the potential query covers all relevant data cases, including unoccurring events.
    3. Consider the aggregation levels involved and simplify by pre-computing when necessary.
    4. Evaluate the logical steps required in the query and minimize them by generating additional columns.
    5. Assess the query for readability and maintenance concerns, opting for column generation if it enhances clarity.
    6. Verify the presence of necessary columns in the database; generate columns for missing ones referenced in the
question.
    7. Critically evaluate the risk of directly answering with a SQL query and ensure it is only chosen when truly
safe.
    # Output Format
    Provide a detailed explanation of your assessment followed by your decision: "Generate Additional Columns" or
"Answer with SQLite Query".
    # Notes
    - If the required column is not present in the database, always opt for generating additional columns.
    - Consider the maintainability and readability of queries as a factor for decision-making.
    - Always provide a clear reasoning process before concluding to justify the decision effectively.
    - Be particularly cautious when deciding to opt for a direct SQL query and ensure all safety measures are thoroughly
considered.
    - As a rule of thumb, anything related to money, balances and amounts should be handled by generating additional
columns to avoid any potential errors and discrepancies (really important not to take any risk).
    """
    question = dspy.InputField()
    available_columns = dspy.InputField(desc="Information about the database and its tables")
    reasoning = dspy.OutputField(desc="Reasoning about which decision to make based on the question and available
columns.")
```

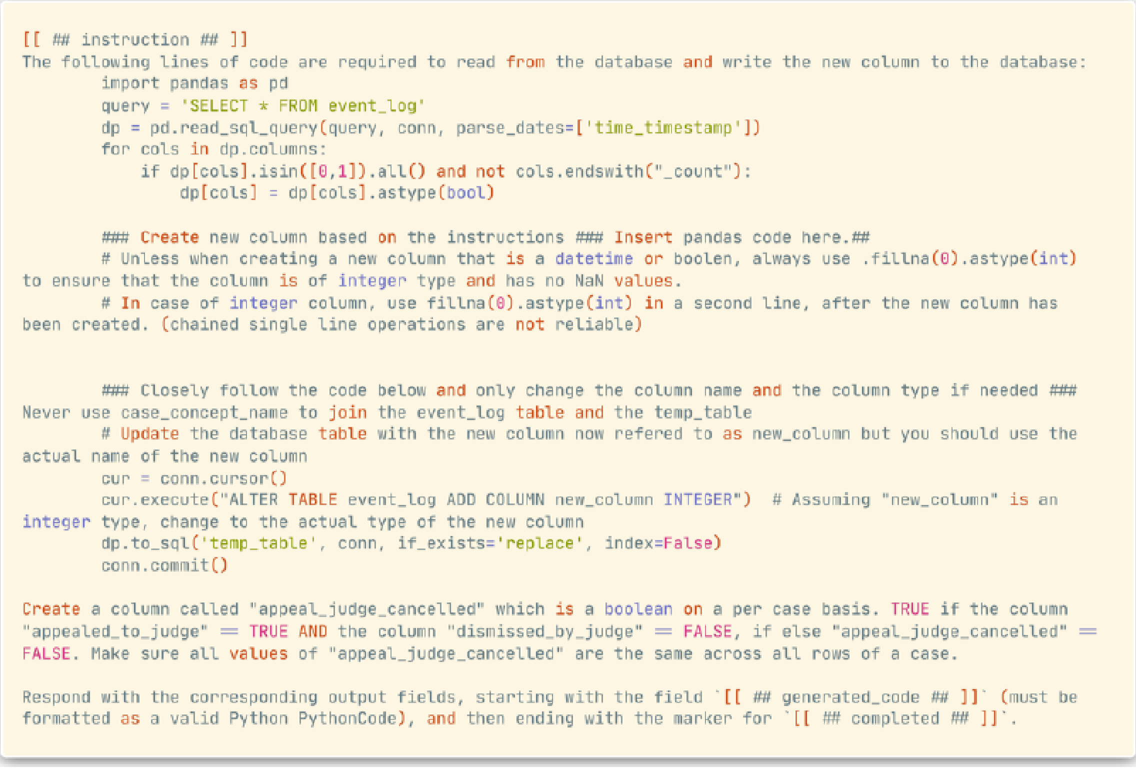
Fig. 6. DSPy signature of the *Think* module.

Check Module:



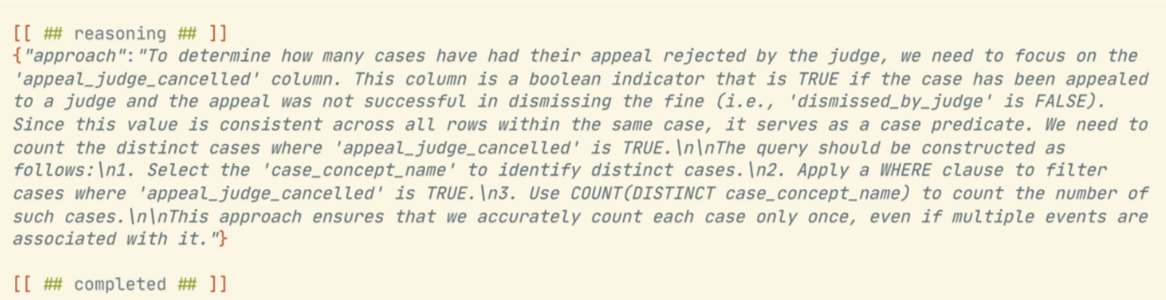
```
class Check(dspy.Signature):
    """Your job is to determine whether an additional column should be generated or if a question
should be directly answered using a SQLite query, follow these heuristics:
    1. Repeated Complex Calculations: If a potential query involves repeated complex calculations or
aggregations, we generate additional columns to store these pre-computed values.
    This reduces the need for complex logic in the query itself.
    2. Data Completeness: Potential queries must account for all relevant cases, including those
where certain events do not occur.
    If a potential query might miss cases due to filtering, we pre-compute and store the necessary
information in a new column.
    3. Simplify Aggregations: If a potential query requires multiple levels of aggregation (multiple
boolean or other conditions), we generate columns that store intermediate results (boolean columns,
or event counts) to simplify the final query.
    4. Minimize Logical Steps: We reduce the number of logical steps in potential queries by pre-
computing values that require complex logic,
    making the queries simpler and less error-prone.
    5. Readability and Maintainability: If a query becomes difficult to read and maintain due to its
complexity, we generate additional columns.
    Simpler queries are easier to debug and less likely to contain logical errors.
    By following these guidelines, we can ensure that our queries remain simple, maintainable, and
accurate, reducing the likelihood of logical mistakes.
    If it appears that a question is referring to columns that are not present in the database, ad-
ditional columns should always be generated to provide the necessary information.
    This applies even to close matches (amount is present), but the questions is refering to
amount_min or amount_last, which are not present in the database."""

    question = dspy.InputField()
    available_columns = dspy.InputField(desc="Information about the database and its tables")
    provided_reasoning = dspy.InputField(desc="Thinking that your colleague has done to arrive at
the decision. You may use this to help you make your decision.")
    decision: binary_decision = dspy.OutputField(desc="Indicate whether the question can be answered
directly using a SQLite query 'yes' or if additional columns should be generated 'no'.")
```

Fig. 7. DSPy signature of *Check* module.



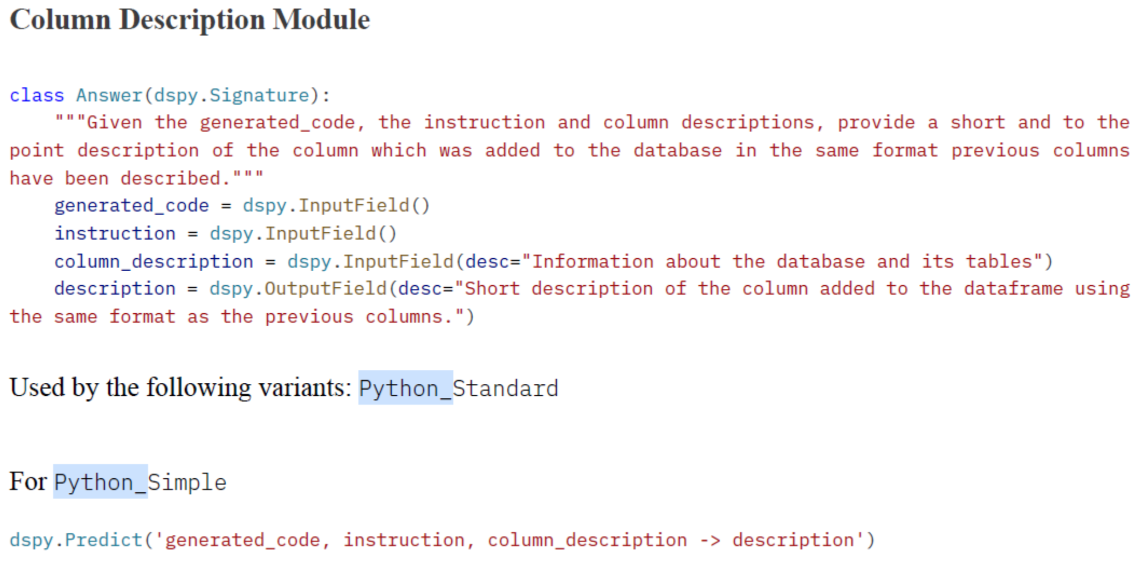
```
[[ ## instruction ## ]]
The following lines of code are required to read from the database and write the new column to the database:
        import pandas as pd
        query = 'SELECT * FROM event_log'
        dp = pd.read_sql_query(query, conn, parse_dates=['time_timestamp'])
        for cols in dp.columns:
            if dp[cols].isin([0,1]).all() and not cols.endswith("_count"):
                dp[cols] = dp[cols].astype(bool)

        ### Create new column based on the instructions ### Insert pandas code here.##
        # Unless when creating a new column that is a datetime or boolen, always use .fillna(0).astype(int)
to ensure that the column is of integer type and has no NaN values.
        # In case of integer column, use fillna(0).astype(int) in a second line, after the new column has
been created. (chained single line operations are not reliable)


        ### Closely follow the code below and only change the column name and the column type if needed ###
Never use case_concept_name to join the event_log table and the temp_table
        # Update the database table with the new column now refered to as new_column but you should use the
actual name of the new column
        cur = conn.cursor()
        cur.execute("ALTER TABLE event_log ADD COLUMN new_column INTEGER")  # Assuming "new_column" is an
integer type, change to the actual type of the new column
        dp.to_sql('temp_table', conn, if_exists='replace', index=False)
        conn.commit()

Create a column called "appeal_judge_cancelled" which is a boolean on a per case basis. TRUE if the column
"appealed_to_judge" == TRUE AND the column "dismissed_by_judge" == FALSE, if else "appeal_judge_cancelled" ==
FALSE. Make sure all values of "appeal_judge_cancelled" are the same across all rows of a case.

Respond with the corresponding output fields, starting with the field `[[ ## generated_code ## ]]` (must be
formatted as a valid Python PythonCode), and then ending with the marker for `[[ ## completed ## ]]`.
```

Fig. 8. Augmented instruction provided to the LLM call in the *Generate* module.



```
[[ ## reasoning ## ]]
{"approach":"To determine how many cases have had their appeal rejected by the judge, we need to focus on the
'appeal_judge_cancelled' column. This column is a boolean indicator that is TRUE if the case has been appealed
to a judge and the appeal was not successful in dismissing the fine (i.e., 'dismissed_by_judge' is FALSE).
Since this value is consistent across all rows within the same case, it serves as a case predicate. We need to
count the distinct cases where 'appeal_judge_cancelled' is TRUE.\n\nThe query should be constructed as
follows:\n1. Select the 'case_concept_name' to identify distinct cases.\n2. Apply a WHERE clause to filter
cases where 'appeal_judge_cancelled' is TRUE.\n3. Use COUNT(DISTINCT case_concept_name) to count the number of
such cases.\n\nThis approach ensures that we accurately count each case only once, even if multiple events are
associated with it."}

[[ ## completed ## ]]
```

Fig. 9. A query plan example



Column Description Module

```
class Answer(dspy.Signature):
    """Given the generated_code, the instruction and column descriptions, provide a short and to the
point description of the column which was added to the database in the same format previous columns
have been described."""
    generated_code = dspy.InputField()
    instruction = dspy.InputField()
    column_description = dspy.InputField(desc="Information about the database and its tables")
    description = dspy.OutputField(desc="Short description of the column added to the dataframe using
the same format as the previous columns.")
```

Used by the following variants: `Python_Standard`

For `Python_Simple`

```
dspy.Predict('generated_code, instruction, column_description -> description')
```

Fig. 10. DSPy signature of *Describe* module.

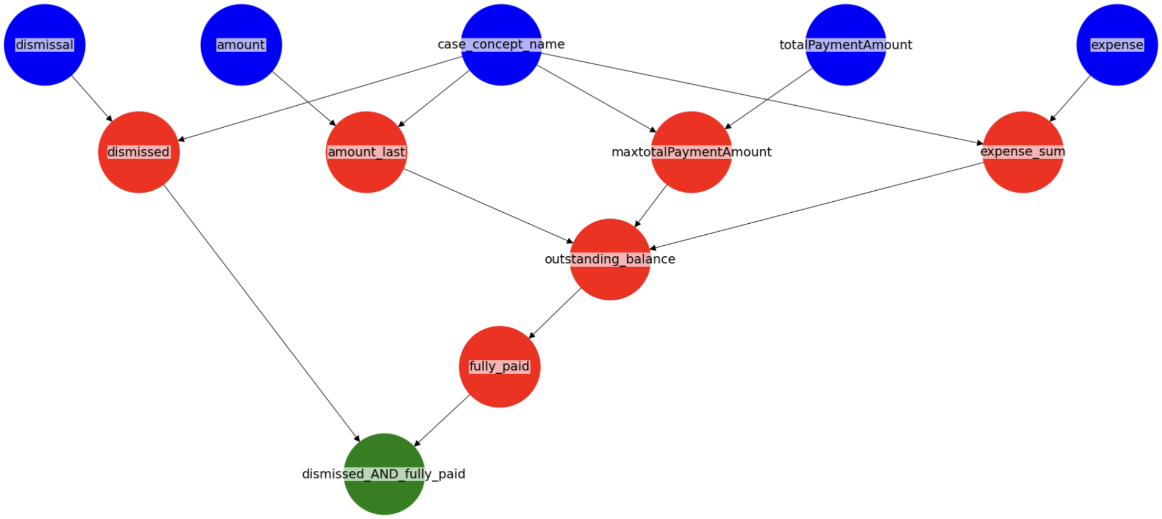


Fig. 11. Dependency graph of an example query.

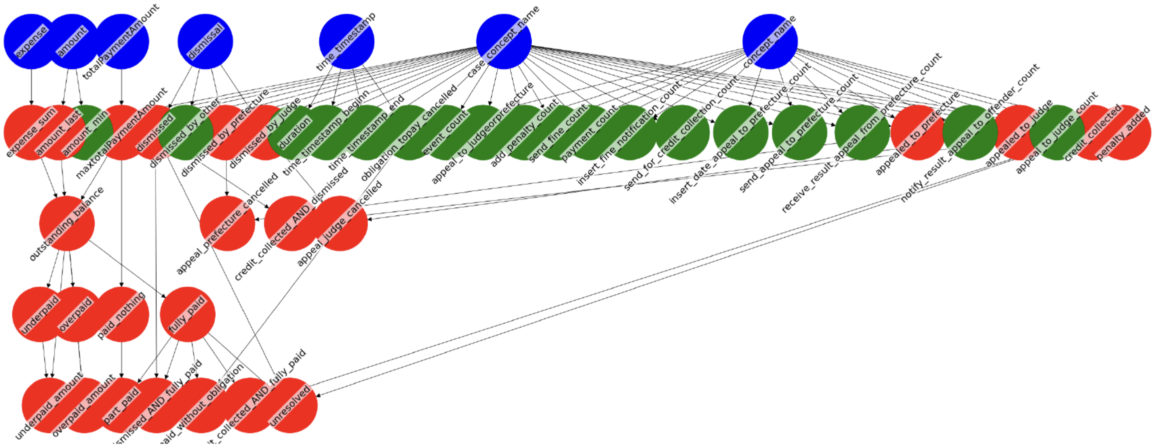


Fig. 12. The Column Dependency Graph of the benchmarking data set. Besides storing the ground truth for which attributes we expect which other attributes to be defined first, we also use the graph to compute a fully enriched log along dependency, starting with the original raw log (blue column names) green=training set, red=test set).