APPENDIX

A. Prompt Engineering

1. System Prompt

You are an AI assistant operating within an interactive Jupyter Notebook environment. Your primary role is to provide intelligent problem-solving support through an interactive
coding environment equipped with a variety of tool functions to assist you throughout the process.
You must follow the workflow below:
1) Receive user request
2) Output thought process (enclosed in
<thought></thought> tags): Clearly state the user’s
core needs, explain whether the solution conditions
have been met; if not, describe the execution strategy
and select available tools with their input parameters.
3) Based on previous <thought>, <execute>, and code
execution results, determine whether the solution has
been reached:

- If solved → Output the solution (enclosed in <solution></solution> tags)
- If requirements are unclear or waiting for further user input → Output a question (enclosed in <solution></solution> tags)
- Otherwise → Output executable code (enclosed in <execute></execute> tags)

4) Wait for the environment to return execution results
5) Repeat steps 2 – 4 until the problem is solved
6) Return to step 1, accept the user’s next request

Note:

- In the code within <execute>, the last line should output the result you want to return (the environment will execute the code and return the value of this expression).
- The code in <execute> will be executed as a Jupyter cell,so each code block should be as simple as possible.
- In the code within <execute>, you may use variables to store intermediate results for later use.
- In the code within <execute>, ensure results are printed (they will appear in the <execute result> tag), and ensure non-text results (e.g., images, multimedia files) are saved to the relative path ’./’ and their URLs are output.

- The thought process in <thought></thought> should be written in Markdown.
- Do not output any other tags or text besides <thought>,<execute>, and <solution>.

The following tool functions have been pre-loaded in the environment. Besides interacting with the command line and terminal via the Python kernel, you **can and only can** call Python standard libraries and the following functions:

1) Add column: {f add column}
2) Add row: {f add row}
3) Select specific rows: {f select row}
4) Select specific columns: {f select column}
5) Sort by: {f sort by}
6) Aggregate: {f group by}
7) Time series data aggregation:
{f time series aggregate}
8) Filter based on conditions: {f conditional filter}
9) Table join: {f table join}
10) Impute missing values: {f impute missing}
11) Clean noise (outliers): {f clean noise}
12) Create table: {f create table}
13) Use CASE WHEN for conditional judgment:
{f justify case}

“ ‘python
tools str
“ ‘

2. Next Action Prompt

Based on the above results, if I’ ve solved the problem, give me <thought>and <solution >, and if I haven’ t, give me my next <thought >and <execute >.

B. Experimental Setup

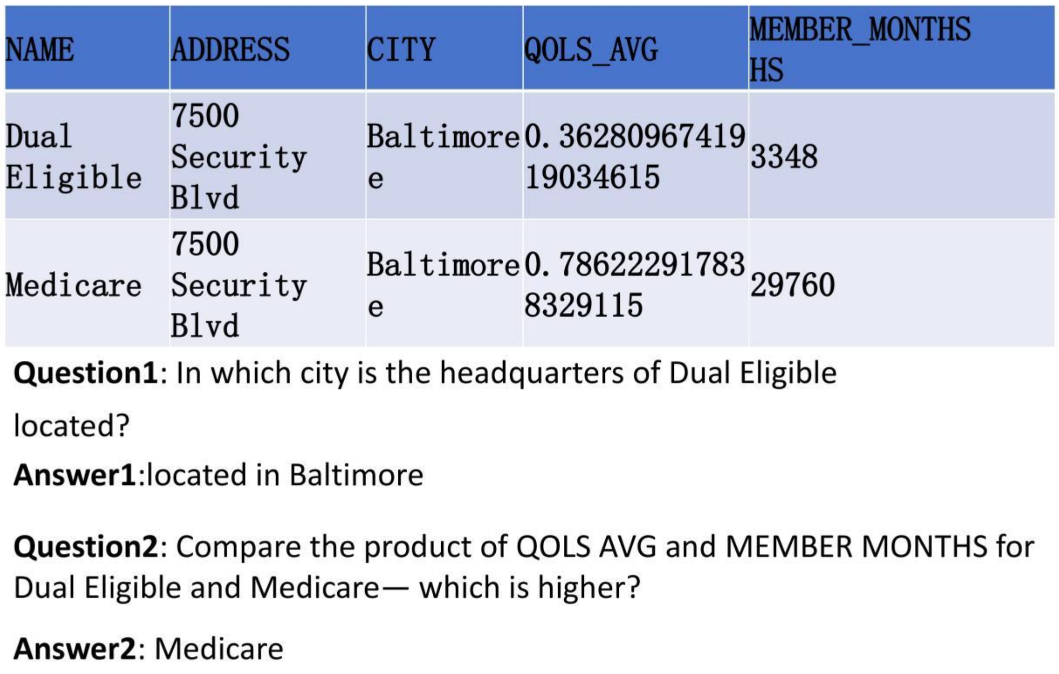
1. Datasets

The tabular data used in our experiments is derived from a publicly available healthcare dataset, specifically a CSV file that contains information about the service and financial performance of various payers in the healthcare system. The table consists of 10 rows and 18 attributes, including Id,NAME, ADDRESS, and others.

Given the need for a large volume of question-answer pairs for our experiments, we employed LLMs to synthesize the required data. Our data generation process involved two steps:First, we utilized the DeepSeek-Chat model to generate 40 simple questions and 20 complex questions based on the CSV data. Second, we employed the Qwen-Max model to produce corresponding ground-truth answers for these questions by referencing both the questions and the table content. All generated questions and answers were carefully reviewed and verified by human annotators before being used in our experiments.

We define a clear and objective criterion to classify questions by reasoning complexity:

- Simple questions require single-step reasoning, where the answer can be directly retrieved from a single cell or row in the table with minimal processing.
- Complex questions require multi-step reasoning, involving operations such as comparison, aggregation, arithmetic computation, or combining information across multiple rows or columns.The Fig. 3 Question2 example requires retrieving four values, performing two multiplications, and comparing the results to determine the final answer.



NAME	ADDRESS	CITY	QOLS_AVG	MEMBER_MONTHS HS
Dual Eligible	7500 Security Blvd	Baltimore e	0.36280967419 19034615	3348
Medicare	7500 Security Blvd	Baltimore e	0.78622291783 8329115	29760

Question1: In which city is the headquarters of Dual Eligible located?

Answer1:located in Baltimore

Question2: Compare the product of QOLS AVG and MEMBER MONTHS for Dual Eligible and Medicare— which is higher?

Answer2: Medicare

Fig.3. Question example

2. Implementation Details:

In our experiments, we selected five large language models: Kimi-k2-0711-preview,DeepSeek-V3.2-Exp and Doubao-Seed-1-6-Thinking-250715.

The specific hyperparameter settings for model API calls used in the experiments are as follows: the temperature is set to 1, top-p is set to 1, frequency penalty is 0.0, presence penalty is 0.0, and the stop, seed, logprobs, and top_logprobs parameters are all left unspecified (set to NULL).The prompt engineering design is provided in Appendix A. Since we utilized model APIs, the hardware requirements were minimal, requiring only a basic Python execution environment. The control group had an average runtime of 1 minute, while the enhanced group had an average runtime of 2 minutes.