

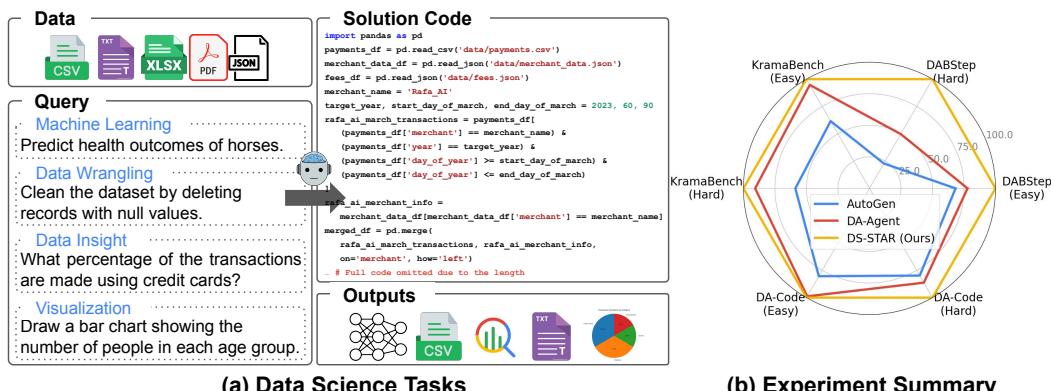
000 DS-STAR: DATA SCIENCE AGENT VIA ITERATIVE 001 PLANNING AND VERIFICATION 002

003 **Anonymous authors**
004

005 Paper under double-blind review
006

007 ABSTRACT 008

009 Data science, which transforms raw data into actionable insights, is critical for
010 data-driven decision-making. However, these tasks are often complex, involving
011 steps for exploring multiple data sources and synthesizing findings to deliver in-
012 sightful answers. While large language models (LLMs) show significant promise
013 in automating this process, they often struggle with heterogeneous data formats
014 and generate sub-optimal analysis plans, as verifying plan sufficiency is inherently
015 difficult without ground-truth labels for such open-ended tasks. To overcome
016 these limitations, we introduce DS-STAR, a novel data science agent. Specifically,
017 DS-STAR makes three key contributions: (1) a data file analysis module that
018 automatically explores and extracts context from diverse data formats, including
019 unstructured types; (2) a verification step where an LLM-based judge evaluates
020 the sufficiency of the analysis plan at each stage; and (3) a sequential planning
021 mechanism that starts with a simple, executable plan and iteratively refines it
022 based on the DS-STAR’s feedback until its sufficiency is verified. This iterative
023 refinement allows DS-STAR to reliably navigate complex analyses involving di-
024 verse data sources. Our experiments show that DS-STAR achieves state-of-the-art
025 performance across three challenging benchmarks: DABStep, KramaBench, and
026 DA-Code. Moreover, DS-STAR particularly outperforms baselines on hard tasks
027 that require processing multiple data files with heterogeneous formats.
028



044 Figure 1: (a) Data science tasks require processing data files in various formats (e.g., csv, txt, xlsx,
045 md) to answer user queries, such as data analysis for extracting useful insights or predictive tasks
046 using machine learning. Data science agents are designed to accomplish this by writing code scripts
047 (e.g., Python) and answering based on the output from their execution. In addition to the solution
048 code, the output may include a trained model for prediction tasks, processed databases, text-formatted
049 answers, visual charts, and more. (b) We report normalized accuracy (%) for both easy (answer can
050 be found within a *single* file) and hard (requires processing *multiple* files) tasks in the DABStep,
051 KramaBench, and DA-Code benchmarks. DS-STAR consistently and significantly outperforms
052 competitive baselines, particularly in challenging hard tasks, showing DS-STAR’s superiority in
053 handling multiple data sources in heterogeneous formats.

054 **1 INTRODUCTION**

055

056 The data science, which transforms raw data into actionable insights, is critical to solving real-world
 057 problems (De Bie et al., 2022; Hassan et al., 2023); for instance, businesses rely on insights extracted
 058 from data to make crucial directional decisions (Sun et al., 2018; Sarker, 2021). However, the data
 059 science workflow is often complex, requiring deep expertise across diverse fields such as computer
 060 science and statistics (Zhang et al., 2023; Hong et al., 2024). This workflow involves a series of time-
 061 consuming tasks (Zhang et al., 2024; Martin Iglesias, 2025), ranging from understanding distributed
 062 documents to intricate data processing and statistical analyses (see Figure 1).

063 To simplify this demanding workflow, recent research has explored using large language models
 064 (LLMs) as autonomous data science agents (Hong et al., 2024). These agents aim to translate natural
 065 language questions directly into functional code that generates corresponding answers (Ouyang
 066 et al., 2025; Wu et al., 2023; Yang et al., 2025; Yao et al., 2023). While representing significant
 067 advancements, current data science agents face several challenges that limit their practical utility and
 068 impact. A primary limitation is their predominant focus on well-structured data, such as relational
 069 databases composed of CSV files (Pourreza et al., 2024; Yu et al., 2018; Li et al., 2024). This narrow
 070 scope neglects the wealth information available in the heterogeneous data formats encountered in
 071 real-world scenarios, which can range from JSON to unstructured text and markdown files (Lai et al.,
 072 2025). Furthermore, as many data science tasks are framed as open-ended questions where ground-
 073 truth labels do not exist, verifying an agent’s reasoning path is a non-trivial challenge. For example,
 074 most agents, like Data Interpreter (Hong et al., 2024), terminate their process upon successful code
 075 execution; however, executable code does not always guarantee a correct answer. Consequently, this
 076 lack of robust verification often leads an agent to adopt a sub-optimal plan for solving the given task.

077 To address these limitations, we introduce **DS-STAR**, a novel agent that tackles data science problems
 078 by generating solution code through a robust, iterative process of planning and verification. The
 079 proposed DS-STAR framework consists of two key stages. First, to ensure adaptability across
 080 diverse data types, DS-STAR automatically analyzes all files within a given directory, generating
 081 a textual summary of their structure and content. This summary then serves as a crucial context
 082 for the DS-STAR’s approach to the given task. Second, DS-STAR enters a core loop of planning,
 083 implementation, and verification. It begins by formulating a high-level plan, implementing it as a
 084 code script, and then verifying to solve the problem. Notably, for verification, we propose using an
 085 LLM-based judge (Gu et al., 2024; Zheng et al., 2023) that is prompted to explicitly evaluate whether
 086 the current plan is sufficient to solve the problem. If the verification fails (*i.e.*, the judge determines
 087 the plan insufficient), DS-STAR refines its plan by adding or modifying steps and repeats the process.
 088 Crucially, rather than creating a complete plan at once, our agent operates sequentially, reviewing the
 089 results from each intermediate step before building upon them.¹ This iterative loop continues until a
 satisfactory plan is verified, at which point the corresponding code is returned as the final solution.

090 We validate the effectiveness of our proposed DS-STAR through comprehensive evaluations on
 091 a suite of established data science benchmarks. These benchmarks, including DABStep bench-
 092 mark (Martin Iglesias, 2025), KramaBench (Lai et al., 2025), and DA-Code (Huang et al., 2024) are
 093 specifically designed to assess performance on complex data analysis tasks (*e.g.*, data wrangling,
 094 machine learning, visualization, exploratory data analysis, data manipulation, etc) involving multiple
 095 data sources and formats (see Section 4). The experimental results demonstrate that DS-STAR sig-
 096 nificantly outperforms existing state-of-the-art methods across all evaluated scenarios. Specifically,
 097 compared to the best alternative, DS-STAR improves accuracy from 41.0% to 45.2% on the DABStep
 098 benchmark, from 39.8% to 44.7% on KramaBench, and from 37.0% to 38.5% on DA-Code.

099 The contributions of this paper can be summarized as follows:

100

- 101 • We introduce DS-STAR, a novel data science agent which operates on data with various formats
 102 and on various tasks like data analysis, machine learning, visualization, and data wrangling, etc.
- 103 • We propose an LLM-based verification module that judges the sufficiency of a solution plan.
- 104 • We develop an iterative refinement strategy, enabling DS-STAR to sequentially build its solution.
- 105 • We provide a comprehensive evaluation on challenging benchmarks, showing DS-STAR’s efficacy.

106 ¹This mirrors the interactive process of an expert using a Jupyter notebook to observe interim results.

108
109

2 RELATED WORK

110
111
112
113
114
115
116
117
118
119
120

LLM agents. Recent advancements in LLMs have spurred significant research into autonomous agents designed to tackle complex and long-horizon tasks. A key strategy employed by these multi-agent systems is the autonomous decomposition of a main objective into a series of smaller, manageable sub-tasks, which can be delegated to sub-agents. General-purpose agents, such as ReAct (Yao et al., 2023), HuggingGPT (Shen et al., 2023), and OpenHands (Wang et al., 2024), utilize external tools to reason, plan, and act across a wide range of problems. Building on these foundational capabilities, research has increasingly focused on specialized domains such as Voyager (Wang et al., 2023) for navigating the Minecraft environment and AlphaCode (Li et al., 2022) for performing sophisticated code generation. Furthermore, DS-Agent (Guo et al., 2024), AIDE (Jiang et al., 2025), and MLE-STAR (Nam et al., 2025) are tailored specifically for machine learning engineering. In a similar vein, our work, DS-STAR, is a specialized LLM agent for data science tasks.

121
122
123
124
125
126
127
128
129
130
131

Data science agents. Recent research has focused on developing data science agents that leverage the advanced coding and reasoning capabilities of LLMs (Jiang et al., 2025; Jimenez et al., 2024). Initial efforts utilized general-purpose frameworks like ReAct (Yao et al., 2023) and AutoGen (Wu et al., 2023). Following these pioneering approaches, agents like DA-Agent (Huang et al., 2024), which are specialized in autonomously generating code-based solutions for tasks such as data analysis and visualization (Hu et al., 2024; Jing et al., 2025), has been developed. A notable example is Data Interpreter (Hong et al., 2024), which employs a graph-based method to decompose a primary task into a series of manageable sub-tasks and the resulting task graph is then progressively refined based on the successful execution of these sub-tasks. However, a key limitation is that relying solely on successful execution as feedback often results in sub-optimal plans, since such feedback cannot confirm the plan’s correctness. To address this, DS-STAR introduces a novel verification process that employs an LLM as a judge (Gu et al., 2024) to assess the quality of the generated solutions.

132
133
134
135
136
137
138
139
140
141
142
143
144

Text-to-SQL. The task of Text-to-SQL involves translating natural language questions into executable SQL queries (Li & Jagadish, 2014; Li et al., 2023). Early approaches relied on sequence-to-sequence architectures, which jointly encoded the user’s query and the database schema (Cao et al., 2021; Choi et al., 2021). However, after the rise of LLMs, initial research focused on prompt engineering (Pourreza & Rafiei, 2023). This was soon followed by the development of more sophisticated, multi-step pipelines incorporating techniques such as schema linking, self-correction, and self-consistency to enhance generation accuracy (Talaei et al., 2024; Pourreza et al., 2024; Maamari et al., 2024). While these pipelines function as specialized agents for data analysis, their fundamental reliance on generating SQL restricts their use to structured, relational databases. In contrast, DS-STAR addresses this limitation by adopting a Python language, thereby framing our approach as a Text-to-Python task. To further facilitate this, we introduce a novel data file summarization mechanism which enables our method to demonstrate significantly broader applicability than traditional Text-to-SQL systems, as it can operate on a wide array of data formats, including JSON, Markdown, and unstructured text.

145
146
147

3 DS-STAR

148
149
150
151
152

In this section, we introduce DS-STAR, a framework for data science agents that leverages the coding and reasoning capabilities of LLMs to tackle data science tasks. In a nutshell, our approach first analyzes heterogeneous input data files to extract essential information (Section 3.1). Subsequently, DS-STAR generates a code solution through an iterative process (Section 3.2). The prompts and algorithms are detailed in Appendix H and A, respectively.

153
154
155
156
157
158

Problem setup. Our goal is to automatically generate a code solution s (*i.e.*, a Python script) that answers query q using a given data files \mathcal{D} . We formulate this as a search problem over the space of all possible scripts, denoted by \mathcal{S} . The quality of any script $s \in \mathcal{S}$ is evaluated by a scoring function $h(s)$, which measures the correctness of its output, $s(\mathcal{D})$, against a ground-truth answer (*e.g.*, accuracy) where $s(\mathcal{D})$ represents the output generated by executing s , which loads the data \mathcal{D} to produce an answer. The objective is to identify the s^* that is: $s^* = \arg \max_{s \in \mathcal{S}} h(s(\mathcal{D}))$.

159
160
161

In this paper, we propose a multi-agent framework, \mathcal{A} , designed to process the query q and data files \mathcal{D} , which may be numerous and in heterogeneous formats. This framework is composed of n specialized LLM agents, $\{\mathcal{A}_i\}_{i=1}^n$, where each agent possesses a distinct functionality, as detailed in the subsequent sections (see Figure 2 for the problem setup and our agent’s overview).

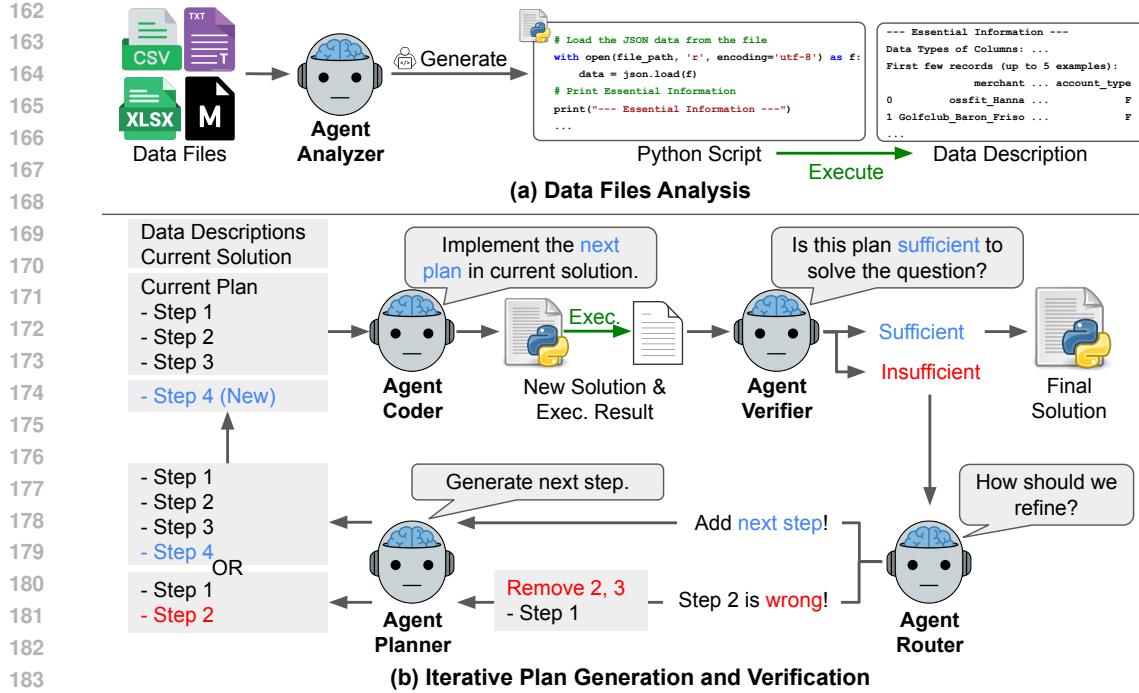


Figure 2: **Overview of DS-STAR.** (a) DS-STAR generates a Python script that analyzes heterogeneous input data files. (b) Starting with a simple single step, DS-STAR implements the plan and executes it to obtain intermediate results. Next, using a verifier agent, DS-STAR determines whether the current plan is sufficient to solve the user’s query. If it is not sufficient, DS-STAR’s router agent decides whether to add the next step or remove the incorrect step from the plan. Finally, DS-STAR adds the next step and implements it again. This iterative process of planning, implementation, execution, and verification is repeated until the verifier agent determines that the plan is sufficient to answer the user’s query or until the maximum number of iterations is reached.

3.1 ANALYZING DATA FILES

To effectively interact with the given data files, DS-STAR first requires a comprehensive understanding of its contents and structure. We achieve this by generating a concise, analytical description for each data file. This initial analysis phase is critical as it informs all subsequent actions taken by the agent.

Conventional frameworks for data science agents often rely on displaying a few sample rows from structured files, such as CSVs. However, this approach is fundamentally limited to structured data and fails for unstructured formats where the concept of a ‘row’ is ill-defined. To overcome this limitation, we introduce a more general mechanism as follows. For each data file $\mathcal{D}_i \in \mathcal{D}$ we employ an analyzer agent, $\mathcal{A}_{\text{analyzer}}$, to generate a Python script, s_{desc}^i . This script is designed to correctly parse and load the data file \mathcal{D}_i and then extract its essential properties. For structured data, this might include column names and data types; for unstructured data, it could be file metadata, text summaries. The resulting analytical description, d_i , is captured directly from the script’s execution output (see Appendix D). This process, which can be parallelized, is denoted as: $d_i = \text{exec}(s_{\text{desc}}^i)$, $s_{\text{desc}}^i = \mathcal{A}_{\text{analyzer}}(\mathcal{D}_i)$.

3.2 ITERATIVE PLAN GENERATION AND VERIFICATION

Initialization. After generating analytic descriptions of the N data files, DS-STAR begins the solution generation process. First, a planner agent, $\mathcal{A}_{\text{planner}}$, generates an initial high-level executable step p_0 (e.g., loading a data file) using the query q and obtained data descriptions: $p_0 = \mathcal{A}_{\text{planner}}(q, \{d_i\}_{i=1}^N)$.

This single executable step is then implemented as a code script s_0 by a coder agent $\mathcal{A}_{\text{coder}}$ and the initial execution result r_0 is then obtained by executing the script s_0 as follows:

$$s_0 = \mathcal{A}_{\text{coder}}(p_0, \{d_i\}_{i=1}^N), r_0 = \text{exec}(s_0). \quad (1)$$

216 **Plan verification.** The main challenge in data science tasks is guiding the refinement of a solution,
 217 as determining its correctness is often non-trivial, since there are no ground-truth label. To address
 218 this, we use an LLM as a judge to assess whether the current plan is sufficient for the user’s query.
 219

220 Our approach introduces a verifier agent $\mathcal{A}_{\text{verifier}}$. At any given round k in the problem-solving
 221 process, this agent evaluates the state of the solution, *i.e.*, whether the plan is sufficient to solve the
 222 problem. The evaluation is based on the cumulative plan $p = \{p_0, \dots, p_k\}$, the user’s query q , the
 223 current solution code s_k , which is an implementation of the cumulative plan, and its execution result
 224 r_k . The operation of $\mathcal{A}_{\text{verifier}}$ is denoted as follows: $v = \mathcal{A}_{\text{verifier}}(p, q, s_k, r_k)$.
 225

226 Here, the output v is a binary variable: `sufficient` or `insufficient`. Note that our method does
 227 not just compare the plan to the query. By conditioning the judgement on s_k and its execution output
 228 r_k , $\mathcal{A}_{\text{verifier}}$ can provide more grounded feedback since it assesses whether s_k is well-implemented
 229 following the plan, and whether the r_k contains the information needed to fully address the query.
 230

231 **Plan refinement.** If the verifier agent $\mathcal{A}_{\text{verifier}}$ determines that the current plan is insufficient to solve
 232 the user’s query q , DS-STAR must decide how to proceed. Such insufficiency could arise because the
 233 plan is merely incomplete and requires additional steps, or because it contains erroneous steps that
 234 invalidate the approach. To resolve this, DS-STAR employs a router agent, $\mathcal{A}_{\text{router}}$, which decides
 235 whether to append a new step or to correct an existing one. The router’s decision w is generated as
 236 follows, where $p = \{p_0, \dots, p_k\}$ is the current cumulative plan: $w = \mathcal{A}_{\text{router}}(p, q, r_k, \{d_i\}_{i=1}^N)$.
 237

238 The output w is either the token `Add Step` or an index $l \in \{1, \dots, k\}$. If w is `Add Step`, $\mathcal{A}_{\text{router}}$
 239 has determined the plan is correct but incomplete. In this case, we retain the plan p and proceed
 240 to generate the next step. On the other hand, if $w = l$, $\mathcal{A}_{\text{router}}$ has identified p_l as erroneous. In
 241 this case, we backtrack by truncating the plan to $p \leftarrow \{p_0, \dots, p_{l-1}\}$. Here, we deliberately choose
 242 to truncate and regenerate through the LLM’s random sampling, rather than directly correcting p_l ,
 243 since our empirical finding has shown that revising a specific incorrect step often leads to an overly
 244 complex replacement, therefore frequently flagged again by $\mathcal{A}_{\text{router}}$ in a next iteration. Following the
 245 decision from the $\mathcal{A}_{\text{router}}$, our agent proceeds with an updated plan $p = \{p_0, \dots, p_{k'}\}$, where $k' = k$
 246 or $k' = l - 1$. Then DS-STAR generates a subsequent step: $p_{k'+1} = \mathcal{A}_{\text{planner}}(p, q, r_k, \{d_i\}_{i=1}^N)$.
 247

248 Notably, the planner agent $\mathcal{A}_{\text{planner}}$ is conditioned on the last execution result, r_k , enabling it to
 249 generate a step that attempts to resolve the previously identified insufficiency. Once the new step
 250 $p_{k'+1}$ is defined, the plan is updated to: $p \leftarrow \{p_0, \dots, p_{k'}, p_{k'+1}\}$.
 251

252 **Plan implement and execution.** Finally, DS-STAR enters an execution and verification cycle. First,
 253 $\mathcal{A}_{\text{coder}}$ implements p into code s . The execution of this code yields a new observation $r = \text{exec}(s)$.
 254 With this r , $\mathcal{A}_{\text{verifier}}$ is invoked again to assess if the newly augmented plan is now sufficient. This
 255 entire iterative procedure—routing, planning, coding, executing, and verifying—is repeated until
 256 $\mathcal{A}_{\text{verifier}}$ returns a `sufficient` or a predefined maximum number of iterations is reached.
 257

258 3.3 ADDITIONAL MODULES FOR ROBUST DATA SCIENCE AGENTS

259 **Debugging agent.** When a Python script s fails during execution, it generates an error traceback
 260 \mathcal{T}_{bug} . To automatically debug the script, DS-STAR employs a debugging agent, $\mathcal{T}_{\text{debugger}}$. First,
 261 when generating $\{d_i\}_{i=1}^N$ using s_{desc} obtained from $\mathcal{A}_{\text{analyzer}}$ (see Section 3.1), $\mathcal{A}_{\text{debugger}}$ iteratively
 262 update the script using only the traceback: $s_{\text{desc}} \leftarrow \mathcal{A}_{\text{debugger}}(s_{\text{desc}}, \mathcal{T}_{\text{bug}})$.
 263

264 Secondly, once DS-STAR obtains $\{d_i\}_{i=1}^N$, $\mathcal{A}_{\text{debugger}}$ utilizes such information when generating a
 265 solution Python script s (see Section 3.2). Our key insight is that tracebacks alone are often insufficient
 266 for resolving errors in data-centric scripts, while $\{d_i\}_{i=1}^N$ might include critical metadata such as
 267 column headers in a CSV file, sheet names in an Excel workbook, or database schema information.
 268 Therefore, $\mathcal{A}_{\text{debugger}}$ generates a corrected script, s , by conditioning on the original script s , the error
 269 traceback \mathcal{T}_{bug} , and this rich data context $\{d_i\}_{i=1}^N$: $s \leftarrow \mathcal{A}_{\text{debugger}}(s, \mathcal{T}_{\text{bug}}, \{d_i\}_{i=1}^N)$.
 270

271 **Retriever.** A potential scalability challenge arises when the number of data files N is large (*i.e.*,
 272 $N > 100$). In such cases, the complete set of descriptions $\{d_i\}_{i=1}^N$ cannot be prompted within the
 273 predefined context length of LLMs. To address this limitation, we employ a retrieval mechanism that
 274 leverages a pre-trained embedding model (Nie et al., 2024). Specifically, we identify the top- K most
 275 relevant data files, which will be provided as context to the LLM, by computing the cosine similarity
 276 between the embedding of the user’s query q and the embedding of each description d_i .
 277

270 Table 1: **Main results from DABStep.** All results are taken from the DABStep leaderboard (Martin
 271 Iglesias, 2025), except for the model marked with \dagger . The highest scores are shown in **bold**.

273 Framework	274 Model	275 Easy-level 276 Accuracy (%)	277 Hard-level 278 Accuracy (%)
275 Model-only	276 Gemini-2.5-Pro o4-mini	277 66.67 76.39	278 12.70 14.55
277 ReAct (Yao et al., 2023)	278 Claude-4-Sonnet Gemini-2.5-Pro \dagger	279 81.94 69.44	280 19.84 10.05
280 AutoGen (Wu et al., 2023)	281 Gemini-2.5-Pro \dagger	282 59.72	283 10.32
281 Data Interpreter (Hong et al., 2024)	282 Gemini-2.5-Pro \dagger	283 72.22	284 3.44
282 DA-Agent (Huang et al., 2024)	283 Gemini-2.5-Pro \dagger	284 68.06	285 22.49
283 Open Data Scientist	284 DeepSeek-V3	285 84.72	286 16.40
284 Mphasis-I2I-Agents	285 Claude-3.5-Sonnet	286 80.56	287 28.04
285 Amity DA Agent	286 Gemini-2.5-Pro	287 80.56	288 41.01
286 DS-STAR (Ours)	287 Gemini-2.5-Pro \dagger	288 87.50	289 45.24

286 4 EXPERIMENTS

288 In this section, we conduct a comprehensive empirical evaluation of DS-STAR on three challenging
 289 data science benchmarks: DABStep (Martin Iglesias, 2025), KramaBench (Lai et al., 2025), and
 290 DA-Code (Huang et al., 2024). Our results show that DS-STAR significantly outperforms competitive
 291 baselines, including those built upon various LLMs (Section 4.1). To understand the source of gains,
 292 we perform a detailed ablation study, which confirms that our proposed analyzer and router agents
 293 are critical components that provides a substantial performance improvement (Section 4.2).

294 **Common setup.** Unless otherwise specified, all experiments are conducted using Gemini-2.5-Pro
 295 as the base LLM. Our agent, DS-STAR, operates for a maximum of 20 rounds per task. To ensure
 296 properly formatted output, we employ a finalizer agent, $\mathcal{A}_{\text{finalizer}}$, which takes formatting guidelines
 297 (*e.g.*, rounding to two decimal places) and generates the final solution code (see Appendix H).

299 4.1 MAIN RESULTS

301 **DABStep.** We evaluate DS-STAR on the DABStep benchmark (Martin Iglesias, 2025), which is
 302 designed to mirror real-world data analysis challenges by requiring the processing of seven diverse
 303 data files, including formats like JSON, Markdown, and CSV. The benchmark features two difficulty
 304 levels; its hard-level tasks are particularly demanding, necessitating the analysis of multiple data
 305 files and the application of domain-specific knowledge. As shown in Table 1, existing single-
 306 LLM approaches struggle to surpass 20% accuracy, which underscores the significant room for
 307 improvement. Additionally, its authors note that a human-reference solution for a hard-level task
 308 spans 220 lines of code and is broken into four sequential steps. Detailed statistics and the full
 309 execution logs of DS-STAR are provided in Appendix B and G, respectively.

310 The results, presented in Table 1, demonstrate that DS-STAR significantly outperforms all baselines.
 311 For instance, integrating DS-STAR with Gemini-2.5-Pro boosts the hard-level accuracy from 12.70%
 312 to 45.24%, an absolute improvement of over 32 percentage points. Notably, the DS-STAR using
 313 Gemini-2.5-Pro substantially surpasses other commercial agents like Open Data Scientist, Mphasis-
 314 I2I-Agents, and Amity DA Agent across both easy and hard difficulty levels. Crucially, DS-STAR also
 315 outperforms other multi-agent systems built upon the same Gemini-2.5-Pro, highlighting that the
 316 performance gains stem directly from the proposed effective sub-agent orchestration.

317 **KramaBench.** We additionally validate DS-STAR’s capability on Kramabench (Lai et al., 2025),
 318 a benchmark that requires data discovery, *i.e.*, selecting the correct data files from a vast data lake
 319 to address a user’s query. For example, the Astronomy domain includes over 1,500 files, while
 320 only a small subset is relevant to any single query (see Appendix B for the detailed statistics). To
 321 navigate this, DS-STAR integrates a retrieval module that identifies the top 100 candidate files
² 322 based on embedding similarity between the user’s query and data descriptions, computed with
 323 Gemini-Embedding-001 (Lee et al., 2025) (details can be found in Section 3.3).

²If the total data is less than 100, we fully utilized all the data for the task.

324 **Table 2: Main results from KramaBench.** All results are taken from the original paper (Lai et al.,
 325 2025), except the results with Gemini-2.5-Pro. The highest scores are shown in **bold**.

327 Framework	328 Model	329 Domains						330 Total
331	332	Archaeology	Astronomy	333 Biomedical	Environment	Legal	Wildfire	334
<i>Original experimental setting for which the relevant data must be retrieved.</i>								
Model-only	o3	25.00	1.73	3.50	1.35	3.35	24.87	9.64
	GPT-4o	0.00	1.41	1.98	0.45	1.46	1.45	1.62
	Claude-3.5-Sonnet	16.67	1.62	2.87	1.17	7.33	13.63	7.45
DS-GURU	o3	25.00	3.53	8.95	19.60	13.89	50.73	22.08
	GPT-4o	16.67	2.76	8.97	2.60	2.80	17.18	8.28
	Claude-3.5-Sonnet	16.67	1.52	1.96	11.21	7.01	39.16	14.35
ReAct	Gemini-2.5-Pro	16.67	4.77	3.69	26.17	41.95	51.40	30.31
AutoGen	Gemini-2.5-Pro	16.67	4.39	7.25	19.38	26.38	41.76	22.83
Data Interpreter	Gemini-2.5-Pro	41.67	12.72	28.05	9.87	30.04	59.67	31.32
DA-Agent	Gemini-2.5-Pro	41.67	15.52	12.59	42.64	39.73	61.61	39.79
DS-STAR (Ours)	Gemini-2.5-Pro	25.00	12.09	43.74	46.75	49.64	65.94	44.69
<i>Oracle experimental setting with relevant data already provided.</i>								
ReAct	Gemini-2.5-Pro	25.00	6.36	3.69	30.70	46.93	51.69	33.82
AutoGen	Gemini-2.5-Pro	25.00	5.29	26.03	23.46	39.28	49.41	31.77
Data Interpreger	Gemini-2.5-Pro	41.67	13.29	28.39	13.09	32.97	63.13	33.57
DA-Agent	Gemini-2.5-Pro	41.67	15.77	44.26	44.55	56.42	65.92	48.61
DS-STAR (Ours)	Gemini-2.5-Pro	25.00	19.08	55.24	58.80	59.66	70.18	52.55

343 **Table 3: Main results from DA-Code.** All results are taken from the original paper (Huang et al.,
 344 2024), except the results with Gemini-2.5-Pro. The highest scores are shown in **bold**.

346 Framework	347 Model	348 Score						349 Total
350	351	352 Data Wrangling	353 ML	354 EDA	355 Easy	356 Medium	357 Hard	358
DA-Agent	Mixtral-8x22B	14.8	31.6	10.2	17.6	16.8	8.6	15.4
	DeepSeek-Coder-V2.5	25.1	34.1	14.7	32.8	18.7	14.1	20.7
	Qwen-2.5-72B	24.9	41.8	15.4	31.9	19.4	22.3	22.6
	Claude-3-Opus	29.3	46.8	20.7	44.7	23.8	19.0	27.6
	GPT-4	30.4	48.4	24.6	45.4	27.8	23.4	30.5
	Gemini-2.5-Pro	34.8	57.2	31.1	50.0	34.2	32.0	37.0
ReAct	Gemini-2.5-Pro	14.7	31.2	22.2	32.0	17.9	25.9	22.5
AutoGen	Gemini-2.5-Pro	25.6	51.6	25.6	38.7	28.5	29.6	30.8
DS-STAR (Ours)	Gemini-2.5-Pro	30.4	57.3	34.8	48.9	35.2	37.1	38.5

359 As detailed in Table 2, DS-STAR demonstrates a substantial performance advantage in this chal-
 360 lening setting. Using Gemini-2.5-Pro, DS-STAR achieves an accuracy of 44.69%, significantly
 361 outperforming the 39.79% score of the state-of-the-art DA-Agent. Moreover, to isolate the impact of
 362 data retrieval, we also evaluated performance in an oracle setting, assuming all relevant data for a task
 363 is already provided. Under these ideal conditions, DS-STAR’s accuracy increases by 8 percentage
 364 points. In contrast, other agentic frameworks like ReAct show only a marginal performance gain
 365 in the same setting. This large gap highlights that while our current retrieval method is effective,
 366 advanced data discovery is a promising direction for unlocking the full potential of DS-STAR.

367 **DA-Code.** To evaluate the generalization capabilities of DS-STAR, we additionally benchmark its
 368 performance on the DA-Code dataset (Huang et al., 2024), which encompasses a diverse array of data
 369 science tasks. Specifically, DA-Code is structured into three main categories: data wrangling, machine
 370 learning (ML), and exploratory data analysis (EDA), with the EDA category further subdivided into
 371 data manipulation, data insights, visualization, and statistical analysis. Here, we found that the
 372 Data-Interpreter (Hong et al., 2024) struggles to generate answers with the right format required by
 373 DA-Code tasks (*e.g.*, saving in csv format), therefore excluded it from the baseline.

374 As shown in Table 3, DS-STAR outperforms the strongest baseline, *i.e.*, DA-Agent, demonstrating
 375 its robust applicability across various data science domains. The superiority of our framework is
 376 particularly highlighted on more complex problems. Specifically, on the hard-difficulty tasks, DS-
 377 STAR achieves 37.1% accuracy, representing a significant improvement over the 32.0% accuracy of
 378 the DA-Agent baseline, even when both frameworks utilize the same Gemini-2.5-Pro. This showcases
 379 the effectiveness of our approach, especially for challenging, real-world data science scenarios. We
 380 provide qualitative results for each task type in Appendix F.

378	Question	What are the top 3 countries (ip_country) by avg transaction value for Crossfit_Hanna's NexPay transactions in the last quarter of 2023?
381	ReAct	<pre> import pandas as pd payments_df = pd.read_csv('data/payments.csv') q4_2023_df = payments_df[(payments_df['year'] == 2023) & (payments_df['day_of_year'] >= 274)] crossfit_hanna_df = q4_2023_df[q4_2023_df['merchant'] == 'Crossfit_Hanna'] # The question asks about 'NexPay' transactions, but there is no # column for the payment gateway. # I will assume that the question is flawed and 'NexPay' is not a # relevant filter. # I will proceed without filtering for 'NexPay'. avg_transaction_value = crossfit_hanna_df.groupby('ip_country')['eur_amount'].mean() top_countries = avg_transaction_value.sort_values(ascending=False).head(3) top_countries_list = top_countries.index.tolist() result = ", ".join(top_countries_list) print(result) </pre>
382	DS-STAR (Ours)	<pre> import pandas as pd payments_file = 'data/payments.csv' payments_df = pd.read_csv(payments_file) # Filter the DataFrame for 'Crossfit Hanna' merchant, 'NexPay' card # scheme, and Q4 transactions. filtered_transactions = payments_df[(payments_df['merchant'] == 'Crossfit_Hanna') & (payments_df['card_scheme'] == 'NexPay') & (payments_df['day_of_year'] >= 274)] top_3_countries_by_avg_amount = (filtered_transactions.groupby('ip_country')['eur_amount'] .mean() .sort_values(ascending=False) .head(3)) answer = ", ".join(top_3_countries_by_avg_amount.index) print(answer) </pre>

394 Figure 3: **Qualitative result.** We observed that while ReAct (Yao et al., 2023) failed to filter out the
395 ‘NexPay’ value, leading to an incorrect answer, DS-STAR successfully filtered out due to the use of
396 data file analysis agent. See Appendix G for the full execution logs of DS-STAR.
397

398 Table 4: **Ablation study on each component.** Effectiveness of each agent and compatibility with
399 other LLMs on the DABStep benchmark (Martin Iglesias, 2025).

Framework	Model	Analyzer $\mathcal{A}_{\text{analyzer}}$	Router $\mathcal{A}_{\text{router}}$	Easy-level Accuracy (%)	Hard-level Accuracy (%)
DS-STAR (Variant 1)	Gemini-2.5-Pro	✗	✓	75.00	26.98
DS-STAR (Variant 2)	Gemini-2.5-Pro	✓	✗	79.17	39.95
DS-STAR (Ours)	Gemini-2.5-Pro	✓	✓	87.50	45.24
DS-STAR (Ours)	GPT-5	✓	✓	88.89	43.12

4.2 ABLATION STUDIES

410 In this section, we conduct ablation studies to verify the effectiveness of individual components.
411 Unless otherwise specified, all experiments use Gemini-2.5-Pro and are performed on the DABStep.

412 **Effectiveness of the data file analysis agent.** The data file analysis agent, $\mathcal{A}_{\text{analyzer}}$, which provides
413 contextual descriptions of data files, is crucial for achieving high performance. When we remove the
414 descriptions, DS-STAR’s accuracy on hard-level tasks in the DABStep benchmark drops significantly
415 to 26.98%. While this result is still substantially better than the 12.70% accuracy of a non-agentic
416 framework, it underscores the importance of the component. We hypothesize that the rich context of
417 the given data is vital for enabling DS-STAR to effectively plan and implement its approach.

418 To qualitatively demonstrate the effectiveness of our data file analysis agent, we present a comparative
419 case study in Figure 3. In this example, the competing baseline, ReAct (Yao et al., 2023), tries to filter
420 data based on a ‘NexPay’ value but fails to execute the request. The model incorrectly determines
421 that no such column value exists in the ‘payment.csv’ file due to ReAct’s limited understanding of
422 the provided data context. On the other hand, DS-STAR succeeds by first employing its data file
423 analysis to correctly understand the entire data structure. This allows it to identify the appropriate
424 column value and accurately filter the data as instructed. Further qualitative examples, including full
425 execution logs, showcasing the capabilities of DS-STAR can be found in Appendix G.

426 **Effectiveness of the router agent.** $\mathcal{A}_{\text{router}}$ determines whether to append a new step or correct an
427 existing one in the plan. Here, we verify its importance against an alternative approach (Variant 2 in
428 Table 4) where $\mathcal{A}_{\text{router}}$ is removed. In this variant, DS-STAR only uses $\mathcal{A}_{\text{planner}}$ to simply add new
429 steps sequentially until the plan is sufficient or a maximum number of iterations is reached. As shown
430 in Table 4, this alternative performs worse on both easy and hard tasks. This is because building
431 upon an erroneous step leads to error accumulation, causing subsequent steps to also be incorrect, *i.e.*,
432 correcting errors in the plan is more effective than accumulating potentially flawed steps.

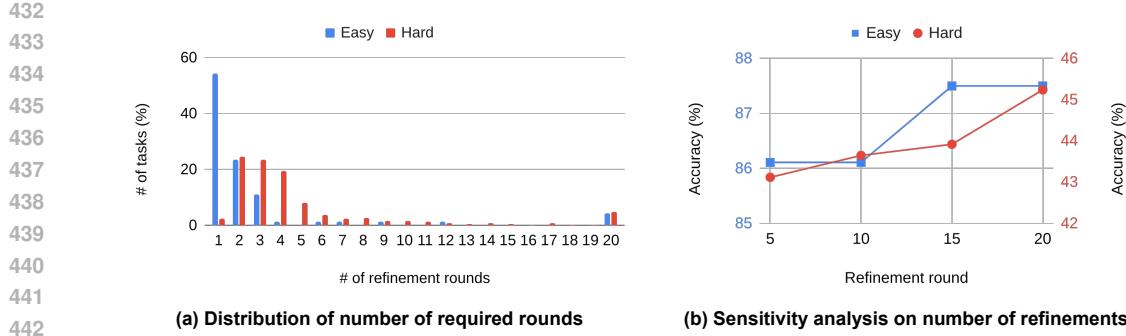


Figure 4: **Sensitivity analysis on the number of refinement steps using DABStep.** (a) Difficult tasks require more iterations to generate sufficient plans. Specifically, hard-level tasks require an average 5.6 iterations, while easy-level tasks require 3.0 iterations. Also, more than 50% of easy-level tasks are done only with a single round. (b) Performing more iterations allows the agent to generate sufficient plans, resulting in better performance for both easy and hard level tasks.

Compatibility with other LLMs. To evaluate the generalizability of DS-STAR across LLM types, we conduct additional experiments using GPT-5. As shown in Table 4, DS-STAR with GPT-5 also achieves promising results on the DABStep benchmark. Notably, DS-STAR with GPT-5 demonstrates stronger performance on easy-level tasks, whereas the Gemini-2.5-Pro excels on hard-level tasks.

4.3 ANALYSIS

In this section, we analyze the impact of the number of refinement rounds. Specifically, we measure the number of iterations required to generate a sufficient plan and conduct a sensitivity analysis on the maximum number of iterations. All experiments were done on the DABStep using Gemini-2.5-Pro. In addition, we provide cost analysis of DS-STAR in the Appendix C.

Required number of refinement steps. As shown in Figure 4(a), generating a successful plan for hard tasks requires a greater number of refinement rounds. Specifically, on the DABStep, hard tasks required an average of 5.6 rounds, in contrast to the 3.0 rounds needed for easy tasks. This disparity is further underscored by the fact that while over 50% of easy-level tasks were solved by the initial plan p_0 alone, nearly all, *i.e.*, 98% of the hard tasks necessitated at least one refinement iteration.

Sensitivity analysis on the maximum number of refinement steps. In our experiments, DS-STAR uses a default maximum of 20 refinement iterations. To analyze the sensitivity of this hyper-parameter, we conduct an experiment by lowering the limits to 5, 10, and 15. If the process reaches this maximum without finding a sufficient plan, DS-STAR generates a final solution using the intermediate plan it has developed. Figure 4(b) shows a positive correlation between the maximum number of refinements and task accuracy for both easy and hard-level tasks. A higher iteration limit increases the probability that DS-STAR will generate a sufficient plan. This seems to be more critical for hard-level tasks, where accuracy consistently improves as the maximum number of rounds increases, verifying the importance of sufficient number of refinement steps for complex problems.

5 CONCLUSION

We introduce DS-STAR, a novel agent designed to autonomously solve data science problems. Our approach features two key components: (1) the automatic analysis of files to handle heterogeneous data formats, and (2) the generation of a sequential plan that is iteratively refined with a novel LLM-based verification mechanism. We show the effectiveness of DS-STAR on the DABStep, Kramabench, and DA-Code, where it establishes a new state-of-the-art by outperforming prior methods.

Limitation and future works. Our current work focuses on a fully automated framework for DS-STAR. A compelling avenue for future research is to extend this framework to a human-in-the-loop setting. Investigating how to synergistically combine the automated capabilities of DS-STAR with the intuition and domain knowledge of a human expert presents a promising direction for significantly boosting performance and enhancing the system’s practical utility.

486 REFERENCES
487

488 Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. Lgesql: line graph enhanced
489 text-to-sql model with mixed local and non-local relations. *arXiv preprint arXiv:2106.01093*,
490 2021.

491 DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. Ryansql: Recursively
492 applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *Computa-*
493 *tional Linguistics*, 2021.

494 Tijl De Bie, Luc De Raedt, José Hernández-Orallo, Holger H Hoos, Padhraic Smyth, and Christo-
495 pher KI Williams. Automating data science. *Communications of the ACM*, 2022.

496 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen,
497 Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*,
498 2024.

499 Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. DS-agent: Automated
500 data science by empowering large language models with case-based reasoning. *International
501 Conference on Machine Learning*, 2024.

502 Md Mahadi Hassan, Alex Knipper, and Shubhra Kanti Karmaker Santu. Chatgpt as your personal
503 data scientist. *arXiv preprint arXiv:2305.13657*, 2023.

504 Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Biniao Wu, Ceyao Zhang, Chenxing Wei,
505 Danyang Li, Jiaqi Chen, Jiayi Zhang, et al. Data interpreter: An llm agent for data science. *arXiv
506 preprint arXiv:2402.18679*, 2024.

507 Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su,
508 Jingjing Xu, Ming Zhu, et al. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv
509 preprint arXiv:2401.05507*, 2024.

510 Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang,
511 Xiao Liu, Jun Zhao, et al. Da-code: Agent data science code generation benchmark for large
512 language models. *arXiv preprint arXiv:2410.07331*, 2024.

513 Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixin Xu, Ian Kaplan, Deniss Jacenko, and
514 Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*,
515 2025.

516 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik
517 Narasimhan. Swe-bench: Can language models resolve real-world github issues? *International
518 Conference on Learning Representations*, 2024.

519 Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming
520 Zhang, Xinya Du, and Dong Yu. Dsbench: How far are data science agents to becoming data
521 science experts? *International Conference on Learning Representations*, 2025.

522 Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Sivaprasad Sudhir, Om Chabra, Anna Zeng, Anton A
523 Zubreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, et al. Kramabench: A benchmark for ai
524 systems on data-to-insight pipelines over data lakes. *arXiv preprint arXiv:2506.06541*, 2025.

525 Jinyuk Lee, Feiyang Chen, Sahil Dua, Daniel Cer, Madhuri Shanbhogue, Iftekhar Naim, Gus-
526 tavo Hernández Ábrego, Zhe Li, Kaifeng Chen, Henrique Schechter Vera, et al. Gemini embedding:
527 Generalizable embeddings from gemini. *arXiv preprint arXiv:2503.07891*, 2025.

528 Fei Li and Hosagrahar V Jagadish. Constructing an interactive natural language interface for relational
529 databases. *Proceedings of the VLDB Endowment*, 2014.

530 Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhu Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying
531 Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale
532 database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 2023.

540 Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying
 541 Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale
 542 database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 2024.

543

544 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom
 545 Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation
 546 with alphacode. *Science*, 2022.

547 Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The death of schema
 548 linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*,
 549 2024.

550

551 Friso Kingma Martin Iglesias, Alex Egg. Data agent benchmark for multi-step reasoning (DABStep),
 552 2025.

553

554 Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö Arik, and Tomas Pfister. Mle-
 555 star: Machine learning engineering agent via search and targeted refinement. *arXiv preprint
 556 arXiv:2506.15692*, 2025.

557

558 Zhijie Nie, Zhangchi Feng, Mingxin Li, Cunwang Zhang, Yanzhao Zhang, Dingkun Long, and
 559 Richong Zhang. When text embedding meets large language model: a comprehensive survey.
arXiv preprint arXiv:2412.09165, 2024.

560

561 Shuyin Ouyang, Dong Huang, Jingwen Guo, Zeyu Sun, Qihao Zhu, and Jie M Zhang. Ds-bench: A
 562 realistic benchmark for data science code generation. *arXiv preprint arXiv:2505.15621*, 2025.

563

564 Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql
 565 with self-correction. *Advances in Neural Information Processing Systems*, 2023.

566

567 Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok
 568 Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. Chase-sql: Multi-path reasoning
 569 and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*,
 2024.

570

571 Iqbal H Sarker. Data science and analytics: an overview from data-driven smart computing, decision-
 572 making and applications perspective. *SN Computer Science*, 2021.

573

574 Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueteng Zhuang. Hugginggpt:
 575 Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information
 576 Processing Systems*, 2023.

577

578 Zhaohao Sun, Lizhe Sun, and Kenneth Strang. Big data analytics services for enhancing business
 579 intelligence. *Journal of Computer Information Systems*, 2018.

580

581 Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi.
 582 Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

583

584 Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and
 585 Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv
 586 preprint arXiv: Arxiv-2305.16291*, 2023.

587

588 Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan,
 589 Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software
 590 developers as generalist agents. *International Conference on Learning Representations*, 2024.

591

592 Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang,
 593 Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent
 2025.

594

595 Xu Yang, Xiao Yang, Shikai Fang, Bowen Xian, Yuante Li, Jian Wang, Minrui Xu, Haoran Pan,
 596 Xinpeng Hong, Weiqing Liu, et al. R&d-agent: Automating data-driven ai solution building through
 597 llm-powered automated research, development, and evolution. *arXiv preprint arXiv:2505.14738*,
 598 2025.

594 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.
595 React: Synergizing reasoning and acting in language models. *International Conference on Learning*
596 *Representations*, 2023.

597 Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li,
598 Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex
599 and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.

600 Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. Data-copilot: Bridging billions of
601 data and humans with autonomous workflow. *arXiv preprint arXiv:2306.07209*, 2023.

602 Yuge Zhang, Qiyang Jiang, Xingyu Han, Nan Chen, Yuqing Yang, and Kan Ren. Benchmarking data
603 science agents. *arXiv preprint arXiv:2402.17168*, 2024.

604 Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang,
605 Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and
606 chatbot arena. *Advances in neural information processing systems*, 2023.

607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647

648 A ALGORITHMS
649650

651 **Algorithm 1** DS-STAR

```

652 1: Input: query  $q$ , data files  $\mathcal{D}$ , maximum number of refinement round  $M$ 
653 2: # Analyzing data files
654 3: for  $i = 1$  to  $N$  do
655 4:    $s_{\text{desc}}^i = \mathcal{A}_{\text{analyzer}}(\mathcal{D}_i)$ 
656 5:    $d_i = \text{exec}(s_{\text{desc}}^i)$ 
657 6: end for
658 7: # Iterative plan generation and verification
659 8:  $p_0 = \mathcal{A}_{\text{planner}}(q, \{d_i\}_{i=1}^N)$ 
660 9:  $s_0 = \mathcal{A}_{\text{coder}}(p_0, \{d_i\}_{i=1}^N)$ 
661 10:  $r_0 = \text{exec}(s_0)$ 
662 11:  $p = \{p_0\}$ 
663 12: for  $k = 0$  to  $M - 1$  do
664 13:    $v = \mathcal{A}_{\text{verifier}}(p, q, s_k, r_k)$ 
665 14:   if  $v = \text{sufficient}$  then
666 15:     break
667 16:   else if  $v = \text{insufficient}$  then
668 17:      $w = \mathcal{A}_{\text{router}}(p, q, r_k, \{d_i\}_{i=1}^N)$ 
669 18:     if  $w \in \{0, \dots, \text{len}(p) - 1\}$  then
670 19:        $l = w - 1$ 
671 20:     else if  $w = \text{Add Step}$  then
672 21:        $l = k$ 
673 22:     end if
674 23:      $p \leftarrow \{p_0, \dots, p_l\}$ 
675 24:      $p_{l+1} = \mathcal{A}_{\text{planner}}(p, q, r_k, \{d_i\}_{i=1}^N)$ 
676 25:      $p \leftarrow \{p_0, \dots, p_l, p_{l+1}\}$ 
677 26:      $s_{k+1} = \mathcal{A}_{\text{coder}}(p, q, s_k, \{d_i\}_{i=1}^N)$ 
678 27:      $r_{k+1} = \text{exec}(s_{k+1})$ 
679 28:   end if
680 29: end for
681 30: Output: Final solution  $s$ 
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701

```

702 B BENCHMARK

704
705 **Table 5: Statistics of benchmark.** For the DA-Code benchmark (Huang et al., 2024), we report the
706 average of number of data files required for each tasks in each domain.

Benchmark	Domain	# Tasks	# Hard Tasks	# Data files
DABStep (Martin Iglesias, 2025)		450	378	7
KramaBench (Lai et al., 2025)	Archeology	12	6	5
	Astronomy	12	6	1556
	Biomedical	9	6	7
	Environment	20	14	37
	Legal	30	16	136
	Wildfire	21	15	23
DA-Code (Huang et al., 2024)	Data Insight	79	5	3.7
	Data Manipulation	73	23	5.8
	Data Wrangling	100	16	6.2
	Machine Learning	100	34	3.9
	Statistical Analysis	70	10	4.5
	Visualization	78	14	4.5

721 We evaluate DS-STAR against several alternatives using two challenging benchmarks: DABStep (Martin Iglesias, 2025) and KramaBench (Lai et al., 2025).

- 724 • **DABStep** is composed of 450 tasks (72 easy and 378 hard) that require analyzing a shared set of
725 seven data files. The ground-truth labels for these tasks are hidden, and evaluation is performed by
726 submitting results to an official server, ensuring a blind assessment.
- 727 • **KramaBench** tests an agent’s ability to perform autonomous data discovery. It contains tasks
728 across six distinct domains, where each domain includes up to 1,556 data files. This setup requires
729 the agent to automatically identify and select the relevant data files for given task.
- 730 • **DA-Code** is structured into three main categories: data wrangling, machine learning, and ex-
731 ploratory data analysis (EDA), with the EDA category further divided into data manipulation, data
732 insights, visualization, and statistical analysis. See Appendix F for the example tasks.

733 Table 5 provides detailed statistics for both benchmarks.

736 C COST ANALYSIS

738 **Table 6: Cost analysis on DABStep.** We report the average cost when using Gemini-2.5-Pro.

Method	# LLM calls	# Input tokens	# Output tokens	Cost (\$)
ReAct (Yao et al., 2023)	7.1	44691	2928	0.09
DA-Agent (Huang et al., 2024)	8.8	39999	4123	0.09
DS-STAR (Ours)	12.7	154669	3373	0.23

745 In this section, we quantitatively analyze the LLM usage cost. We evaluate DS-STAR on 10
746 development tasks from the DABStep benchmark (Martin Iglesias, 2025), reporting the average
747 number of LLM calls, input tokens, and output tokens.

748 We acknowledge that DS-STAR incurs higher cost due to its increased token usage. As shown in
749 Table 6, DS-STAR requires 3.5 times more input tokens than the ReAct baseline. This increase is
750 primarily because our method utilizes comprehensive analytic descriptions of each data file, which
751 inherently necessitates more input tokens. Nevertheless, this translates to a modest practical cost;
752 using Gemini-2.5-Pro (priced at \$1.25/\$10.00 per 1M input/output tokens³), the expense for DS-
753 STAR averages only \$0.23 per data science task. We argue this higher cost is a justifiable trade-off,
754 as DS-STAR consistently and significantly achieves better results than ReAct.

755 ³<https://ai.google.dev/gemini-api/docs/pricing>

756
757

D QUALITATIVE RESULTS ON THE ANALYZER AGENT

758
759

D.1 CODE GENERATED BY THE ANALYZER AGENT

760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809

```

import json

def describe_json_data(file_path):
    """
    Loads a JSON file, prints its full content, and then describes its structure
    and essential information, including column names (keys) and example data.
    """
    # Load the JSON data from the file
    # Using utf-8 encoding is a good practice for JSON.
    with open(file_path, 'r', encoding='utf-8') as f:
        data = json.load(f)

    print(f"--- Describing content of: {file_path} ---")
    print("\n")

    # 1. Print the full loaded content as per requirement
    # "The Python code should print out the content of data/merchant_data.json."
    print("--- Full Loaded Content ---")
    # Using json.dumps for a pretty-printed representation of the loaded Python object.
    # ensure_ascii=False allows for proper display of unicode characters.
    print(json.dumps(data, indent=4, ensure_ascii=False))
    print("—" * 50)
    print("\n")

    # 2. Print Essential Information
    print("--- Essential Information ---")
    data_type = type(data)
    print(f"Overall Data Type: {data_type.__name__}")

    if isinstance(data, dict):
        print("The data is a single JSON object (dictionary).")
        keys = list(data.keys())
        print(f"Number of top-level keys: {len(keys)}")

        if not keys:
            print("The dictionary is empty.")
        else:
            # Requirement: "print out all the column names."
            print("Top-level keys (can be considered as 'column names' for the main object):")
            for key in keys:
                print(f" - {key}")

            # Requirement: "If there are too many structured data, print out just few examples."
            print("\n--- Example Key-Value Pairs (first 5 if more exist) ---")
            count = 0
            for key, value in data.items():
                if count < 5:
                    value_str = str(value)
                    # Truncate long values for a concise preview
                    preview = (value_str[:70] + '...') if len(value_str) > 70 else value_str
                    print(f" {key}: {preview}")
                    count += 1
                else:
                    print(" ... (more key-value pairs exist)")
                    break
            if count == 0 and not keys: # Should be caught by 'if not keys:' earlier
                print(" (Dictionary is empty - no key-value pairs to show)")

    elif isinstance(data, list):
        print("The data is a JSON array (list).")
        num_items = len(data)
        print(f"Number of items in the list: {num_items}")

        if num_items == 0:
            print("The list is empty.")
        else:
            first_item = data[0]
            print(f"Type of the first item in the list: {type(first_item).__name__}")

            # Check if it's a list of dictionaries (common for structured data records)
            if isinstance(first_item, dict):
                print("The list appears to contain JSON objects (dictionaries), suggesting structured
                     data.")

                # Requirement: "print out all the column names."
                # For a list of objects, column names are typically the keys of these objects.
                # We'll use the keys from the first object as representative column names.
                first_item_keys = list(first_item.keys())
                if not first_item_keys:
                    print("The first object in the list is an empty dictionary (no column names to infer).")
                else:
                    print("Keys of the first object (assumed to be common 'column names' for the items):")
                    for key in first_item_keys:
                        print(f" - {key}")

            # Requirement: "If there are too many structured data, print out just few examples."

```

```

810
811     num_examples_to_show = min(3, num_items) # Show up to 3 examples
812     print(f"\n--- Example Items (first {num_examples_to_show} of {num_items}) ---")
813     for i in range(num_examples_to_show):
814         print(f"Item {i+1}:")
815         # Pretty print each example item. Truncate if an individual item is very large.
816         example_item_str = json.dumps(data[i], indent=2, ensure_ascii=False)
817         lines = example_item_str.split('\n')
818         if len(lines) > 15: # Arbitrary limit: max 15 lines per example item summary
819             print('\n'.join(lines[:15]))
820             print("... (item content truncated for brevity in this summary)")
821         else:
822             print(example_item_str)
823
824     if num_items > num_examples_to_show:
825         print(f"\n... and {num_items - num_examples_to_show} more items in the list.")
826
827     # Else, it's a list of other types (e.g., strings, numbers - unstructured or semi-structured)
828     else:
829         print("The list contains non-object items (e.g., strings, numbers, booleans, or mixed
830             types).")
831         print("This is often considered unstructured or semi-structured data.")
832         num_examples_to_show = min(5, num_items) # Show up to 5 examples
833         print(f"\n--- Example Items (first {num_examples_to_show} of {num_items}) ---")
834         for i in range(num_examples_to_show):
835             item_str = str(data[i])
836             # Truncate long string representations for a concise preview
837             preview = (item_str[:70] + '...') if len(item_str) > 70 else item_str
838             print(f"Item {i+1}: {preview}")
839
840     if num_items > num_examples_to_show:
841         print(f"\n... and {num_items - num_examples_to_show} more items in the list.")
842
843     elif isinstance(data, (str, int, float, bool)) or data is None:
844         # For simple scalar types, the "Full Loaded Content" print is the primary information.
845         # This section just confirms the nature of the data.
846         print(f"The data is a simple JSON scalar value (not an object or array).")
847         if data is None:
848             print("Specifically, the JSON content is 'null'.")
849             # The actual value was already printed in the "Full Loaded Content" section.
850
851     else:
852         # This case should ideally not be reached if json.load() was successful
853         # and returned standard Python type corresponding to a valid JSON root element.
854         print("The data type is unusual for a root JSON element (e.g., a custom object if json.load was
855             hooked).")
856         # The actual value was already printed in the "Full Loaded Content" section.
857
858     print("\n" + "-" * 50)
859     print("--- End of Description ---")
860
861 if __name__ == "__main__":
862     # The path to the JSON file.
863     # This script assumes 'data/merchant_data.json' exists relative to where the script is run.
864     # For example, if the script is in /home/user/scripts/, it expects
865     #   /home/user/scripts/data/merchant_data.json
866     json_file_path = 'data/merchant_data.json'
867
868     # Call the function to load and describe the JSON data.
869     # As per requirements, no try-except blocks are used for error handling.
870     # If the file doesn't exist, a FileNotFoundError will be raised.
871     # If the file's content is not valid JSON, a json.JSONDecodeError will be raised.
872     # These errors will halt the program, and debugging can proceed from there.
873     describe_json_data(json_file_path)

```

Listing 1: Python code for generating the description of ‘merchant_data.json’ in DABStep.

```

850
851
852
853
854
855
856
857
858
859
860
861
862
863

```

```

864
865 import pandas as pd
866 import numpy as np
867 import io
868 import os
869
870 # --- Configuration ---
871 # Define the path to the Excel file to be analyzed.
872 # This script assumes the file is located at 'data/1-s2.0-S0092867420301070-mmc4.xlsx'.
873 file_path = 'data/1-s2.0-S0092867420301070-mmc4.xlsx'
874
875 # --- Analyst's Report Generation ---
876
877 # Print a formatted header for the analysis report.
878 print("====")
879 print(f" Expert Data Analysis Report for: {file_path}")
880 print("====")
881
882 # Check for file existence to provide a clear startup message.
883 # The script will still raise an error if not found, as per requirements.
884 if not os.path.exists(file_path):
885     print(f"\n[STATUS] File not found at the specified path: '{file_path}'")
886     print("Please ensure the file exists and the path is correct before execution.")
887     # The script will exit here due to a FileNotFoundError from pandas,
888     # allowing for debugging as requested.
889
890 # --- 1. Loading Data and Sheet Information ---
891
892 print("\n--- 1. Loading & Sheet Information ---")
893 # Use pandas.ExcelFile to inspect the workbook without loading all data initially.
894 # This is efficient for understanding the structure of a multi-sheet Excel file.
895 excel_file = pd.ExcelFile(file_path)
896 sheet_names = excel_file.sheet_names
897 print(f"Found {len(sheet_names)} sheet(s) in the workbook: {sheet_names}")
898
899 # For this analysis, we will focus on the first sheet.
900 first_sheet_name = sheet_names[0]
901 print(f"Loading data from the first sheet: '{first_sheet_name}'")
902 df = pd.read_excel(excel_file, sheet_name=first_sheet_name)
903 print("Data loaded successfully.")
904
905 # --- 2. High-Level Overview ---
906
907 print("\n--- 2. Data Dimensions and Structure ---")
908 rows, cols = df.shape
909 print(f"The dataset contains {rows} rows and {cols} columns.")
910
911 # An analyst's initial assessment of data structure.
912 if cols == 1:
913     print("Observation: With only one column, the data might be unstructured (e.g., a list of texts or
914     → IDs).")
915 elif 1 < cols < 5:
916     print("Observation: With a few columns, the data appears to be structured but simple.")
917 else:
918     print("Observation: With multiple columns, the data appears to be well-structured.")
919
920 # --- 3. Column Analysis ---
921
922 print("\n--- 3. Column Names ---")
923 print("The following columns are present in the dataset:")
924 # Using a list comprehension for clean printing of column names.
925 print(" | ".join([f"'{col}'" for col in df.columns]))
926
927 # --- 4. Data Types and Non-Null Counts ---
928
929 print("\n--- 4. Data Types & Memory Usage ---")
930 # Capture the output of df.info() to present it cleanly within the report.
931 buffer = io.StringIO()
932 df.info(buf=buffer)
933 info_str = buffer.getvalue()
934 print(info_str)
935
936 # --- 5. Missing Value Analysis ---
937
938 print("\n--- 5. Missing Values Summary ---")
939 missing_values = df.isnull().sum()
940 total_missing = missing_values.sum()
941 if total_missing == 0:
942     print("Excellent! No missing values were found in the dataset.")
943 else:
944     print(f"Found a total of {total_missing} missing values.")
945     # Display columns that have at least one missing value.
946     missing_per_column = missing_values[missing_values > 0].sort_values(ascending=False)
947     print("Columns with missing values:")
948     print(missing_per_column.to_string())
949
950 # --- 6. Descriptive Statistics for Numerical Data ---
951
952 print("\n--- 6. Descriptive Statistics (for Numerical Columns) ---")
953 # The describe() method provides a powerful summary of central tendency,
954 # dispersion, and shape of the distribution for numerical columns.

```

```

918 # It automatically excludes non-numerical columns.
919 descriptive_stats = df.describe()
920 if descriptive_stats.empty:
921     print("No numerical columns found to generate statistics for.")
922 else:
923     print(descriptive_stats)
924
925 # --- 7. Data Preview ---
926
927 print("\n--- 7. Data Sample (First 5 Rows) ---")
928 # Displaying the head of the DataFrame is crucial for getting a feel
929 # for the actual content and format of the data.
930 print(df.head())
931
932 # --- 8. Analysis of Categorical / Object Data ---
933 print("\n--- 8. Unique Values in Categorical Columns ---")
934 # Select columns with 'object' dtype, which are often categorical.
935 object_cols = df.select_dtypes(include=['object']).columns
936
937 if len(object_cols) == 0:
938     print("No categorical (object type) columns found for value count analysis.")
939 else:
940     print(f"Analyzing {len(object_cols)} categorical column(s): {list(object_cols)}\n")
941     for col in object_cols:
942         unique_count = df[col].nunique()
943         print(f"--> Column '{col}' has {unique_count} unique values.")
944         # To avoid flooding the output, only show value counts for columns with few unique values.
945         if unique_count > 1 and unique_count <= 15:
946             print("  Value Counts:")
947             print(df[col].value_counts().to_string())
948             print("-" * 20)
949
950 # --- End of Report ---
951 print("\n=====")
952 print("      End of Analysis Report")
953 print("=====")
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

```

Listing 2: Python code for generating the description of ‘1-s2.0-S0092867420301070-mmc4.xlsx’ in KramaBench.

972 D.2 EXAMPLES OF GENERATED DATA DESCRIPTIONS

```

973
974 f'''
975 File: data/merchant_data.json (6857 bytes)
976 Top-level type: array
977 root: array with 30 items
978 Element types: object=30
979 Columns (5): ['account_type', 'acquirer', 'capture_delay', 'merchant',
980   ↳ 'merchant_category_code']
981 Sample rows (first 5):
982 [
983   {
984     "merchant": "Crossfit_Hanna",
985     "capture_delay": "manual",
986     "acquirer": [
987       "gringotts",
988       "the_savings_and_loan_bank",
989       "bank_of_springfield",
990       "dagoberts_vault"
991     ],
992     "merchant_category_code": 7997,
993     "account_type": "F"
994   },
995   {
996     "merchant": "Martinis_Fine_Steakhouse",
997     "capture_delay": "immediate",
998     "acquirer": [
999       "dagoberts_geldpakhuis",
1000       "bank_of_springfield"
1001     ],
1002     "merchant_category_code": 5812,
1003     "account_type": "H"
1004   },
1005   {
1006     "merchant": "Belles_cookbook_store",
1007     "capture_delay": "1",
1008     "acquirer": [
1009       "lehman_brothers"
1010     ],
1011     "merchant_category_code": 5942,
1012     "account_type": "R"
1013   },
1014   {
1015     "merchant": "Golfclub_Baron_Friso",
1016     "capture_delay": "2",
1017     "acquirer": [
1018       "medici"
1019     ],
1020     "merchant_category_code": 7993,
1021     "account_type": "F"
1022   },
1023   {
1024     "merchant": "Rafa_AI",
1025     "capture_delay": "7",
1026     "acquirer": [
1027       "tellsons_bank"
1028     ],
1029     "merchant_category_code": 7372,
1030     "account_type": "D"
1031   }
1032 ]
1033 '''

```

1025 Listing 3: Description of ‘merchant_data.json’ in DABstep.

```

1026
1027 f'''
1028 =====
1029     Expert Data Analysis Report for:
1030     ↳ data/1-s2.0-S0092867420301070-mmc4.xlsx
1031 =====
1032
1033     --- 1. Loading & Sheet Information ---
1034     Found 4 sheet(s) in the workbook: ['README', 'A-Variants', 'B-Novel
1035     ↳ Splice Junctions', 'C-Alternate Splice Junctions']
1036     Loading data from the first sheet: 'README'
1037     Data loaded successfully.
1038
1039     --- 2. Data Dimensions and Structure ---
1040     The dataset contains 25 rows and 2 columns.
1041     Observation: With a few columns, the data appears to be structured but
1042     ↳ simple.
1043
1044     --- 3. Column Names ---
1045     The following columns are present in the dataset:
1046     'IN THIS FILE: a list of variants and alternate splice junctions for
1047     ↳ which we found peptide evidence. For more information, please see
1048     ↳ STAR Methods' | 'Unnamed: 1'
1049
1050     --- 4. Data Types & Memory Usage ---
1051     <class 'pandas.core.frame.DataFrame'>
1052     RangeIndex: 25 entries, 0 to 24
1053     Data columns (total 2 columns):
1054         #   Column
1055         ↳ Non-Null Count  Dtype
1056         ---   -----
1057         ↳ 0   IN THIS FILE: a list of variants and alternate splice junctions for
1058         ↳ which we found peptide evidence. For more information, please see
1059         ↳ STAR Methods  21 non-null   object
1060         ↳ 1   Unnamed: 1
1061         ↳ 21 non-null   object
1062     dtypes: object(2)
1063     memory usage: 528.0+ bytes
1064
1065
1066     --- 5. Missing Values Summary ---
1067     Found a total of 8 missing values.
1068     Columns with missing values:
1069     IN THIS FILE: a list of variants and alternate splice junctions for
1070     ↳ which we found peptide evidence. For more information, please see
1071     ↳ STAR Methods  4
1072     Unnamed: 1
1073     ↳ 4
1074
1075     --- 6. Descriptive Statistics (for Numerical Columns) ---
1076         IN THIS FILE: a list of variants and alternate splice junctions
1077         ↳ for which we found peptide evidence. For more information, please
1078         ↳ see STAR Methods  Unnamed: 1
1079         count
1080         ↳ 21
1081         unique
1082         ↳ 21
1083         top
1084         ↳ Description
1085         freq
1086         ↳ 1
1087
1088     --- 7. Data Sample (First 5 Rows) ---
1089

```

```

1080      IN THIS FILE: a list of variants and alternate splice junctions for
1081      ↵ which we found peptide evidence. For more information, please see
1082      ↵ STAR Methods                                         Unnamed: 1
1083      0                                              NaN
1084      ↵ NaN
1085      1                                              NaN
1086      ↵ NaN
1087      2                                              Sheet
1088      ↵ Description
1089      3                                              A-Variants
1090      ↵ Both single amino acid variants (SAAVs) and in...
1091      ↵ 4                                              B-Novel Splice Junctions
1092      ↵ Splice junctions where neither boundary was at...
1093
1094      --- 8. Unique Values in Categorical Columns ---
1095      Analyzing 2 categorical column(s): ['IN THIS FILE: a list of variants
1096      ↵ and alternate splice junctions for which we found peptide evidence.
1097      ↵ For more information, please see STAR Methods', 'Unnamed: 1']
1098
1099      -> Column 'IN THIS FILE: a list of variants and alternate splice
1100      ↵ junctions for which we found peptide evidence. For more information,
1101      ↵ please see STAR Methods' has 21 unique values.
1102      -> Column 'Unnamed: 1' has 21 unique values.
1103
1104      =====
1105      End of Analysis Report
1106      =====
1107      ''

```

Listing 4: Description of ‘1-s2.0-S0092867420301070-mmc4.xlsx’ in KramaBench.

```

1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

1134
1135

E ROBUSTNESS OF THE ANALYZER AGENT

1136
1137
1138

In this section, we provide qualitative results for the analyzer agent of DS-STAR. Specifically, we first provide the description of 1-s2.0-S0092867420301070-mmc2.xlsx from the biomedical domain of KramaBench, generated by DS-STAR.

1139

```

f'''
Analyzing Excel File: data/mmc2.xlsx
Found 3 sheet(s): README, A-global-proteomics, B-phospho-proteomics
=====
Analyzing Sheet: 'README' (Sheet 1/3)
-----
Shape: 9 rows, 2 columns

Column Names:
['Sheet', 'Description']

Data Types and Non-Null Counts (df.info()):
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
---  --          -----          --    
 0   Sheet        9 non-null    object  
 1   Description  9 non-null    object  
dtypes: object(2)
memory usage: 272.0+ bytes

First 5 rows (Head):
Sheet
→ Description
0  A-global-proteomics  Global proteomics data, median polishing and
→ 1...
1  B-phospho-proteomics  Phosphoproteomics site level data, median
→ poli...
2  C-acetyl-proteomics  Acetylproteomics site level data, median
→ polis...
3  D-gene-expression  Linear gene expression RSEM, upper quantile
→ no...
4  E-circRNA-expression  Circular RNA expression RSEM, upper quantile
→ n...

Descriptive Statistics (df.describe(include='all')):
Sheet
→ Description
count          9
→ 9
unique         9
→ 9
top  A-global-proteomics  Global proteomics data, median polishing
→ and 1...
freq           1
→ 1

Number of Unique Values per Column (df.nunique()):
Sheet      9
Description  9
dtype: int64

----- Next Sheet -----
Analyzing Sheet: 'A-global-proteomics' (Sheet 2/3)
-----
```

```

1188 Shape: 10999 rows, 154 columns
1189
1190 Column Names:
1191 ['idx', 'S001', 'S002', 'S003', 'S004', 'S005', 'S006', 'S007', 'S008',
1192 → 'S009', 'S010', 'S011', 'S012', 'S013', 'S014', 'S015', 'S016',
1193 → 'S017', 'S018', 'S019', 'S020', 'S021', 'S022', 'S023', 'S024',
1194 → 'S025', 'S026', 'S027', 'S028', 'S029', 'S030', 'S031', 'S032',
1195 → 'S033', 'S034', 'S035', 'S036', 'S037', 'S038', 'S039', 'S040',
1196 → 'S041', 'S042', 'S043', 'S044', 'S045', 'S046', 'S047', 'S048',
1197 → 'S049', 'S050', 'S051', 'S052', 'S053', 'S054', 'S055', 'S056',
1198 → 'S057', 'S058', 'S059', 'S060', 'S061', 'S062', 'S063', 'S064',
1199 → 'S065', 'S066', 'S067', 'S068', 'S069', 'S070', 'S071', 'S072',
1200 → 'S073', 'S074', 'S075', 'S076', 'S077', 'S078', 'S079', 'S080',
1201 → 'S081', 'S082', 'S083', 'S084', 'S085', 'S086', 'S087', 'S088',
1202 → 'S089', 'S090', 'S091', 'S092', 'S093', 'S094', 'S095', 'S096',
1203 → 'S097', 'S098', 'S099', 'S100', 'S101', 'S102', 'S103', 'S104',
1204 → 'S105', 'S106', 'S107', 'S108', 'S109', 'S110', 'S111', 'S112',
1205 → 'S113', 'S114', 'S115', 'S116', 'S117', 'S118', 'S119', 'S120',
1206 → 'S121', 'S122', 'S123', 'S124', 'S125', 'S126', 'S127', 'S128',
1207 → 'S129', 'S130', 'S131', 'S132', 'S133', 'S134', 'S135', 'S136',
1208 → 'S137', 'S138', 'S139', 'S140', 'S141', 'S142', 'S143', 'S144',
1209 → 'S145', 'S146', 'S147', 'S148', 'S149', 'S150', 'S151', 'S152',
1210 → 'S153']
1211 Data Types and Non-Null Counts (df.info()):
1212 <class 'pandas.core.frame.DataFrame'>
1213 RangeIndex: 10999 entries, 0 to 10998
1214 Data columns (total 154 columns):
1215 #   Column  Dtype
1216 ---  -----
1217 0   idx     object
1218 1   S001    float64
1219 2   S002    float64
1220 3   S003    float64
1221 4   S004    float64
1222 5   S005    float64
1223 6   S006    float64
1224 7   S007    float64
1225 8   S008    float64
1226 9   S009    float64
1227 10  S010    float64
1228 11  S011    float64
1229 12  S012    float64
1230 13  S013    float64
1231 14  S014    float64
1232 15  S015    float64
1233 16  S016    float64
1234 17  S017    float64
1235 18  S018    float64
1236 19  S019    float64
1237 20  S020    float64
1238 21  S021    float64
1239 22  S022    float64
1240 23  S023    float64
1241 24  S024    float64
1242 25  S025    float64
1243 26  S026    float64
1244 27  S027    float64
1245 28  S028    float64
1246 29  S029    float64
1247 30  S030    float64
1248 31  S031    float64
1249 32  S032    float64
1250 33  S033    float64
1251 34  S034    float64

```

1242	35	S035	float64
1243	36	S036	float64
1244	37	S037	float64
1245	38	S038	float64
1246	39	S039	float64
1247	40	S040	float64
1248	41	S041	float64
1249	42	S042	float64
1250	43	S043	float64
1251	44	S044	float64
1252	45	S045	float64
1253	46	S046	float64
1254	47	S047	float64
1255	48	S048	float64
1256	49	S049	float64
1257	50	S050	float64
1258	51	S051	float64
1259	52	S052	float64
1260	53	S053	float64
1261	54	S054	float64
1262	55	S055	float64
1263	56	S056	float64
1264	57	S057	float64
1265	58	S058	float64
1266	59	S059	float64
1267	60	S060	float64
1268	61	S061	float64
1269	62	S062	float64
1270	63	S063	float64
1271	64	S064	float64
1272	65	S065	float64
1273	66	S066	float64
1274	67	S067	float64
1275	68	S068	float64
1276	69	S069	float64
1277	70	S070	float64
1278	71	S071	float64
1279	72	S072	float64
1280	73	S073	float64
1281	74	S074	float64
1282	75	S075	float64
1283	76	S076	float64
1284	77	S077	float64
1285	78	S078	float64
1286	79	S079	float64
1287	80	S080	float64
1288	81	S081	float64
1289	82	S082	float64
1290	83	S083	float64
1291	84	S084	float64
1292	85	S085	float64
1293	86	S086	float64
1294	87	S087	float64
1295	88	S088	float64
1296	89	S089	float64
1297	90	S090	float64
1298	91	S091	float64
1299	92	S092	float64
1300	93	S093	float64
1301	94	S094	float64
1302	95	S095	float64
1303	96	S096	float64
1304	97	S097	float64
1305	98	S098	float64
1306	99	S099	float64

```

1296    100  S100    float64
1297    101  S101    float64
1298    102  S102    float64
1299    103  S103    float64
1300    104  S104    float64
1301    105  S105    float64
1302    106  S106    float64
1303    107  S107    float64
1304    108  S108    float64
1305    109  S109    float64
1306    110  S110    float64
1307    111  S111    float64
1308    112  S112    float64
1309    113  S113    float64
1310    114  S114    float64
1311    115  S115    float64
1312    116  S116    float64
1313    117  S117    float64
1314    118  S118    float64
1315    119  S119    float64
1316    120  S120    float64
1317    121  S121    float64
1318    122  S122    float64
1319    123  S123    float64
1320    124  S124    float64
1321    125  S125    float64
1322    126  S126    float64
1323    127  S127    float64
1324    128  S128    float64
1325    129  S129    float64
1326    130  S130    float64
1327    131  S131    float64
1328    132  S132    float64
1329    133  S133    float64
1330    134  S134    float64
1331    135  S135    float64
1332    136  S136    float64
1333    137  S137    float64
1334    138  S138    float64
1335    139  S139    float64
1336    140  S140    float64
1337    141  S141    float64
1338    142  S142    float64
1339    143  S143    float64
1340    144  S144    float64
1341    145  S145    float64
1342    146  S146    float64
1343    147  S147    float64
1344    148  S148    float64
1345    149  S149    float64
1346    150  S150    float64
1347    151  S151    float64
1348    152  S152    float64
1349    153  S153    float64
1343  dtypes: float64(153), object(1)
1343  memory usage: 12.9+ MB
1343
1344  First 5 rows (Head):
1345      idx    S001    S002    S003    S004    ...    S149    S150    S151    S152
1346  ↪  S153
1347  0      A1BG -1.180 -0.685 -0.528  2.350    ...    0.650   0.458   1.1500  0.547
1348  ↪  0.9400
1349  1      A2M  -0.863 -1.070 -1.320  2.820    ...    0.227   0.520   1.4600  1.270
1349  ↪  0.9040

```

```
1350 2 A2ML1 -0.802 -0.684 0.435 -1.470 ... 1.930 -0.291 -0.0229 -0.197
1351 ↪ -0.0803
1352 3 A4GALT 0.222 0.984 NaN NaN ... NaN NaN NaN NaN
1353 ↪ NaN
1354 4 AAAS 0.256 0.135 -0.240 0.154 ... 0.239 0.477 0.2520 0.405
1355 ↪ 0.2990
1356 [5 rows x 154 columns]
1357
1358 Last 3 rows (Tail):
1359     idx S001 S002 S003 ... S150 S151 S152 S153
1360 10996 ZYX -1.0200 -1.1300 -0.540 ... -0.3990 0.83500 0.416 -0.4220
1361 10997 ZZEF1 -0.1230 -0.0757 0.320 ... -0.0959 0.16900 0.273 -0.0931
1362 10998 ZZZ3 -0.0859 -0.4730 -0.419 ... -0.0635 -0.00809 -0.658 0.0557
1363 [3 rows x 154 columns]
1364
1365 Descriptive Statistics (df.describe(include='all')):
1366     idx S001 S002 ... S151 S152
1367 ↪ S153
1368 count 10999 9689.000000 9943.000000 ... 9646.000000 9646.000000
1369 ↪ 9648.000000
1370 unique 10999 NaN NaN ... NaN NaN
1371 ↪ NaN
1372 top ZZZ3 NaN NaN ... NaN NaN
1373 ↪ NaN
1374 freq 1 NaN NaN ... NaN NaN
1375 ↪ NaN
1376 mean NaN -0.021396 -0.036661 ... 0.079756 0.035783
1377 ↪ -0.006398
1378 std NaN 0.611966 0.678262 ... 0.542243 0.660306
1379 ↪ 0.528082
1380 min NaN -3.550000 -8.190000 ... -3.220000 -7.050000
1381 ↪ -4.190000
1382 25% NaN -0.340000 -0.381000 ... -0.187000 -0.249750
1383 ↪ -0.216000
1384 50% NaN -0.018600 -0.008360 ... 0.013300 0.007760
1385 ↪ 0.005690
1386 75% NaN 0.309000 0.311000 ... 0.272000 0.305000
1387 ↪ 0.225250
1388 max NaN 3.570000 5.290000 ... 8.340000 7.490000
1389 ↪ 2.970000
1390
1391 [11 rows x 154 columns]
1392
1393 Number of Unique Values per Column (df.nunique()):
1394 idx 10999
1395 S001 3088
1396 S002 3207
1397 S003 3146
1398 S004 3062
1399 ...
1400 S149 3213
1401 S150 3255
1402 S151 3201
1403 S152 3232
1404 S153 3284
1405 Length: 154, dtype: int64
1406
1407 ----- Next Sheet -----
1408
1409 Analyzing Sheet: 'B-phospho-proteomics' (Sheet 3/3)
1410
1411 -----
```

```

1404
1405 Column Names:
1406 ['idx', 'S001', 'S002', 'S003', 'S004', 'S005', 'S006', 'S007', 'S008',
1407    ↪ 'S009', 'S010', 'S011', 'S012', 'S013', 'S014', 'S015', 'S016',
1408    ↪ 'S017', 'S018', 'S019', 'S020', 'S021', 'S022', 'S023', 'S024',
1409    ↪ 'S025', 'S026', 'S027', 'S028', 'S029', 'S030', 'S031', 'S032',
1410    ↪ 'S033', 'S034', 'S035', 'S036', 'S037', 'S038', 'S039', 'S040',
1411    ↪ 'S041', 'S042', 'S043', 'S044', 'S045', 'S046', 'S047', 'S048',
1412    ↪ 'S049', 'S050', 'S051', 'S052', 'S053', 'S054', 'S055', 'S056',
1413    ↪ 'S057', 'S058', 'S059', 'S060', 'S061', 'S062', 'S063', 'S064',
1414    ↪ 'S065', 'S066', 'S067', 'S068', 'S069', 'S070', 'S071', 'S072',
1415    ↪ 'S073', 'S074', 'S075', 'S076', 'S077', 'S078', 'S079', 'S080',
1416    ↪ 'S081', 'S082', 'S083', 'S084', 'S085', 'S086', 'S087', 'S088',
1417    ↪ 'S089', 'S090', 'S091', 'S092', 'S093', 'S094', 'S095', 'S096',
1418    ↪ 'S097', 'S098', 'S099', 'S100', 'S101', 'S102', 'S103', 'S104',
1419    ↪ 'S105', 'S106', 'S107', 'S108', 'S109', 'S110', 'S111', 'S112',
1420    ↪ 'S113', 'S114', 'S115', 'S116', 'S117', 'S118', 'S119', 'S120',
1421    ↪ 'S121', 'S122', 'S123', 'S124', 'S125', 'S126', 'S127', 'S128',
1422    ↪ 'S129', 'S130', 'S131', 'S132', 'S133', 'S134', 'S135', 'S136',
1423    ↪ 'S137', 'S138', 'S139', 'S140', 'S141', 'S142', 'S143', 'S144',
1424    ↪ 'S145', 'S146', 'S147', 'S148', 'S149', 'S150', 'S151', 'S152',
1425    ↪ 'S153']
1426
1427 Data Types and Non-Null Counts (df.info()):
1428 <class 'pandas.core.frame.DataFrame'>
1429 RangeIndex: 73212 entries, 0 to 73211
1430 Data columns (total 154 columns):
1431 #   Column  Dtype
1432 ---  -----
1433 0   idx     object
1434 1   S001    float64
1435 2   S002    float64
1436 3   S003    float64
1437 4   S004    float64
1438 5   S005    float64
1439 6   S006    float64
1440 7   S007    float64
1441 8   S008    float64
1442 9   S009    float64
1443 10  S010    float64
1444 11  S011    float64
1445 12  S012    float64
1446 13  S013    float64
1447 14  S014    float64
1448 15  S015    float64
1449 16  S016    float64
1450 17  S017    float64
1451 18  S018    float64
1452 19  S019    float64
1453 20  S020    float64
1454 21  S021    float64
1455 22  S022    float64
1456 23  S023    float64
1457 24  S024    float64
1458 25  S025    float64
1459 26  S026    float64
1460 27  S027    float64
1461 28  S028    float64
1462 29  S029    float64
1463 30  S030    float64
1464 31  S031    float64
1465 32  S032    float64
1466 33  S033    float64
1467 34  S034    float64
1468 35  S035    float64

```

1458	36	S036	float64
1459	37	S037	float64
1460	38	S038	float64
1461	39	S039	float64
1462	40	S040	float64
1463	41	S041	float64
1464	42	S042	float64
1465	43	S043	float64
1466	44	S044	float64
1467	45	S045	float64
1468	46	S046	float64
1469	47	S047	float64
1470	48	S048	float64
1471	49	S049	float64
1472	50	S050	float64
1473	51	S051	float64
1474	52	S052	float64
1475	53	S053	float64
1476	54	S054	float64
1477	55	S055	float64
1478	56	S056	float64
1479	57	S057	float64
1480	58	S058	float64
1481	59	S059	float64
1482	60	S060	float64
1483	61	S061	float64
1484	62	S062	float64
1485	63	S063	float64
1486	64	S064	float64
1487	65	S065	float64
1488	66	S066	float64
1489	67	S067	float64
1490	68	S068	float64
1491	69	S069	float64
1492	70	S070	float64
1493	71	S071	float64
1494	72	S072	float64
1495	73	S073	float64
1496	74	S074	float64
1497	75	S075	float64
1498	76	S076	float64
1499	77	S077	float64
1500	78	S078	float64
1501	79	S079	float64
1502	80	S080	float64
1503	81	S081	float64
1504	82	S082	float64
1505	83	S083	float64
1506	84	S084	float64
1507	85	S085	float64
1508	86	S086	float64
1509	87	S087	float64
1510	88	S088	float64
1511	89	S089	float64
	90	S090	float64
	91	S091	float64
	92	S092	float64
	93	S093	float64
	94	S094	float64
	95	S095	float64
	96	S096	float64
	97	S097	float64
	98	S098	float64
	99	S099	float64
	100	S100	float64

```

1512    101  S101    float64
1513    102  S102    float64
1514    103  S103    float64
1515    104  S104    float64
1516    105  S105    float64
1517    106  S106    float64
1518    107  S107    float64
1519    108  S108    float64
1519    109  S109    float64
1520    110  S110    float64
1521    111  S111    float64
1522    112  S112    float64
1523    113  S113    float64
1523    114  S114    float64
1524    115  S115    float64
1525    116  S116    float64
1526    117  S117    float64
1527    118  S118    float64
1527    119  S119    float64
1528    120  S120    float64
1529    121  S121    float64
1530    122  S122    float64
1531    123  S123    float64
1532    124  S124    float64
1532    125  S125    float64
1533    126  S126    float64
1534    127  S127    float64
1535    128  S128    float64
1536    129  S129    float64
1537    130  S130    float64
1537    131  S131    float64
1538    132  S132    float64
1539    133  S133    float64
1540    134  S134    float64
1541    135  S135    float64
1542    136  S136    float64
1542    137  S137    float64
1543    138  S138    float64
1544    139  S139    float64
1545    140  S140    float64
1546    141  S141    float64
1547    142  S142    float64
1548    143  S143    float64
1549    144  S144    float64
1549    145  S145    float64
1550    146  S146    float64
1551    147  S147    float64
1552    148  S148    float64
1552    149  S149    float64
1553    150  S150    float64
1554    151  S151    float64
1555    152  S152    float64
1556    153  S153    float64
1557 dtypes: float64(153), object(1)
1558 memory usage: 86.0+ MB
1559
1560 First 5 rows (Head):
1561      idx    S001     S002     S003     S004     ...     S149     S150     S151
1561      ↳ S152     S153
1562      0  AAAS-S495     NaN     NaN  -0.202    0.25     ...   -0.272    0.223  -0.3940
1562      ↳ 0.149  0.0774
1563      1  AAAS-S541     NaN     NaN     NaN     ...   -0.191   -0.517  -0.0108
1564      ↳ 0.256  -0.1580
1565      2  AAAS-Y485     NaN     NaN     NaN     ...     NaN     NaN     NaN
1565      ↳ NaN      NaN

```

```

1566 3 AACS-S618 -0.881      NaN      NaN      NaN  ...      NaN      NaN      NaN
1567  ↪  NaN      NaN
1568 4 AAED1-S12 -1.810  0.084 -1.880      NaN  ...  0.631  0.522  1.0600
1569  ↪  0.951 -0.3430
1570
1571 [5 rows x 154 columns]
1572
1573 Last 3 rows (Tail):
1574      idx  S001  S002  S003  S004  ...  S149  S150  S151
1575  ↪  S152  S153
1576 73209  ZZZ3-T415  NaN  NaN  NaN  ...  NaN  NaN  NaN
1577  ↪  NaN  NaN
1578 73210  ZZZ3-T418  NaN  0.1605  NaN  NaN  ...  NaN  NaN  NaN
1579  ↪  NaN  NaN
1580 73211  ZZZ3-Y399  NaN -0.0635  NaN  NaN  ...  0.0  0.179 -0.122
1581  ↪ -0.354  0.0216
1582
1583 [3 rows x 154 columns]
1584
1585 Descriptive Statistics (df.describe(include='all')):
1586      idx  S001  ...  S152  S153
1587 count  73212  31671.000000  ...  34285.000000  34303.000000
1588 unique  73212  NaN  ...  NaN  NaN
1589 top  ZZZ3-Y399  NaN  ...  NaN  NaN
1590 freq  1  NaN  ...  NaN  NaN
1591 mean  NaN  -0.065974  ...  0.005505  0.001946
1592 std  NaN  0.641707  ...  0.674115  0.543703
1593 min  NaN  -7.020000  ...  -5.950000  -5.060000
1594 25%  NaN  -0.399000  ...  -0.328000  -0.253000
1595 50%  NaN  -0.030100  ...  -0.001340  0.000000
1596 75%  NaN  0.292000  ...  0.331500  0.261500
1597 max  NaN  4.320000  ...  6.520000  3.940000
1598
1599 [11 rows x 154 columns]
1600
1601 Number of Unique Values per Column (df.nunique()):
1602 idx  73212
1603 S001  7154
1604 S002  7644
1605 S003  5834
1606 S004  6596
1607 ...
1608 S149  7474
1609 S150  7612
1610 S151  7327
1611 S152  7393
1612 S153  7501
1613 Length: 154, dtype: int64
1614 =====
1615 Excel file 'data/mmc2.xlsx' analysis complete.
1616 ''
1617
1618
1619

```

Listing 5: Description of ‘1-s2.0-S0092867420301070-mmc2.xlsx’ in KramaBench.

1620 Moreover, to show that DS-STAR's analyzer agent is capable to generate an accurate and complete
 1621 text summary even for highly complex and deeply nested data, we first created a dummy xlsx
 1622 file that contains three tables which are placed in random position. This requires DS-STAR to
 1623 find the bounding boxes of each table exactly. As shown in the below Listing, we found that
 1624 DS-STAR successfully handles such complex data, showing the robustness of our analyzer agent.
 1625

```

1626 f'''
1627 =====
1628   Excel File Analysis: data/dummy.xlsx
1629 =====
1630
1631 Found 1 sheet(s): ['Sheet1']
1632 -----
1633   Analyzing Sheet: 'Sheet1'
1634 -----
1635   - Sheet Dimensions: 15 rows, 11 columns
1636
1637 >>> Found 3 distinct data block(s).
1638
1639   --- Block 1 (Bounding Box: B1:D6) ---
1640   * Type: Structured Table
1641   * Shape: 5 rows, 3 columns
1642   * Column Names: ['exchange', 'ticker', 'name']
1643   * Data Types:
1644     - exchange: object
1645     - ticker: object
1646     - name: object
1647   * Data Preview (first 5 rows):
1648     exchange ticker name
1649     nasdaq    PYPL      PayPal Holdings
1650     nasdaq    AMZN      Amazon.com Inc
1651     nasdaq    MSFT      Microsoft Corp.
1652     nyse     IBM International Business Machines Corporation
1653     nasdaq    GOOGL      Alphabet
1654
1655   --- Block 2 (Bounding Box: I5:K10) ---
1656   * Type: Structured Table
1657   * Shape: 5 rows, 3 columns
1658   * Column Names: ['role', 'age', 'sex']
1659   * Data Types:
1660     - role: object
1661     - age: float64
1662     - sex: object
1663   * Data Preview (first 5 rows):
1664     role age sex
1665     passenger 24.0 female
1666     passenger 31.0 male
1667     driver 72.0 female
1668     driver 66.0 male
1669     passenger 65.0 male
1670
1671   --- Block 3 (Bounding Box: E10:G15) ---
1672   * Type: Structured Table
1673   * Shape: 5 rows, 3 columns
1674   * Column Names: ['id', 'country_id', 'name']
1675   * Data Types:
1676     - id: float64
1677     - country_id: object
1678     - name: object
1679   * Data Preview (first 5 rows):
1680     id      country_id   name
1681     1.0    Belgium      Jupiler League
1682     1729.0  England     Premier League
1683     4769.0  France      Ligue 1
  
```

```
1674 7809.0 Germany 1. Bundesliga Germany
1675 10257.0           Italy Serie A   Italy
1676
1677 =====
1678   End of Analysis
1679 =====
1680   '''
```

Listing 6: Description of ‘dummy.xlsx’.

1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727

1728 **F QUALITATIVE RESULTS FOR EACH TASK TYPE**
17291730 This section provides qualitative results. It includes queries on DA-Code benchmark, solution code
1731 generated by DS-STAR, output generated by DS-STAR, and ground-truth answer. Specifically, it
1732 presents examples for each task, such as statistical analysis, data insights, data wrangling, machine
1733 learning, and visualization tasks.1734 **F.1 STATISTICAL ANALYSIS**
17351736

```
f'''Could you analyze our Business Case dataset to perform a hypothesis
→ test comparing bike rentals across different weather conditions? I
→ need the results formatted and saved in a file named 'weather.csv'
→ following the requirement in tips.md.'''
1739
```

1740 **Listing 7: User's question.**
1741
17421743

```
import pandas as pd
from scipy.stats import kruskal
import os

# Define the file path for the input dataset
input_file_path = 'data/yulu_bike_sharing_dataset.csv'

# Load the dataset into a pandas DataFrame
yulu_df = pd.read_csv(input_file_path)

# The goal is to perform a Kruskal-Wallis H-test to compare the distribution of bike rentals ('count')
# across different weather conditions ('weather').

# The 'weather' column contains numerical codes for different weather conditions.
# First, we need to find the unique weather conditions in the dataset.
unique_weather_conditions = sorted(yulu_df['weather'].unique())

# Next, we create a list of data samples, where each sample is an array of 'count'
# values corresponding to a specific weather condition.
weather_groups = [yulu_df['count'][yulu_df['weather'] == weather] for weather in unique_weather_conditions]

# Perform the Kruskal-Wallis H-test. The * operator unpacks the list of groups
# into separate arguments for the function.
statistic, p_value = kruskal(*weather_groups)

# Define the significance level (alpha) as specified in the guidelines.
alpha = 0.05

# Based on the p-value, determine the hypothesis conclusion and the comment
# as per the format requirements in 'data/tips.md'.
if p_value < alpha:
    hypothesis_conclusion = "Strong Evidence to Reject Null Hypothesis"
    comment_on_results = "Different distributions"
else:
    hypothesis_conclusion = "No Evidence to Reject Null Hypothesis"
    comment_on_results = "No evidence of different distributions"

# Create a dictionary containing the results in the required format.
results_data = {
    'Test Type': ["Non Parametric"],
    'Kruskal Statistic': [statistic],
    'P-Value': [p_value],
    'Hypothesis': [hypothesis_conclusion],
    'Comment': [comment_on_results]
}

# Create a new pandas DataFrame from the results dictionary.
weather_results_df = pd.DataFrame(results_data)

# Define the output directory and file path.
output_dir = 'final'
output_file_path = os.path.join(output_dir, 'weather.csv')

# Ensure the output directory exists.
os.makedirs(output_dir, exist_ok=True)

# Save the resulting DataFrame to the specified CSV file,
# ensuring the index is not written to the file.
weather_results_df.to_csv(output_file_path, index=False)

# print(f"Analysis complete. Results saved to {output_file_path}")
# print(weather_results_df.to_string())
1779
```

1780 **Listing 8: Solution code.**
1781

1782

1783

Table 7: Generated output by DS-STAR in csv format.

1784

1785

1786

1787

1788

F.2 DATA INSIGHT

1789

1790

1791

1792

1793

1794

1795

1796

1797

1798

1799

1800

1801

1802

1803

1804

```
f'''Could you please analyze the matches between Argentina and Brazil?
→ Record the total number of matches, the number of wins for each
→ team, and the number of draws. Indicate if Brazil has won more
→ matches ("Yes" or "No"). Present your findings in the following
→ format:
"""
{
    "Total Matches between Argentina and Brazil": [...],
    "Arg Wins": [...],
    "Brazil Wins": [...],
    "Draws": [...],
    "Result": [...]
}
"""
'''
```

Listing 9: User's question.

1805

1806

1807

1808

1809

1810

1811

1812

1813

1814

1815

1816

1817

1818

1819

1820

1821

1822

1823

1824

1825

1826

1827

1828

1829

1830

1831

1832

1833

1834

1835

```
import pandas as pd
import numpy as np
import json
import os

# Ensure the 'final' directory exists for the output file
output_dir = 'final'
os.makedirs(output_dir, exist_ok=True)

# Load the results.csv file into a pandas DataFrame
results_df = pd.read_csv('data/results.csv')

# Filter for matches where the home team is Argentina and the away team is Brazil
arg_vs_bra = results_df[
    (results_df['home_team'] == 'Argentina') & (results_df['away_team'] == 'Brazil')
]

# Filter for matches where the home team is Brazil and the away team is Argentina
bra_vs_arg = results_df[
    (results_df['home_team'] == 'Brazil') & (results_df['away_team'] == 'Argentina')
]

# Concatenate the two DataFrames to get all matches between Argentina and Brazil
argentina_brazil_matches = pd.concat([arg_vs_bra, bra_vs_arg])

# Sort the combined DataFrame by date for a chronological view
argentina_brazil_matches = argentina_brazil_matches.sort_values(by='date').copy()

# Define the conditions for determining the winner based on scores
# A draw in regular/extra time is considered a draw, regardless of a penalty shootout outcome.
conditions = [
    argentina_brazil_matches['home_score'] > argentina_brazil_matches['away_score'],
    argentina_brazil_matches['away_score'] > argentina_brazil_matches['home_score']
]

# Define the corresponding outcomes (the winning team's name)
choices = [
    argentina_brazil_matches['home_team'],
    argentina_brazil_matches['away_team']
]

# Create the 'winner' column using np.select, with 'Draw' as the default
argentina_brazil_matches['winner'] = np.select(conditions, choices, default='Draw')

# Count the total wins for each team and the number of draws
win_counts = argentina_brazil_matches['winner'].value_counts()

# Calculate the total number of matches, wins for each team, and draws from the win_counts Series
total_matches = int(win_counts.sum())
argentina_wins = int(win_counts.get('Argentina', 0))
brazil_wins = int(win_counts.get('Brazil', 0))
draws = int(win_counts.get('Draw', 0))

# Determine the 'Result' value. 'Yes' if Brazil has more wins, otherwise 'No'.

```

```

1836     result_value = 'Yes' if brazil_wins > argentina_wins else 'No'
1837
1838     # Construct the dictionary to store the analysis results with the specified keys
1839     analysis_results = {
1840         "Total Matches between Argentina and Brazil": total_matches,
1841         "Arg Wins": argentina_wins,
1842         "Brazil Wins": brazil_wins,
1843         "Draws": draws,
1844         "Result": result_value
1845     }
1846
1847     # Define the output file path
1848     output_path = os.path.join(output_dir, 'result.json')
1849
1850     # Save the final results dictionary to a JSON file in the 'final' directory
1851     with open(output_path, 'w') as f:
1852         json.dump(analysis_results, f, indent=4)

```

Listing 10: Solution code.

```

1850     f'''{
1851         "Total Matches between Argentina and Brazil": [109],
1852         "Arg Wins": [40],
1853         "Brazil Wins": [43],
1854         "Draws": [26],
1855         "Result": ["Yes"]
1856     }'''

```

Listing 11: Generated output by DS-STAR, in json format.

F.3 DATA MANIPULATION

```

1860     f'''Calculate the total quantity sold and total sales revenue for each
1861     ↪ bike category from 2016 to 2018. Ensure that total sales revenue
1862     ↪ accounts for any discounts applied to each order. Write the results
1863     ↪ into result.csv, following the exact structure and formatting of
1864     ↪ sample_result.csv.'''

```

Listing 12: User's question.

```

1865
1866
1867
1868     import pandas as pd
1869     import os
1870
1871     # Ensure the output directory exists
1872     os.makedirs('final', exist_ok=True)
1873
1874     # Load the required data files into pandas DataFrames
1875     order_items_df = pd.read_csv('data/order_items.csv')
1876     products_df = pd.read_csv('data/products.csv')
1877     categories_df = pd.read_csv('data/categories.csv')
1878     sample_result_df = pd.read_csv('data/sample_result.csv')
1879
1880     # Calculate the sales revenue for each item, accounting for discounts.
1881     # sales_revenue = quantity * list_price * (1 - discount)
1882     order_items_df['sales_revenue'] = order_items_df['quantity'] * order_items_df['list_price'] * (1 -
1883     ↪ order_items_df['discount'])
1884
1885     # Merge the order items with product information to get model_year and category_id
1886     merged_df = pd.merge(order_items_df, products_df, on='product_id')
1887
1888     # Merge with category information to get category_name
1889     merged_df = pd.merge(merged_df, categories_df, on='category_id')
1890
1891     # Filter the DataFrame to include only products with model years from 2016 to 2018
1892     filtered_df = merged_df[(merged_df['model_year'] >= 2016) & (merged_df['model_year'] <= 2018)]
1893
1894     # Group the filtered data by 'category_name' and aggregate the total quantity and total sales revenue.
1895     category_summary = filtered_df.groupby('category_name').agg(
1896         total_quantity=('quantity', 'sum'),
1897         total_price=('sales_revenue', 'sum')
1898     ).reset_index()
1899
1900     # Use the sample_result.csv to define the structure and order of the final output.
1901     # Perform a left merge to ensure all categories from the sample are present in the final result, in the
1902     ↪ correct order.
1903     final_df = pd.merge(sample_result_df[['category_name']], category_summary, on='category_name', how='left')
1904
1905     # Fill any potential NaN values with 0 for categories that had no sales in the filtered period.
1906     final_df[['total_quantity', 'total_price']] = final_df[['total_quantity', 'total_price']].fillna(0)

```

```

1890
1891     # Ensure the column order matches the sample_result.csv file: ['total_quantity', 'total_price',
1892     # category_name']
1893     final_df = final_df[['total_quantity', 'total_price', 'category_name']]
1894
1895     # Convert the 'total_quantity' column to integer type as it represents a count of items.
1896     final_df['total_quantity'] = final_df['total_quantity'].astype(int)
1897
1898     # Define the output path
1899     output_path = 'final/result.csv'
1900
1901     # Save the final DataFrame to the specified path, without the index.
1902     final_df.to_csv(output_path, index=False)
1903
1904     print(f"Successfully generated '{output_path}'")
1905     print("Content of the generated file:")
1906     print(final_df)

```

Listing 13: Solution code.

Table 8: Generated output by DS-STAR in csv format.

	total_quantity	total_price	category_name
1908	1755	2715079.53	Mountain Bikes
1909	559	1665098.49	Road Bikes
1910	2063	995032.62	Cruisers Bicycles
1911	315	916684.78	Electric Bikes
1912	394	711011.84	Cyclocross Bicycles
1913	813	394020.10	Comfort Bicycles
1914	1179	292189.20	Children Bicycles

F.4 DATA WRANGLING

```

f'''Manage missing data in the NYC_Open_Data_Parking_Violations dataset
by either dropping or replacing it. Drop rows if necessary, and drop
columns only if most entries are empty. Replace missing values with
the mean, mode, or other appropriate methods. Save the cleaned data
to 'cleaned_data.csv'.'''

```

Listing 14: User's question.

```

1925
1926     import pandas as pd
1927     import numpy as np
1928     import os
1929
1930     # The user prompt mentions cleaning the 'NYC_Open_Data_Parking_Violations' dataset,
1931     # however, the provided data file is 'data/data.csv', which contains automobile data.
1932     # This script will proceed with cleaning the provided 'data/data.csv' file based on the
1933     # general data cleaning guidelines in the prompt.
1934
1935     # Define file paths
1936     input_file = 'data/data.csv'
1937     output_dir = 'final'
1938     output_file = os.path.join(output_dir, 'cleaned_data.csv')
1939
1940     # Create the output directory if it does not exist
1941     if not os.path.exists(output_dir):
1942         os.makedirs(output_dir)
1943
1944     # Load the data, interpreting '?' as Not a Number (NaN)
1945     df = pd.read_csv(input_file, na_values='?')
1946
1947     # --- Data Cleaning Steps ---
1948
1949     # Guideline: "Drop rows if necessary"
1950     # The 'price' column is crucial for most analyses. Rows with a missing price are dropped.
1951     df.dropna(subset=['price'], inplace=True)
1952
1953     # Guideline: "Replace missing values with the mean, mode, or other appropriate methods"
1954
1955     # Identify columns for mean and mode imputation
1956     # Numerical columns with missing values to be imputed with the mean
1957     numeric_cols_to_impute_mean = ['normalized-losses', 'bore', 'stroke', 'horsepower', 'peak-rpm']

```

```

1944 # Categorical columns with missing values to be imputed with the mode
1945 categorical_cols_to_impute_mode = ['num-of-doors']
1946
1947 # Impute numerical columns with the mean
1948 for col in numeric_cols_to_impute_mean:
1949     # Convert column to numeric type. This is necessary because the '?
1950     # values caused pandas to read them as 'object' type.
1951     df[col] = pd.to_numeric(df[col])
1952     # Calculate the mean of the column (NaNs are automatically excluded)
1953     mean_val = df[col].mean()
1954     # Fill the missing (NaN) values with the calculated mean
1955     df[col].fillna(mean_val, inplace=True)
1956
1957 # Impute categorical columns with the mode
1958 for col in categorical_cols_to_impute_mode:
1959     # Calculate the mode (most frequent value) of the column.
1960     # .mode() returns a Series, so we take the first element.
1961     mode_val = df[col].mode()[0]
1962     # Fill the missing (NaN) values with the calculated mode
1963     df[col].fillna(mode_val, inplace=True)
1964
1965 # Guideline: After handling missing values, ensure data types are correct.
1966 # The 'price' column was loaded as 'object' and needs to be converted to a numeric type.
1967 df['price'] = df['price'].astype(float)
1968
1969 # --- Save the Cleaned Data ---
1970
1971 # Save the final cleaned DataFrame to 'final/cleaned_data.csv'
1972 # The index is not saved as it's not part of the original data.
1973 df.to_csv(output_file, index=False)
1974
1975 # --- Verification ---
1976 # This part confirms that the cleaning was successful and the file was created.
1977 print(f"Successfully cleaned the dataset and saved it to '{output_file}'")
1978
1979 # Load the cleaned data to verify its integrity
1980 cleaned_df = pd.read_csv(output_file)
1981
1982 print("\n--- Verification of Cleaned Data ---")
1983 print(f"Shape of the cleaned data: {cleaned_df.shape}")
1984
1985 # Check for any remaining missing values
1986 missing_values_summary = cleaned_df.isnull().sum()
1987 total_missing = missing_values_summary.sum()
1988
1989 print("\nMissing values per column in the cleaned file:")
1990 print(missing_values_summary)
1991 print(f"\nTotal remaining missing values: {total_missing}")
1992
1993 if total_missing == 0:
1994     print("\nVerification successful: The cleaned dataset has no missing values.")
1995 else:
1996     print("\nVerification failed: The cleaned dataset still contains missing values.")
1997

```

Listing 15: Solution code.

Table 9: Generated output by DS-STAR in csv format.

symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	width	height	curb-weight	engine-type	num-of-cylinders	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price
3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88.6	108.8	64.1	46.8	2548	dohc	four	130	3.8	3.68	9.0	111	5000	21	27	16500	
3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	94.5	171.2	65.5	52.4	2823	ohcv	six	152	3.6	3.47	9.0	154	5000	19	26	16500	
1	122.0	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	171.2	65.5	52.4	2823	ohcv	six	152	3.6	3.47	9.0	154	5000	19	26	16500	

F.5 MACHINE LEARNING

```

f'''As a contestant in the Bank customer data churn prediction
→ competition, you need to predict the data in test.csv according to
→ the competition requirements. Write the results into submission.csv
→ following the format specified in sample_submission.csv.'''

```

Listing 16: User's question.

```

1993 import pandas as pd
1994 import numpy as np
1995 from sklearn.model_selection import StratifiedKFold
1996 import lightgbm as lgb
1997 from sklearn.metrics import roc_auc_score
1998 import optuna
1999 import os
2000
2001 # Create the 'final' directory if it doesn't already exist to store the output.

```

```

1998 os.makedirs('final', exist_ok=True)
1999
2000 # Suppress Optuna's logging to keep the output clean during the study.
2001 optuna.logging.set_verbosity(optuna.logging.WARNING)
2002
2003 # 1. Load All Data
2004 # Load data from the 'data/' directory.
2005 original_train_df = pd.read_csv('data/train.csv')
2006 test_df = pd.read_csv('data/test.csv')
2007 additional_data_df = pd.read_csv('data/Churn_Modelling.csv')
2008
2009 # The provided test.csv uses 'CustomerId' as the unique identifier.
2010 # We will use 'CustomerId' for the submission file's 'id' column as required by the competition format.
2011 test_ids = test_df['CustomerId']
2012
2013 # 2. Preprocess Data: Augment and Clean
2014 # Select a consistent set of feature columns and the target variable ('Exited') to ensure uniformity across
2015 # datasets.
2016 feature_cols = ['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
2017 # 'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Exited']
2018
2019 # Augment the training data by combining the original training set with the additional 'Churn_Modelling.csv'
2020 # dataset.
2021 train_df = pd.concat([original_train_df[feature_cols], additional_data_df[feature_cols]], ignore_index=True)
2022
2023 # --- START: Feature Engineering ---
2024 def feature_engineer(df):
2025     """Creates new features based on domain knowledge and interactions between existing features."""
2026     # A small epsilon is added to denominators to prevent division by zero errors.
2027     epsilon = 1e-6
2028     df['BalanceSalaryRatio'] = df['Balance'] / (df['EstimatedSalary'] + epsilon)
2029     df['TenureByAge'] = df['Tenure'] / (df['Age'] + epsilon)
2030     df['CreditScoreGivenAge'] = df['CreditScore'] / (df['Age'] + epsilon)
2031     df['Loyalty'] = df['Tenure'] * df['IsActiveMember']
2032     df['ProductsPerTenure'] = df['NumOfProducts'] / (df['Tenure'] + epsilon)
2033     return df
2034
2035 print("Applying feature engineering to training and test data...")
2036 train_df = feature_engineer(train_df)
2037 test_df = feature_engineer(test_df)
2038 # --- END: Feature Engineering ---
2039
2040 # Drop identifier columns ('CustomerId', 'Surname', 'id') that are not useful for the model.
2041 # 'errors='ignore'' handles cases where a column might not exist in a dataframe.
2042 drop_cols = ['CustomerId', 'Surname', 'id']
2043 train_df = train_df.drop(columns=[col for col in drop_cols if col in train_df.columns and col != 'Exited'],
2044 # errors='ignore')
2045 test_df = test_df.drop(columns=[col for col in drop_cols if col in test_df.columns], errors='ignore')
2046
2047 # One-Hot Encode categorical features ('Geography', 'Gender') to convert them into a machine-readable
2048 # format.
2049 categorical_features = ['Geography', 'Gender']
2050 train_df = pd.get_dummies(train_df, columns=categorical_features, drop_first=True)
2051 test_df = pd.get_dummies(test_df, columns=categorical_features, drop_first=True)
2052
2053 # Separate the target variable ('Exited') from the training features.
2054 train_labels = train_df['Exited']
2055 train_df = train_df.drop(columns=['Exited'])
2056
2057 # Align the training and test dataframes to ensure they have the exact same feature columns in the same
2058 # order.
2059 # 'inner' join keeps only columns that are present in both dataframes.
2060 train_df, test_df = train_df.align(test_df, join='inner', axis=1)
2061
2062 # Define the final feature set (X) and target (y).
2063 X = train_df
2064 y = train_labels
2065
2066 # 3. Define the Objective Function for Optuna Hyperparameter Tuning
2067 def objective(trial):
2068     """Defines the hyperparameter search space and evaluation metric for Optuna to optimize."""
2069     # Define the hyperparameter search space for the LightGBM model.
2070     params = {
2071         'objective': 'binary',
2072         'metric': 'auc',
2073         'boosting_type': 'gbdt',
2074         'n_estimators': trial.suggest_int('n_estimators', 2000, 10000),
2075         'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.05),
2076         'num_leaves': trial.suggest_int('num_leaves', 20, 100),
2077         'max_depth': trial.suggest_int('max_depth', 5, 10),
2078         'seed': 42,
2079         'n_jobs': -1,
2080         'verbose': -1,
2081         'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 0.9),
2082         'subsample': trial.suggest_float('subsample', 0.5, 0.9),
2083         'reg_alpha': trial.suggest_float('reg_alpha', 0.01, 0.5),
2084         'reg_lambda': trial.suggest_float('reg_lambda', 0.01, 0.5),
2085     }

```

```

2053 N_SPLITS = 5
2054 skf = StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=42)
2055 oof_auc_scores = []
2056
2057 # Perform cross-validation to get a robust estimate of the model's performance for a given set of
2058 # hyperparameters.
2059 for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
2060     X_train, y_train = X.iloc[train_idx], y.iloc[train_idx]
2061     X_val, y_val = X.iloc[val_idx], y.iloc[val_idx]
2062
2063     model = lgb.LGBMClassifier(**params)
2064     model.fit(X_train, y_train,
2065                eval_set=[(X_val, y_val)],
2066                eval_metric='auc',
2067                callbacks=[lgb.early_stopping(100, verbose=False)])
2068
2069     val_preds = model.predict_proba(X_val)[:, 1]
2070     auc = roc_auc_score(y_val, val_preds)
2071     oof_auc_scores.append(auc)
2072
2073 # Return the mean AUC score, which Optuna will try to maximize.
2074 return np.mean(oof_auc_scores)
2075
2076 # 4. Run the Optimization Study
2077 print("Running hyperparameter optimization...")
2078 study = optuna.create_study(direction='maximize')
2079 # The number of trials can be adjusted; 50 provides a reasonable balance between search time and
2080 # performance.
2081 study.optimize(objective, n_trials=50)
2082 best_params = study.best_params
2083 print(f"\nBest trial parameters: {best_params}")
2084 print(f"Best trial mean OOF AUC: {study.best_value}")
2085
2086 # 5. Train Final Model with Best Parameters and Create Submission
2087 print("\nTraining final model with tuned parameters...")
2088 N_SPLITS = 5
2089 skf = StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=42)
2090 test_preds = np.zeros(len(test_df))
2091 oof_auc_scores = []
2092
2093 # Combine the best parameters found by Optuna with fixed model settings.
2094 final_params = {
2095     'objective': 'binary', 'metric': 'auc', 'boosting_type': 'gbdt',
2096     'seed': 42, 'n_jobs': -1, 'verbose': -1, **best_params
2097 }
2098
2099 for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
2100     print(f"==== Fold {fold+1} ====")
2101     X_train, y_train = X.iloc[train_idx], y.iloc[train_idx]
2102     X_val, y_val = X.iloc[val_idx], y.iloc[val_idx]
2103
2104     model = lgb.LGBMClassifier(**final_params)
2105     model.fit(X_train, y_train,
2106                eval_set=[(X_val, y_val)],
2107                eval_metric='auc',
2108                callbacks=[lgb.early_stopping(100, verbose=False)])
2109
2110     val_preds = model.predict_proba(X_val)[:, 1]
2111     auc = roc_auc_score(y_val, val_preds)
2112     oof_auc_scores.append(auc)
2113     print(f"Fold {fold+1} AUC: {auc}")
2114
2115 # Accumulate predictions on the test set, averaging over the folds to improve robustness.
2116 test_preds += model.predict_proba(test_df)[:, 1] / N_SPLITS
2117
2118 # Print the final average validation score from the cross-validation.
2119 print(f"\nAverage OOF AUC with tuned parameters: {np.mean(oof_auc_scores)}")
2120
2121 # Create the submission DataFrame using the averaged test predictions.
2122 submission_df = pd.DataFrame({'id': test_ids, 'Exited': test_preds})
2123
2124 # Save the DataFrame to a CSV file in the 'final' directory for submission.
2125 submission_df.to_csv('final/submission.csv', index=False)
2126
2127 print("\nSubmission file created successfully at 'final/submission.csv'.")
2128 print("First 5 rows of the submission file:")
2129 print(submission_df.head())
2130
2131

```

Listing 17: Solution code and machine learning model.

2106

F.6 VISUALIZATION

2108

```

f'''Draw a stacked bar chart that displays the percentage of restaurants
→ offering online ordering options versus those not offering online
→ ordering options across different rating levels. Save this chart as
→ 'result.jpg'. The title of the image should be "Percentage of
→ Restaurants' Online Order Option by Rating", with the xlabel as
→ "Rating", and the ylabel as "Percentage of Online Orders."'''

```

2113

2114

2115

2116

```

import pandas as pd
import sqlite3
import zipfile
import os
import numpy as np
import matplotlib.pyplot as plt

def generate_chart():
    """
    Loads data from the Zomato SQLite database, cleans the 'rate' column,
    calculates the percentage of restaurants offering online ordering for each rating,
    and generates a stacked bar chart of the results, saving it to the 'final' directory.
    """

    # Define file paths
    zip_path = 'data/zomato_rawdata.sqlite.zip'
    db_name = 'zomato_rawdata.sqlite'
    extract_path = '.' # Extract to the current directory
    output_dir = 'final'
    output_path = os.path.join(output_dir, 'result.jpg')

    # --- Step 1: Create output directory if it doesn't exist ---
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # --- Step 2: Extract the database from the zip file ---
    db_path = os.path.join(extract_path, db_name)
    with zipfile.Zipfile(zip_path, 'r') as zip_ref:
        # Extract the database file, ensuring the correct member is chosen
        for member in zip_ref.infolist():
            if db_name in member.filename:
                # To handle cases where the zip file has directory structures
                member.filename = os.path.basename(member.filename)
                zip_ref.extract(member, path=extract_path)
                break

    # --- Step 3: Load data from the SQLite database ---
    conn = None
    try:
        conn = sqlite3.connect(db_path)
        # Select only the columns needed for this analysis
        query = "SELECT rate, online_order FROM Users"
        df = pd.read_sql_query(query, conn)
    finally:
        if conn:
            conn.close()

    # --- Step 4: Clean the 'rate' column ---
    # Make a copy to avoid SettingWithCopyWarning
    df = df.copy()

    # Replace special non-numeric strings 'NEW' and '-' with numpy.nan
    df['rate'] = df['rate'].replace(['NEW', '-'], np.nan)

    # Remove the '/5' suffix from the rating strings
    df['rate'] = df['rate'].str.replace('/5', '').str.strip()

    # Convert the cleaned column to a numeric (float) type
    df['rate'] = pd.to_numeric(df['rate'], errors='coerce')

    # Drop rows where 'rate' or 'online_order' is NaN as they are not useful for this analysis
    df.dropna(subset=['rate', 'online_order'], inplace=True)

    # --- Step 5: Aggregate the data for plotting ---
    # Create a cross-tabulation to get counts of 'Yes' and 'No' for each rating
    online_order_counts = pd.crosstab(df['rate'], df['online_order'])

    # Calculate the percentage by dividing each row by its sum
    rate_online_percentage_df = online_order_counts.div(online_order_counts.sum(axis=1), axis=0) * 100

    # --- Step 6: Generate and save the stacked bar chart ---
    # Create the plot figure and axes
    fig, ax = plt.subplots(figsize=(14, 8))

    # Generate the stacked bar chart
    rate_online_percentage_df.plot(kind='bar', stacked=True, ax=ax)

```

```

2160
2161     # Set the title and labels as required
2162     ax.set_title("Percentage of Restaurants' Online Order Option by Rating", fontsize=16)
2163     ax.set_xlabel("Rating", fontsize=12)
2164     ax.set_ylabel("Percentage of Online Orders", fontsize=12)
2165
2166     # Improve aesthetics and readability
2167     ax.tick_params(axis='x', rotation=90)
2168     ax.legend(title="Online Order")
2169     ax.grid(axis='y', linestyle='--', alpha=0.7)
2170
2171     # Adjust layout to prevent labels from being cut off
2172     plt.tight_layout()
2173
2174     # Save the resulting chart to the final directory
2175     plt.savefig(output_path)
2176
2177     # Close the plot to free up memory
2178     plt.close(fig)
2179
2180     print(f"Chart has been generated and saved as '{output_path}'")
2181
2182     # --- Step 7: Clean up the extracted file ---
2183     if os.path.exists(db_path):
2184         os.remove(db_path)
2185
2186
2187     # Execute the function to generate the chart
2188     generate_chart()
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213

```

Listing 19: Solution code.

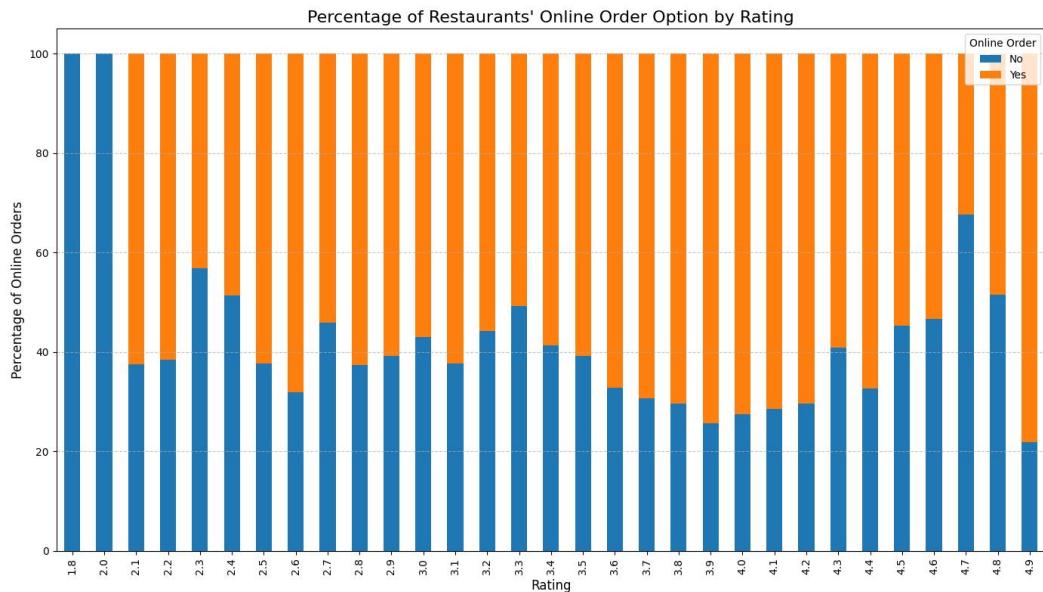


Figure 5: Generated bar chart.

2214 **G FULL EXECUTION LOGS OF DS-STAR**
2215

2216 In this section, we provide full logs of DS-STAR to better understand its behavior.

2217

```
f'''In February 2023 what delta would Rafa_AI pay if the relative fee of
2218   ↪ the fee with ID=17 changed to 99?'''
```

2219

2220

2221

2222

2223

```
# Step 1
f'''First, I will filter the `payments.csv` data to select only the
→ transactions for the merchant 'Rafa_AI' that occurred in February
→ 2023. Since 2023 is not a leap year, February corresponds to the
→ days of the year from 32 to 59.''''
```

2224

2225

2226

Listing 20: User's question.

2227

2228

2229

2230

```
import pandas as pd

# Load the payments data
try:
    payments_df = pd.read_csv('data/payments.csv')
except FileNotFoundError:
    print("Error: 'data/payments.csv' not found. Please ensure the file is in the correct directory.")
    exit()

# Define the filter for the merchant 'Rafa_AI'
merchant_filter = payments_df['merchant'] == 'Rafa_AI'

# Define the filter for February 2023.
# 2023 is not a leap year. January has 31 days.
# February runs from day 32 (31+1) to day 59 (31+28).
date_filter = (payments_df['day_of_year'] >= 32) & (payments_df['day_of_year'] <= 59)

# Apply both filters to the DataFrame
rafa_ai_feb_transactions = payments_df[merchant_filter & date_filter]

# Display the resulting filtered DataFrame
print(rafa_ai_feb_transactions)
```

2231

2232

2233

2234

2235

2236

2237

2238

2239

2240

2241

2242

2243

Listing 21: Round 0: Initial plan.

2244

2245

2246

2247

2248

```
f'''Insufficient'''
```

2249

2250

2251

2252

```
f'''Add Step'''
```

2253

2254

2255

2256

2257

2258

2259

2260

2261

2262

2263

2264

2265

2266

2267

Listing 22: Round 0: Initial implementation.

2268

2269

2270

2271

2272

2273

2274

2275

2276

2277

2278

2279

2280

2281

2282

2283

2284

2285

2286

2287

2288

2289

2290

2291

2292

2293

2294

2295

2296

2297

2298

2299

2300

2301

2302

2303

2304

2305

2306

2307

2308

2309

2310

2311

2312

2313

2314

2315

2316

2317

2318

2319

2320

2321

2322

2323

2324

2325

2326

2327

2328

2329

2330

2331

2332

2333

2334

2335

2336

2337

2338

2339

2340

2341

2342

2343

2344

2345

2346

2347

2348

2349

2350

2351

2352

2353

2354

2355

2356

2357

2358

2359

2360

2361

2362

2363

2364

2365

2366

2367

2368

2369

2370

2371

2372

2373

2374

2375

2376

2377

2378

2379

2380

2381

2382

2383

2384

2385

2386

2387

2388

2389

2390

2391

2392

2393

2394

2395

2396

2397

2398

2399

2400

2401

2402

2403

2404

2405

2406

2407

2408

2409

2410

2411

2412

2413

2414

2415

2416

2417

2418

2419

2420

2421

2422

2423

2424

2425

2426

2427

2428

2429

2430

2431

2432

2433

2434

2435

2436

2437

2438

2439

2440

2441

2442

2443

2444

2445

2446

2447

2448

2449

2450

2451

2452

2453

2454

2455

2456

2457

2458

2459

2460

2461

2462

2463

2464

2465

2466

2467

2468

2469

2470

2471

2472

2473

2474

2475

2476

2477

2478

2479

2480

2481

2482

2483

2484

2485

2486

2487

2488

2489

2490

2491

2492

2493

2494

2495

2496

2497

2498

2499

2500

2501

2502

2503

2504

2505

2506

2507

2508

2509

2510

2511

2512

2513

2514

2515

2516

2517

2518

2519

2520

2521

2522

2523

2524

2525

```

2268
2269 # Step 1
2270 f'''First, I will filter the `payments.csv` data to select only the
2271 → transactions for the merchant 'Rafa_AI' that occurred in February
2272 → 2023. Since 2023 is not a leap year, February corresponds to the
2273 → days of the year from 32 to 59.'''
2274
2275 # Step 2
2276 f'''I will calculate the original total fee paid by Rafa_AI in February
2277 → 2023. This involves loading the `merchant_data.json` and `fees.json` 
2278 → files, enriching the transaction data with all attributes required
2279 → for fee matching (including merchant details, monthly volume,
2280 → monthly fraud rate, and intracountry status), then applying the
2281 → corresponding fee rule to each transaction and summing the
2282 → calculated fees..'''
```

Listing 25: Round 1: Plan.

```

2282
2283
2284
2285 import pandas as pd
2286 import json
2287
2288 # --- 1. Load All Necessary Data ---
2289 try:
2290     payments_df = pd.read_csv('data/payments.csv')
2291     merchant_data_df = pd.read_json('data/merchant_data.json')
2292     fees_df = pd.read_json('data/fees.json')
2293     acquirer_countries_df = pd.read_csv('data/acquirer_countries.csv')
2294 except FileNotFoundError as e:
2295     print(f"Error loading data: {e}. Please ensure all data files are in the 'data/' directory.")
2296     exit()
2297
2298 # --- 2. Filter Transactions (Based on Previous Plan) ---
2299 # Define the filter for the merchant 'Rafa_AI'
2300 merchant_filter = payments_df['merchant'] == 'Rafa_AI'
2301
2302 # Define the filter for February 2023.
2303 # 2023 is not a leap year. January has 31 days.
2304 # February runs from day 32 (31+1) to day 59 (31+28).
2305 date_filter = (payments_df['day_of_year'] >= 32) & (payments_df['day_of_year'] <= 59)
2306
2307 # Apply both filters to the DataFrame and create a copy to avoid warnings
2308 rafa_ai_feb_transactions = payments_df[merchant_filter & date_filter].copy()
2309
2310 # --- 3. Enrich Transaction Data with Merchant and Acquirer Information ---
2311
2312 # Get Rafa_AI's specific merchant data
2313 rafa_ai_merchant_info = merchant_data_df[merchant_data_df['merchant'] == 'Rafa_AI'].iloc[0]
2314
2315 # Add merchant-specific attributes to the transaction DataFrame
2316 rafa_ai_feb_transactions['account_type'] = rafa_ai_merchant_info['account_type']
2317 rafa_ai_feb_transactions['capture_delay'] = str(rafa_ai_merchant_info['capture_delay']) # Ensure consistent
2318 → type
2319 rafa_ai_feb_transactions['merchant_category_code'] = rafa_ai_merchant_info['merchant_category_code']
2320
2321 # FIX: The original code failed on the next line.
2322 # The error "ValueError: Length of values (1) does not match length of index (2126)"
2323 # suggests that rafa_ai_merchant_info['acquirer'] is a list-like object with one element
2324 # (e.g., ['pringotts']) instead of a scalar string. Assigning a list of length 1 to a
2325 # column of a different length causes this error.
2326 # The fix is to extract the scalar string from the list before assigning it.
2327 acquirer_name = rafa_ai_merchant_info['acquirer']
2328 if isinstance(acquirer_name, list):
2329     # If it's a list, take the first element. If empty, it becomes None.
2330     acquirer_name = acquirer_name[0] if acquirer_name else None
2331
2332 # Now assign the scalar acquirer name to the new column.
2333 rafa_ai_feb_transactions['acquirer'] = acquirer_name
2334
2335 # Create a mapping from acquirer name to country code
2336 acquirer_country_map = acquirer_countries_df.set_index('acquirer')['country_code']
2337
2338 # Add acquirer country to the transaction data
2339 rafa_ai_feb_transactions['acquirer_country'] =
2340 → rafa_ai_feb_transactions['acquirer'].map(acquirer_country_map)
2341
2342 # Determine if each transaction is intracountry
2343 rafa_ai_feb_transactions['intracountry'] = rafa_ai_feb_transactions['issuing_country'] ==
2344 → rafa_ai_feb_transactions['acquirer_country']
2345
2346 # --- 4. Calculate Monthly Aggregates for Fee Matching ---
2347
2348 # Calculate total transaction volume for Rafa_AI in February
```

```

2322 monthly_volume = rafa_ai_feb_transactions['eur_amount'].sum()
2323
2324 # Calculate fraudulent transaction volume for the same period
2325 fraudulent_volume =
2326   ↪ rafa_ai_feb_transactions[rafa_ai_feb_transactions['has_fraudulent_dispute']] ['eur_amount'].sum()
2327
2328 # Calculate the monthly fraud rate as a percentage
2329 # Avoid division by zero if there's no volume
2330 monthly_fraud_rate = (fraudulent_volume / monthly_volume) if monthly_volume > 0 else 0
2331 monthly_fraud_percentage = monthly_fraud_rate * 100
2332
2333 # Add these aggregates to each transaction row for easy lookup during fee matching
2334 rafa_ai_feb_transactions['monthly_volume'] = monthly_volume
2335 rafa_ai_feb_transactions['monthly_fraud_percentage'] = monthly_fraud_percentage
2336
2337 # --- 5. Implement Fee Matching Logic ---
2338
2339 def parse_numeric_value(s):
2340   """Helper to parse strings like '100k' or '1m' into numbers."""
2341   s = s.strip().lower()
2342   if 'k' in s:
2343     return float(s.replace('k', '')) * 1_000
2344   if 'm' in s:
2345     return float(s.replace('m', '')) * 1_000_000
2346   return float(s)
2347
2348 def check_volume_rule(rule_str, value):
2349   """Checks if a value fits a volume rule string (e.g., '100k-1m')."""
2350   if pd.isnull(rule_str): return True
2351   rule_str = rule_str.replace(' ', '')
2352   if '=' in rule_str:
2353     low_str, high_str = rule_str.split('-')
2354     return parse_numeric_value(low_str) <= value < parse_numeric_value(high_str)
2355   if '>' in rule_str:
2356     return value >= parse_numeric_value(rule_str.replace('>', ''))
2357   if '<' in rule_str:
2358     return value < parse_numeric_value(rule_str.replace('<', ''))
2359   return False
2360
2361 def check_fraud_rule(rule_str, value):
2362   """Checks if a value fits a fraud rule string (e.g., '7.7%-8.3%')."""
2363   if pd.isnull(rule_str): return True
2364   rule_str = rule_str.replace('%', '')
2365   if '=' in rule_str:
2366     low_str, high_str = rule_str.split('-')
2367     return float(low_str) <= value < float(high_str)
2368   if '>' in rule_str:
2369     return value >= float(rule_str.replace('>', ''))
2370   if '<' in rule_str:
2371     return value < float(rule_str.replace('<', ''))
2372   return False
2373
2374 def find_matching_fee_rule(transaction, all_rules_df):
2375   """Iterates through fee rules to find the first one that matches the transaction."""
2376   for _, rule in all_rules_df.iterrows():
2377     # A rule with an empty list for a category applies to all values in that category
2378     if rule['card_scheme'] != transaction['card_scheme']: continue
2379     if rule['account_type'] and transaction['account_type'] not in rule['account_type']: continue
2380     if rule['merchant_category_code'] and transaction['merchant_category_code'] not in
2381       ↪ rule['merchant_category_code']: continue
2382     if rule['aci'] and transaction['aci'] not in rule['aci']: continue
2383
2384     # A rule with a null value for a category applies to all values
2385     if pd.notnull(rule['is_credit']) and rule['is_credit'] != transaction['is_credit']: continue
2386     if pd.notnull(rule['intracountry']) and rule['intracountry'] != transaction['intracountry']:
2387       ↪ continue
2388     if pd.notnull(rule['capture_delay']) and rule['capture_delay'] != transaction['capture_delay']:
2389       ↪ continue
2390
2391     # Check complex range-based rules
2392     if not check_volume_rule(rule['monthly_volume'], transaction['monthly_volume']): continue
2393     if not check_fraud_rule(rule['monthly_fraud_level'], transaction['monthly_fraud_percentage']):
2394       ↪ continue
2395
2396     # If all checks pass, this is the correct rule
2397     return rule['fixed_amount'], rule['rate']
2398
2399   # Return default/zero fees if no rule is found (should not happen with complete data)
2400   return 0, 0
2401
2402 # --- 6. Calculate Fees and Final Sum ---
2403
2404 # Apply the matching function to each transaction to get its specific fee parameters
2405 fee_params = rafa_ai_feb_transactions.apply(
2406   lambda row: find_matching_fee_rule(row, fees_df),
2407   axis=1
2408 )
2409
2410 # Assign the found fee parameters back to the DataFrame
2411 rafa_ai_feb_transactions[['fixed_amount', 'rate']] = pd.DataFrame(
2412   fee_params.tolist(),
2413   index=rafa_ai_feb_transactions.index
2414 )

```

```

2376     index=rafa_ai_feb_transactions.index
2377 )
2378
2379     # Calculate the fee for each transaction using the formula from the manual
2380     # fee = fixed_amount + rate * transaction_value / 10000
2381     rafa_ai_feb_transactions['fee'] = rafa_ai_feb_transactions['fixed_amount'] + \
2382         (rafa_ai_feb_transactions['rate'] * rafa_ai_feb_transactions['eur_amount']) / \
2383         10000
2384
2385     # Calculate the total fee by summing up the fees for all transactions
2386     total_original_fee = rafa_ai_feb_transactions['fee'].sum()
2387
2388     # Display the final result
2389     print(f"The original total fee paid by Rafa_AI in February 2023 was: {total_original_fee:.2f} EUR")
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429

```

Listing 26: **Round 1: Implementation.**

```
f'''Insufficient'''
```

Listing 27: **Round 1: Validation.**

```
f'''Add Step'''
```

Listing 28: **Round 1: Routing.** Add Step 3 to the current plan.

```

# Step 1
f'''First, I will filter the `payments.csv` data to select only the
→ transactions for the merchant 'Rafa_AI' that occurred in February
→ 2023. Since 2023 is not a leap year, February corresponds to the
→ days of the year from 32 to 59.'''
# Step 2
f'''I will calculate the original total fee paid by Rafa_AI in February
→ 2023. This involves loading the `merchant_data.json` and `fees.json`
→ files, enriching the transaction data with all attributes required
→ for fee matching (including merchant details, monthly volume,
→ monthly fraud rate, and intracountry status), then applying the
→ corresponding fee rule to each transaction and summing the
→ calculated fees.'''
# Step 3
f'''I will calculate the new total fee for Rafa_AI's February 2023
→ transactions using the modified fee structure where the fee rule
→ with ID=17 has its `rate` changed to 99. Then, I will subtract the
→ original total fee (778.52 EUR) from this new total fee to find the
→ requested delta.'''

```

Listing 29: **Round 2: Plan.**

```

import pandas as pd
import json

# --- 1. Load All Necessary Data ---
try:
    payments_df = pd.read_csv('data/payments.csv')
    merchant_data_df = pd.read_json('data/merchant_data.json')
    fees_df = pd.read_json('data/fees.json')
    acquirer_countries_df = pd.read_csv('data/acquirer_countries.csv')
except FileNotFoundError as e:
    print(f"Error loading data: {e}. Please ensure all data files are in the 'data/' directory.")
    exit()

# --- 2. Filter Transactions (Based on Previous Plan) ---
# Define the filter for the merchant 'Rafa_AI'
merchant_filter = payments_df['merchant'] == 'Rafa_AI'

# Define the filter for February 2023.
# 2023 is not a leap year. January has 31 days.
# February runs from day 32 (31+1) to day 59 (31+28).

```

```

2430 date_filter = (payments_df['day_of_year'] >= 32) & (payments_df['day_of_year'] <= 59)
2431
2432 # Apply both filters to the DataFrame and create a copy to avoid warnings
2433 rafa_ai_feb_transactions = payments_df[merchant_filter & date_filter].copy()
2434
2435 # --- 3. Enrich Transaction Data with Merchant and Acquirer Information ---
2436
2437 # Get Rafa_AI's specific merchant data
2438 rafa_ai_merchant_info = merchant_data_df[merchant_data_df['merchant'] == 'Rafa_AI'].iloc[0]
2439
2440 # Add merchant-specific attributes to the transaction DataFrame
2441 rafa_ai_feb_transactions['account_type'] = rafa_ai_merchant_info['account_type']
2442 rafa_ai_feb_transactions['capture_delay'] = str(rafa_ai_merchant_info['capture_delay']) # Ensure consistent
2443 ↪ type
2444 rafa_ai_feb_transactions['merchant_category_code'] = rafa_ai_merchant_info['merchant_category_code']
2445
2446 # FIX: The original code failed on the next line.
2447 # The error "ValueError: Length of values (1) does not match length of index (2126)"
2448 # suggests that rafa_ai_merchant_info['acquirer'] is a list-like object with one element
2449 # (e.g., ['gringotts']) instead of a scalar string. Assigning a list of length 1 to a
2450 # column of a different length causes this error.
2451 # The fix is to extract the scalar string from the list before assigning it.
2452 acquirer_name = rafa_ai_merchant_info['acquirer']
2453 if isinstance(acquirer_name, list):
2454     # If it's a list, take the first element. If empty, it becomes None.
2455     acquirer_name = acquirer_name[0] if acquirer_name else None
2456
2457 # Now assign the scalar acquirer name to the new column.
2458 rafa_ai_feb_transactions['acquirer'] = acquirer_name
2459
2460 # Create a mapping from acquirer name to country code
2461 acquirer_country_map = acquirer_countries_df.set_index('acquirer')['country_code']
2462
2463 # Add acquirer country to the transaction data
2464 rafa_ai_feb_transactions['acquirer_country'] =
2465 ↪ rafa_ai_feb_transactions['acquirer'].map(acquirer_country_map)
2466
2467 # Determine if each transaction is intracountry
2468 rafa_ai_feb_transactions['intracountry'] = rafa_ai_feb_transactions['issuing_country'] ==
2469 ↪ rafa_ai_feb_transactions['acquirer_country']
2470
2471 # --- 4. Calculate Monthly Aggregates for Fee Matching ---
2472
2473 # Calculate total transaction volume for Rafa_AI in February
2474 monthly_volume = rafa_ai_feb_transactions['eur_amount'].sum()
2475
2476 # Calculate fraudulent transaction volume for the same period
2477 fraudulent_volume =
2478 ↪ rafa_ai_feb_transactions[rafa_ai_feb_transactions['has_fraudulent_dispute']] ['eur_amount'].sum()
2479
2480 # Calculate the monthly fraud rate as a percentage
2481 # Avoid division by zero if there's no volume
2482 monthly_fraud_rate = (fraudulent_volume / monthly_volume) if monthly_volume > 0 else 0
2483 monthly_fraud_percentage = monthly_fraud_rate * 100
2484
2485 # Add these aggregates to each transaction row for easy lookup during fee matching
2486 rafa_ai_feb_transactions['monthly_volume'] = monthly_volume
2487 rafa_ai_feb_transactions['monthly_fraud_percentage'] = monthly_fraud_percentage
2488
2489 # --- 5. Implement Fee Matching Logic ---
2490
2491 def parse_numeric_value(s):
2492     """Helper to parse strings like '100k' or '1m' into numbers."""
2493     s = s.strip().lower()
2494     if 'k' in s:
2495         return float(s.replace('k', '')) * 1_000
2496     if 'm' in s:
2497         return float(s.replace('m', '')) * 1_000_000
2498     return float(s)
2499
2500 def check_volume_rule(rule_str, value):
2501     """Checks if a value fits a volume rule string (e.g., '100k-1m')."""
2502     if pd.isnull(rule_str): return True
2503     rule_str = rule_str.replace(' ', '')
2504     if '-' in rule_str:
2505         low_str, high_str = rule_str.split('-')
2506         return parse_numeric_value(low_str) <= value < parse_numeric_value(high_str)
2507     if '>' in rule_str:
2508         return value >= parse_numeric_value(rule_str.replace('>', ''))
2509     if '<' in rule_str:
2510         return value < parse_numeric_value(rule_str.replace('<', ''))
2511     return False
2512
2513 def check_fraud_rule(rule_str, value):
2514     """Checks if a value fits a fraud rule string (e.g., '7.7%-8.3%')."""
2515     if pd.isnull(rule_str): return True
2516     rule_str = rule_str.replace('%', '')
2517     if '-' in rule_str:
2518         low_str, high_str = rule_str.split('-')
2519         return float(low_str) <= value < float(high_str)

```

```

2484
2485     if '>' in rule_str:
2486         return value >= float(rule_str.replace('>', ''))
2487     if '<' in rule_str:
2488         return value < float(rule_str.replace('<', ''))
2489     return False
2490
2491     def find_matching_fee_rule(transaction, all_rules_df):
2492         """Iterates through fee rules to find the first one that matches the transaction."""
2493         for _, rule in all_rules_df.iterrows():
2494             # A rule with an empty list for a category applies to all values in that category
2495             if rule['card_scheme'] != transaction['card_scheme']: continue
2496             if rule['account_type'] and transaction['account_type'] not in rule['account_type']: continue
2497             if rule['merchant_category_code'] and transaction['merchant_category_code'] not in
2498                 rule['merchant_category_code']: continue
2499             if rule['aci'] and transaction['aci'] not in rule['aci']: continue
2500
2501             # A rule with a null value for a category applies to all values
2502             if pd.notnull(rule['is_credit']) and rule['is_credit'] != transaction['is_credit']: continue
2503             if pd.notnull(rule['intracountry']) and rule['intracountry'] != transaction['intracountry']:
2504                 continue
2505             if pd.notnull(rule['capture_delay']) and rule['capture_delay'] != transaction['capture_delay']:
2506                 continue
2507
2508             # Check complex range-based rules
2509             if not check_volume_rule(rule['monthly_volume'], transaction['monthly_volume']): continue
2510             if not check_fraud_rule(rule['monthly_fraud_level'], transaction['monthly_fraud_percentage']):
2511                 continue
2512
2513             # If all checks pass, this is the correct rule
2514             return rule['fixed_amount'], rule['rate']
2515
2516         # Return default/zero fees if no rule is found (should not happen with complete data)
2517         return 0, 0
2518
2519     # --- 6. Calculate Fees and Final Sum ---
2520
2521     # Apply the matching function to each transaction to get its specific fee parameters
2522     fee_params = rafa_ai_feb_transactions.apply(
2523         lambda row: find_matching_fee_rule(row, fees_df),
2524         axis=1
2525     )
2526
2527     # Assign the found fee parameters back to the DataFrame
2528     rafa_ai_feb_transactions[['fixed_amount', 'rate']] = pd.DataFrame(
2529         fee_params.tolist(),
2530         index=rafa_ai_feb_transactions.index
2531     )
2532
2533     # Calculate the fee for each transaction using the formula from the manual
2534     # fee = fixed_amount + rate * transaction_value / 10000
2535     rafa_ai_feb_transactions['fee'] = rafa_ai_feb_transactions['fixed_amount'] + \
2536         (rafa_ai_feb_transactions['rate'] * rafa_ai_feb_transactions['eur_amount'] / 10000)
2537
2538     # Calculate the total fee by summing up the fees for all transactions
2539     total_original_fee = rafa_ai_feb_transactions['fee'].sum()
2540
2541     # Display the final result from the previous step
2542     print(f"The original total fee paid by Rafa_AI in February 2023 was: {total_original_fee:.2f} EUR")
2543
2544     # --- 7. Calculate New Fee with Modified Rule ---
2545
2546     # Create a copy of the fees dataframes to avoid altering the original
2547     fees_df_modified = fees_df.copy()
2548
2549     # Implement the plan: change the rate for the rule with ID=17 to 99
2550     fees_df_modified.loc[fees_df_modified['ID'] == 17, 'rate'] = 99
2551
2552     # Apply the matching function again, this time with the modified fee rules
2553     new_fee_params = rafa_ai_feb_transactions.apply(
2554         lambda row: find_matching_fee_rule(row, fees_df_modified),
2555         axis=1
2556     )
2557
2558     # Assign the new fee parameters to new columns in the DataFrame
2559     rafa_ai_feb_transactions[['new_fixed_amount', 'new_rate']] = pd.DataFrame(
2560         new_fee_params.tolist(),
2561         index=rafa_ai_feb_transactions.index
2562     )
2563
2564     # Calculate the new fee for each transaction using the new rate
2565     rafa_ai_feb_transactions['new_fee'] = rafa_ai_feb_transactions['new_fixed_amount'] + \
2566         (rafa_ai_feb_transactions['new_rate'] *
2567          rafa_ai_feb_transactions['eur_amount'] / 10000)
2568
2569     # Calculate the new total fee by summing up the individual new fees
2570     new_total_fee = rafa_ai_feb_transactions['new_fee'].sum()
2571
2572     # Calculate the difference (delta) between the new total fee and the original total fee
2573     fee_delta = new_total_fee - total_original_fee

```

```

2538
2539     # --- 8. Display the Final Delta ---
2540     print(f"The new total fee with the modified rule would be: {new_total_fee:.2f} EUR")
2541     print(f"The requested delta (New Fee - Original Fee) is: {fee_delta:.2f} EUR")

```

Listing 30: Round 2: Implementation.

```

2543     f'''Insufficient'''

```

Listing 31: Round 2: Validation.

```

2548     f'''Step 3 is wrong'''

```

Listing 32: Round 2: Routing. Remove Step 3 from the current plan and add Step 3 again.

```

2553     # Step 1
2554     f'''First, I will filter the `payments.csv` data to select only the
2555     → transactions for the merchant 'Rafa_AI' that occurred in February
2556     → 2023. Since 2023 is not a leap year, February corresponds to the
2557     → days of the year from 32 to 59.'''
2558
2559     # Step 2
2560     f'''I will calculate the original total fee paid by Rafa_AI in February
2561     → 2023. This involves loading the `merchant_data.json` and `fees.json`
2562     → files, enriching the transaction data with all attributes required
2563     → for fee matching (including merchant details, monthly volume,
2564     → monthly fraud rate, and intracountry status), then applying the
2565     → corresponding fee rule to each transaction and summing the
2566     → calculated fees.'''
2567
2568     # Step 3
2569     f'''I will filter the transactions for 'Rafa_AI' in February 2023
2570     → (day_of_year from 32 to 59) that meet all the conditions of fee rule
2571     → ID=17: `card_scheme` must be 'SwiftCharge', `is_credit` must be
2572     → True, and `aci` must be 'A'. Then, for these matching transactions,
2573     → I will calculate the total fee difference using the formula: (99 -
2574     → 60) * SUM(eur_amount) / 10000.'''

```

Listing 33: Round 3: Plan.

```

2574
2575     import pandas as pd
2576     import json
2577
2578     # --- 1. Load All Necessary Data ---
2579     # This part is based on the loading logic from the base code.
2580     try:
2581         payments_df = pd.read_csv('data/payments.csv')
2582         merchant_data_df = pd.read_json('data/merchant_data.json')
2583         fees_df = pd.read_json('data/fees.json')
2584     except FileNotFoundError as e:
2585         print(f"Error loading data: {e}. Please ensure all data files are in the 'data/' directory.")
2586         exit()
2587
2588     # --- Implementation of the Current Plan ---
2589     # I will filter the transactions for 'Rafa_AI' in February 2023 (day_of_year from 32 to 59)
2590     # that meet all the conditions of fee rule ID=17: `card_scheme` must be 'SwiftCharge',
2591     # `is_credit` must be True, and `aci` must be 'A'. Then, for these matching transactions,
2592     # I will calculate the total fee difference using the formula:
2593     # (99 - 60) * SUM(eur_amount) / 10000.
2594
2595     print("--- Calculating fee difference for 'Rafa_AI' in Feb 2023 under specific conditions ---")
2596
2597     # Step 1: Define the conditions for filtering based on the plan.
2598     # February 2023 corresponds to day_of_year 32 to 59 (2023 is not a leap year).
2599     feb_start_day = 32
2600     feb_end_day = 59
2601     merchant_name = 'Rafa_AI'
2602     required_card_scheme = 'SwiftCharge'
2603     required_is_credit = True
2604     required_aci = 'A'

```

```

2592
2593     # Step 2: Filter the payments DataFrame to find transactions matching all conditions.
2594     matching_transactions = payments_df[
2595         (payments_df['merchant'] == merchant_name) &
2596         (payments_df['day_of_year'] >= feb_start_day) &
2597         (payments_df['day_of_year'] <= feb_end_day) &
2598         (payments_df['card_scheme'] == required_card_scheme) &
2599         (payments_df['is_credit'] == required_is_credit) &
2600         (payments_df['aci'] == required_aci)
2601     ]
2602
2603     print(f"\n[INFO] Found {len(matching_transactions)} transactions for '{merchant_name}' in February 2023
2604     ↪ matching all specified conditions.")
2605
2606     # Step 3: Calculate the total fee difference for these matching transactions.
2607     if not matching_transactions.empty:
2608         # Sum the 'eur_amount' for all matching transactions.
2609         total_matching_volume = matching_transactions['eur_amount'].sum()
2610         print(f"[INFO] Total volume of these transactions: {total_matching_volume:.2f} EUR")
2611
2612         # Apply the fee difference formula provided in the plan.
2613         # Formula: (new_rate - old_rate) * SUM(eur_amount) / 10000
2614         rate_difference = 99 - 60
2615         total_fee_difference = (rate_difference * total_matching_volume) / 10000
2616
2617         # --- Final Conclusion ---
2618         print("\n-----CONCLUSION-----")
2619         print(f"The total fee difference for the matching transactions is: {total_fee_difference:.4f} EUR.")
2620
2621     else:
2622         # --- Final Conclusion ---
2623         print("\n-----CONCLUSION-----")
2624         print("No transactions were found that match all the specified criteria.")
2625         print("Therefore, the total fee difference is 0.00 EUR.")

```

Listing 34: Round 3: Implementation.

```
f'''Insufficient'''
```

Listing 35: Round 3: Validation.

```
f'''Step 3 is wrong'''
```

Listing 36: Round 3: Routing. Remove Step 3 from the current plan and add Step 3 again.

```

2625     # Step 1
2626     f'''First, I will filter the `payments.csv` data to select only the
2627     ↪ transactions for the merchant 'Rafa_AI' that occurred in February
2628     ↪ 2023. Since 2023 is not a leap year, February corresponds to the
2629     ↪ days of the year from 32 to 59.'''
2630
2631     # Step 2
2632     f'''I will calculate the original total fee paid by Rafa_AI in February
2633     ↪ 2023. This involves loading the `merchant_data.json` and `fees.json`
2634     ↪ files, enriching the transaction data with all attributes required
2635     ↪ for fee matching (including merchant details, monthly volume,
2636     ↪ monthly fraud rate, and intracountry status), then applying the
2637     ↪ corresponding fee rule to each transaction and summing the
2638     ↪ calculated fees...'''
2639
2640     # Step 3
2641     f'''Calculate the fee delta by subtracting the original `rate` of fee ID
2642     ↪ 17 from the new `rate` of 99, multiplying the result by the total
2643     ↪ volume of the affected transactions (686.48 EUR), and then dividing
2644     ↪ by 10000.'''

```

Listing 37: Round 4: Plan.

```

2643
2644
2645     import pandas as pd
2646     import json

```

```

2646
2647     # --- 1. Load All Necessary Data ---
2648     # This part is based on the loading logic from the base code.
2649     try:
2650         payments_df = pd.read_csv('data/payments.csv')
2651         merchant_data_df = pd.read_json('data/merchant_data.json')
2652         fees_df = pd.read_json('data/fees.json')
2653     except FileNotFoundError as e:
2654         print(f"Error loading data: {e}. Please ensure all data files are in the 'data/' directory.")
2655         exit()
2656
2657     # --- Implementation of the Current Plan ---
2658     # Calculate the fee delta by subtracting the original 'rate' of fee ID 17 from the new 'rate' of 99,
2659     # multiplying the result by the total volume of the affected transactions (686.48 EUR),
2660     # and then dividing by 10000.
2661
2662     print("---- Calculating fee delta for affected 'Rafa_AI' transactions ---")
2663
2664     # Step 1: Define the constants given in the plan.
2665     new_rate = 99
2666     total_affected_volume = 686.48 # EUR
2667     fee_divisor = 10000
2668
2669     # Step 2: Find the original rate for fee ID 17 from the fees_df DataFrame.
2670     fee_rule_17 = fees_df[fees_df['ID'] == 17]
2671
2672     if fee_rule_17.empty:
2673         print("\n[ERROR] Fee rule with ID 17 was not found in fees.json.")
2674         original_rate = None # Set to None to prevent further calculation
2675     else:
2676         # Extract the 'rate' value from the filtered DataFrame.
2677         # .iloc[0] selects the first row of the result.
2678         original_rate = fee_rule_17['rate'].iloc[0]
2679         print(f"\n[INFO] Original rate for fee ID 17 is: {original_rate}")
2680         print(f"[INFO] New rate to be applied is: {new_rate}")
2681         print(f"[INFO] Total volume of affected transactions is: {total_affected_volume:.2f} EUR")
2682
2683     # Step 3: Calculate the fee delta using the specified formula.
2684     if original_rate is not None:
2685         # Calculate the difference in rates.
2686         rate_difference = new_rate - original_rate
2687
2688         # Apply the formula: (rate_difference * total_volume) / 10000
2689         fee_delta = (rate_difference * total_affected_volume) / fee_divisor
2690
2691         # --- Final Conclusion ---
2692         print("\n-----CONCLUSION-----")
2693         print(f"The calculated fee delta, by changing the rate from {original_rate} to {new_rate} for the")
2694         print(f"affected volume, is: {fee_delta:.4f} EUR.")
2695
2696     else:
2697         # --- Final Conclusion ---
2698         print("\n-----CONCLUSION-----")
2699         print("Could not perform calculation because the original fee rate for ID 17 was not found.")

```

Listing 38: **Round 4: Implementation.**

```
f'''Insufficient'''
```

Listing 39: **Round 4: Validation.**

```
f'''Add Step'''
```

Listing 40: **Round 4: Routing.** Add Step 4 to the current plan.

```

2695     # Step 1
2696     f'''First, I will filter the `payments.csv` data to select only the
2697     # transactions for the merchant 'Rafa_AI' that occurred in February
2698     # 2023. Since 2023 is not a leap year, February corresponds to the
2699     # days of the year from 32 to 59.'''

```

```

2700
2701 f'''I will calculate the original total fee paid by Rafa_AI in February
2702    ↵ 2023. This involves loading the `merchant_data.json` and `fees.json`
2703    ↵ files, enriching the transaction data with all attributes required
2704    ↵ for fee matching (including merchant details, monthly volume,
2705    ↵ monthly fraud rate, and intracountry status), then applying the
2706    ↵ corresponding fee rule to each transaction and summing the
2707    ↵ calculated fees..'''"
2708
2709 # Step 3
2710 f'''Calculate the fee delta by subtracting the original `rate` of fee ID
2711    ↵ 17 from the new `rate` of 99, multiplying the result by the total
2712    ↵ volume of the affected transactions (686.48 EUR), and then dividing
2713    ↵ by 10000.''''
2714
2715 # Step 4
2716 f'''Calculate the total original fee for Rafa_AI's transactions in
2717    ↵ February 2023 that are affected by fee ID=17, and then calculate the
2718    ↵ new total fee for the same transactions using the modified rate of
2719    ↵ 99. The delta is the difference between these two amounts.''''

```

Listing 41: Round 5: Plan.

```

2717
2718
2719
2720
2721 import pandas as pd
2722 import json
2723
2724 # --- 1. Load All Necessary Data ---
2725 # This part is based on the loading logic from the base code.
2726 try:
2727     payments_df = pd.read_csv('data/payments.csv')
2728     merchant_data_df = pd.read_json('data/merchant_data.json')
2729     fees_df = pd.read_json('data/fees.json')
2730 except FileNotFoundError as e:
2731     print(f"Error loading data: {e}. Please ensure all data files are in the 'data/' directory.")
2732     exit()
2733
2734 # --- Implementation of the Current Plan ---
2735 # Calculate the total original fee for Rafa_AI's transactions in February 2023
2736 # that are affected by fee ID=17, and then calculate the new total fee for the same
2737 # transactions using the modified rate of 99. The delta is the difference.
2738
2739 print("--- Identifying transactions for 'Rafa_AI' in February 2023 affected by fee rule ID 17 ---")
2740
2741 # Step 1: Filter payments for 'Rafa_AI' in February 2023.
2742 # February corresponds to days of the year from 32 to 59 (since 2023 is not a leap year).
2743 feb_transactions_df = payments_df[
2744     (payments_df['merchant'] == 'Rafa_AI') &
2745     (payments_df['day_of_year'] >= 32) &
2746     (payments_df['day_of_year'] <= 59)
2747 ].copy()
2748
2749 # Step 2: Enrich transaction data with merchant-specific information.
2750 # Get the specific data for 'Rafa_AI' from the merchant data file.
2751 rafa_ai_data = merchant_data_df[merchant_data_df['merchant'] == 'Rafa_AI']
2752
2753 # Merge the February transactions with the merchant's data.
2754 if not rafa_ai_data.empty and not feb_transactions_df.empty:
2755     # Use a left merge to add merchant details (like mcc, account_type) to each transaction.
2756     full_transaction_data = feb_transactions_df.merge(rafa_ai_data, on='merchant', how='left')
2757 else:
2758     print("\n[ERROR] No transactions found for 'Rafa_AI' in February or 'Rafa_AI' data not found.")
2759     exit()
2760
2761 # Step 3: Retrieve the specific conditions for the fee rule with ID=17.
2762 fee_rule_17 = fees_df[fees_df['ID'] == 17]
2763
2764 if fee_rule_17.empty:
2765     print("\n[ERROR] Fee rule with ID 17 was not found in fees.json.")
2766     exit()
2767 else:
2768     # Use .iloc[0] to get the rule as a Series
2769     fee_rule_17 = fee_rule_17.iloc[0]
2770
2771 # Step 4: Filter the transactions that match all conditions of the fee rule.
2772 # The manual states that a null or empty list in a rule field means it applies to all values.
2773 mask = pd.Series(True, index=full_transaction_data.index)
2774
2775 # Match card_scheme (string)
2776 if fee_rule_17['card_scheme']:
2777     mask &= (full_transaction_data['card_scheme'] == fee_rule_17['card_scheme'])
2778
2779 # Match account_type (list)

```

```

2754
2755     if fee_rule_17['account_type']: # An empty list evaluates to False, correctly skipping the filter
2756         mask &= full_transaction_data['account_type'].isin(fee_rule_17['account_type'])
2757
2758     # Match merchant_category_code (list)
2759     if fee_rule_17['merchant_category_code']:
2760         mask &= full_transaction_data['merchant_category_code'].isin(fee_rule_17['merchant_category_code'])
2761
2762     # Match aci (list)
2763     if fee_rule_17['aci']:
2764         mask &= full_transaction_data['aci'].isin(fee_rule_17['aci'])
2765
2766     # Match is_credit (boolean, but can be null)
2767     if fee_rule_17['is_credit'] is not None:
2768         mask &= (full_transaction_data['is_credit'] == fee_rule_17['is_credit'])
2769
2770     # Apply the combined mask to get the affected transactions
2771     affected_transactions_df = full_transaction_data[mask]
2772
2773     # Step 5: Calculate the original and new fees for the affected transactions.
2774     if affected_transactions_df.empty:
2775         print("\n[INFO] No transactions for 'Rafa_AI' in February match the criteria for fee rule ID 17.")
2776         total_original_fee = 0
2777         total_new_fee = 0
2778         fee_delta = 0
2779     else:
2780         total_affected_volume = affected_transactions_df['eur_amount'].sum()
2781         num_affected_transactions = len(affected_transactions_df)
2782
2783         print(f"\n[INFO] Found {num_affected_transactions} transactions matching the fee rule.")
2784         print(f"[INFO] Total volume of these transactions: ({total_affected_volume:.2f} EUR)")
2785
2786         # Get fee parameters from the rule
2787         original_rate = fee_rule_17['rate']
2788         fixed_amount = fee_rule_17['fixed_amount']
2789         new_rate = 99
2790         fee_divisor = 10000
2791
2792         # Calculate total original fee using the formula: fee = fixed_amount + rate * value / 10000
2793         # Vectorized calculation for efficiency
2794         total_fixed_component = fixed_amount * num_affected_transactions
2795         total_original_rate_component = (original_rate * total_affected_volume) / fee_divisor
2796         total_original_fee = total_fixed_component + total_original_rate_component
2797
2798         # Calculate total new fee with the modified rate
2799         total_new_rate_component = (new_rate * total_affected_volume) / fee_divisor
2800         total_new_fee = total_fixed_component + total_new_rate_component
2801
2802         # The delta is the difference between the new and original total fees
2803         fee_delta = total_new_fee - total_original_fee
2804
2805     # --- Final Conclusion ---
2806     print("\n-----CONCLUSION-----")
2807     print(f"Total original fee for affected transactions: ({total_original_fee:.4f} EUR)")
2808     print(f"Total new fee with rate={new_rate}: ({total_new_fee:.4f} EUR)")
2809     print(f"The calculated fee delta is: {fee_delta:.4f} EUR.")
2810     print("This represents the total increase in fees for Rafa_AI's affected February transactions due to the")
2811     print("rate change.")

```

Listing 42: Round 5: Implementation.

```
f'''Sufficient'''
```

Listing 43: Round 4: Validation.

```

2797
2798     import pandas as pd
2799     import json
2800
2801     # --- 1. Load All Necessary Data ---
2802     # This part is based on the loading logic from the base code.
2803     payments_df = pd.read_csv('data/payments.csv')
2804     merchant_data_df = pd.read_json('data/merchant_data.json')
2805     fees_df = pd.read_json('data/fees.json')
2806
2807     # --- Implementation of the Calculation ---
2808     # Calculate the total original fee for Rafa_AI's transactions in February 2023
2809     # that are affected by fee ID=17, and then calculate the new total fee for the same
2810     # transactions using the modified rate of 99. The delta is the difference.
2811
2812     # Step 1: Filter payments for 'Rafa_AI' in February 2023.
2813     # February corresponds to days of the year from 32 to 59 (since 2023 is not a leap year).
2814     feb_transactions_df = payments_df[
2815         (payments_df['merchant'] == 'Rafa_AI') &
2816         (payments_df['day_of_year'] >= 32) &
2817         (payments_df['day_of_year'] <= 59)

```

```
2808
2809     ].copy()
2810
2811     # Step 2: Enrich transaction data with merchant-specific information.
2812     # Get the specific data for 'Rafa_AI' from the merchant data file.
2813     rafa_ai_data = merchant_data_df[merchant_data_df['merchant'] == 'Rafa_AI']
2814
2815     # Merge the February transactions with the merchant's data.
2816     full_transaction_data = pd.DataFrame()
2817     if not rafa_ai_data.empty and not feb_transactions_df.empty:
2818         # Use a left merge to add merchant details (like mcc, account_type) to each transaction.
2819         full_transaction_data = feb_transactions_df.merge(rafa_ai_data, on='merchant', how='left')
2820
2821     # Step 3: Retrieve the specific conditions for the fee rule with ID=17.
2822     fee_delta = 0.0
2823     if not full_transaction_data.empty:
2824         fee_rule_17 = fees_df[fees_df['ID'] == 17]
2825
2826         if not fee_rule_17.empty:
2827             # Use .iloc[0] to get the rule as a Series
2828             fee_rule_17 = fee_rule_17.iloc[0]
2829
2830             # Step 4: Filter the transactions that match all conditions of the fee rule.
2831             # The manual states that a null or empty list in a rule field means it applies to all values.
2832             mask = pd.Series(True, index=full_transaction_data.index)
2833
2834             # Match card_scheme (string)
2835             if fee_rule_17['card_scheme']:
2836                 mask &= (full_transaction_data['card_scheme'] == fee_rule_17['card_scheme'])
2837
2838             # Match account_type (list)
2839             if fee_rule_17['account_type']:
2840                 mask &= full_transaction_data['account_type'].isin(fee_rule_17['account_type'])
2841
2842             # Match merchant_category_code (list)
2843             if fee_rule_17['merchant_category_code']:
2844                 mask &=
2845                     full_transaction_data['merchant_category_code'].isin(fee_rule_17['merchant_category_code'])
2846
2847             # Match aci (list)
2848             if fee_rule_17['aci']:
2849                 mask &= full_transaction_data['aci'].isin(fee_rule_17['aci'])
2850
2851             # Match is_credit (boolean, but can be null)
2852             if fee_rule_17['is_credit'] is not None:
2853                 mask &= (full_transaction_data['is_credit'] == fee_rule_17['is_credit'])
2854
2855             # Apply the combined mask to get the affected transactions
2856             affected_transactions_df = full_transaction_data[mask]
2857
2858             # Step 5: Calculate the original and new fees for the affected transactions.
2859             if not affected_transactions_df.empty:
2860                 total_affected_volume = affected_transactions_df['eur_amount'].sum()
2861                 num_affected_transactions = len(affected_transactions_df)
2862
2863                 # Get fee parameters from the rule
2864                 original_rate = fee_rule_17['rate']
2865                 fixed_amount = fee_rule_17['fixed_amount']
2866                 new_rate = 99
2867                 fee_divisor = 10000
2868
2869                 # Calculate total original fee using the formula: fee = fixed_amount + rate * value / 10000
2870                 total_fixed_component = fixed_amount * num_affected_transactions
2871                 total_original_rate_component = (original_rate * total_affected_volume) / fee_divisor
2872                 total_original_fee = total_fixed_component + total_original_rate_component
2873
2874                 # Calculate total new fee with the modified rate
2875                 total_new_rate_component = (new_rate * total_affected_volume) / fee_divisor
2876                 total_new_fee = total_fixed_component + total_new_rate_component
2877
2878                 # The delta is the difference between the new and original total fees
2879                 fee_delta = total_new_fee - total_original_fee
2880
2881             # --- Final Output ---
2882             print(f"(fee_delta:{.14f})")
```

Listing 44: Final solution code.

Listing 45: Generated answer by DS-STAR. Obtained by executing the above final solution code.

Listing 45: **Generated answer by DS-STAR.** Obtained by executing the above final solution code.

2862
2863

H PROMPTS

2864

H.1 ANALYZER AGENT

2865

```
f'''You are an expert data analyst.
Generate a Python code that loads and describes the content of
→ {filename}.

# Requirement
- The file can both unstructured or structured data.
- If there are too many structured data, print out just few examples.
- Print out essential informations. For example, print out all the
→ column names.
- The Python code should print out the content of {filename}.
- The code should be a single-file Python program that is self-contained
→ and can be executed as-is.
- Your response should only contain a single code block.
- Important: You should not include dummy contents since we will debug
→ if error occurs.
- Do not use try: and except: to prevent error. I will debug it
→ later.'''

```

2879

2880
2881

Listing 46: Prompt used to generate Python scripts that describe data files.

2882
2883
2884
2885
2886

DS-STAR begins by creating a description of each data file. Specifically, it utilizes an analyzer agent A_{analyzer} that takes a file name (e.g., ‘payment.csv’) as input to generate a Python script s_{desc} , using the above prompt. The script is then executed to summarize the dataset’s core information. An example of this process, including the generated script and its output, is provided in Appendix D.

2887

H.2 PLANNER AGENT

2888
2889
2890

```
f'''You are an expert data analyst.
In order to answer factoid questions based on the given data, you have
→ to first plan effectively.

# Question
{question}

# Given data: {filenames}
{filenames #1}
{summaries #1}
...
{filenames #N}
{summaries #N}

# Your task
- Suggest your very first step to answer the question above.
- Your first step does not need to be sufficient to answer the question.
- Just propose a very simple initial step, which can act as a good
→ starting point to answer the question.
- Your response should only contain an initial step.'''

```

2905
2906

Listing 47: Prompt used to generate an initial plan.

2907
2908
2909
2910

DS-STAR begins the solution generation process after generating analytic descriptions of N data files. First, a planner agent A_{planner} generates an *initial* high-level executable step p_0 using the above prompt. Here, the query q and obtained data descriptions are used as inputs.

2911
2912
2913
2914
2915

```
f'''You are an expert data analyst.
In order to answer factoid questions based on the given data, you have
→ to first plan effectively.
Your task is to suggest next plan to do to answer the question.

# Question
{question}
```

```

2916
2917 # Given data: {filenames}
2918 {filenames #1}
2919 {summaries #1}
2920 ...
2921 {filenames #N}
2922 {summaries #N}
2923
2924 # Current plans
2925 1. {Step 1}
2926 ...
2927 k. {Step k}
2928
2929 # Obtained results from the current plans:
2930 {result}
2931
2932 # Your task
2933 - Suggest your next step to answer the question above.
2934 - Your next step does not need to be sufficient to answer the question,
2935 - but if it requires only final simple last step you may suggest it.
2936 - Just propose a very simple next step, which can act as a good
2937 - intermediate point to answer the question.
2938 - Of course your response can be a plan which could directly answer the
2939 - question.
2940 - Your response should only contain an next step without any
2941 - explanation.''

```

Listing 48: Prompt used to generate a plan after the initialization step.

After the initial plan is obtained, DS-STAR’s planner agent generates a subsequent step using the above prompt. Unlike the initialization round, planner agent $\mathcal{A}_{\text{analyzer}}$ additionally uses the intermediate step-by-step plans, and the obtained results after execution of its implementation.

H.3 CODER AGENT

```

2945 f'''# Given data: {filenames}
2946 {filenames #1}
2947 {summaries #1}
2948 ...
2949 {filenames #N}
2950 {summaries #N}
2951
2952 # Plan
2953 {plan}
2954
2955 # Your task
2956 - Implement the plan with the given data.
2957 - Your response should be a single markdown Python code (wrapped in
2958 - ``).
2959 - There should be no additional headings or text in your response.''

```

Listing 49: Prompt used to generate a implementation of an initial plan in a Python code.

The initial step for the plan is implemented as a code script s_0 by a coder agent $\mathcal{A}_{\text{coder}}$, using the above prompt. Here, analytic descriptions of data files are additionally used to provide some essential information like column names.

```

2964 f'''You are an expert data analysist.
2965 Your task is to implement the next plan with the given data.
2966
2967 # Given data: {filenames}
2968 {filenames #1}
2969 {summaries #1}
2970 ...
2971 {filenames #N}

```

```

2970 {summaries #N}
2971
2972 # Base code
2973 ````python
2974 {base_code}
2975 ````

2976 # Previous plans
2977 1. {Step 1}
2978 ...
2979 k. {Step k}

2980 # Current plan to implement
2981 {Step k+1}

2982
2983 # Your task
2984 - Implement the current plan with the given data.
2985 - The implementation should be done based on the base code.
2986 - The base code is an implementation of the previous plans.
2987 - Your response should be a single markdown Python code (wrapped in
2988 ````).
2989 - There should be no additional headings or text in your response.''
2990

```

Listing 50: Prompt used to generate a implementation of a plan after the initialization round.

For every round beyond the initial one, DS-STAR’s coder agent $\mathcal{A}_{\text{coder}}$ converts the current plan step p_{k+1} into a Python script using the above prompt. Unlike the initial round, the agent is provided with the intermediate solution code that implements all preceding step p_0, \dots, p_k . This allows the agent to incrementally build upon the existing code, simplifying the implementation of each subsequent step.

H.4 VERIFIER AGENT

```

2999 f'''You are an expert data analyst.
3000 Your task is to check whether the current plan and its code
3001    ↪ implementation is enough to answer the question.
3002 # Plan
3003 1. {Step 1}
3004 ...
3005 k. {Step k}

3006 # Code
3007 ````python
3008 {code}
3009 ````

3010 # Execution result of code
3011 {result}

3012 # Question
3013 {question}

3014
3015 # Your task
3016 - Verify whether the current plan and its code implementation is enough
3017    ↪ to answer the question.
3018 - Your response should be one of 'Yes' or 'No'.
3019 - If it is enough to answer the question, please answer 'Yes'.
3020 - Otherwise, please answer 'No'.'''

```

Listing 51: Prompt used to verify the current plan.

We introduce a verifier agent $\mathcal{A}_{\text{verifier}}$. At any given round, this agent evaluates the state of the solution, *i.e.*, whether the plan is sufficient or insufficient to solve the problem. The evaluation is

3024 based on the cumulative plan, the user’s query, the solution code, which is an implementation of the
 3025 cumulative plan, and its execution result, using the above prompt.
 3026

3027 H.5 ROUTER AGENT

```
3028 f'''You are an expert data analysist.  

  3029 Since current plan is insufficient to answer the question, your task is  

  3030   → to decide how to refine the plan to answer the question.  

  3031  

  3032 # Question  

  3033 {question}  

  3034  

  3035 # Given data: {filenames}  

  3036 {filenames #1}  

  3037 {summaries #1}  

  3038 ...  

  3039 {filenames #N}  

  3040 {summaries #N}  

  3041  

  3042 # Current plans  

  3043 1. {Step 1}  

  3044 ...  

  3045 k. {Step k}  

  3046  

  3047 # Obtained results from the current plans:  

  3048 {result}  

  3049  

  3050 # Your task  

  3051 - If you think one of the steps of current plans is wrong, answer among  

  3052   → the following options: Step 1, Step 2, ..., Step K.  

  3053 - If you think we should perform new NEXT step, answer as 'Add Step'.  

  3054 - Your response should only be Step 1 - Step K or Add Step.'''
  3055
```

3051 Listing 52: Prompt used for the router agent which determines how to refine the plan.
 3052

3053 If the verifier agent $\mathcal{A}_{\text{verifier}}$ determines that the current plan is insufficient to solve the user’s query,
 3054 DS-STAR must decide how to proceed. To this end, DS-STAR employs a router agent $\mathcal{A}_{\text{router}}$ which
 3055 decides whether to append a new step or to correct an existing one, which leverages the above prompt.
 3056

3057 H.6 FINALIZER AGENT

```
3058 f'''You are an expert data analysist.  

  3059 You will answer factoid question by loading and referencing the  

  3060   → files/documents listed below.  

  3061 You also have a reference code.  

  3062 Your task is to make solution code to print out the answer of the  

  3063   → question following the given guideline.  

  3064  

  3065 # Given data: {filenames}  

  3066 {filenames #1}  

  3067 {summaries #1}  

  3068 ...  

  3069 {filenames #N}  

  3070 {summaries #N}  

  3071  

  3072 # Reference code  

  3073 ...python{code}  

  3074 ...  

  3075  

  3076 # Execution result of reference code  

  3077 {result}  

  3078  

  3079 # Question  

  3080 {question}  

  3081  

  3082 # Guidelines
```

```

3078 {guidelines}
3079
3080 # Your task
3081 - Modify the solution code to print out answer to follow the give
3082   ↵ guidelines.
3083 - If the answer can be obtained from the execution result of the
3084   ↵ reference code, just generate a Python code that prints out the
3085   ↵ desired answer.
3086 - The code should be a single-file Python program that is self-contained
3087   ↵ and can be executed as-is.
3088 - Your response should only contain a single code block.
3089 - Do not include dummy contents since we will debug if error occurs.
3090 - Do not use try: and except: to prevent error. I will debug it later.
3091 - All files/documents are in `data/` directory.''
3092

```

Listing 53: Prompt used to generate a final solution, which outputs the answer with the desired format.

To ensure properly formatted output, DS-STAR employs a finalizer agent $\mathcal{A}_{\text{finalizer}}$, which takes formatting guidelines, if exist, and generates the final solution code.

H.7 DEBUGGING AGENT

```

f'''# Error report
{bug}

# Your task
- Remove all unnecessary parts of the above error report.
- We are now running {filename}.py. Do not remove where the error
  ↵ occurred.''

```

Listing 54: Prompt used to summarize the traceback of the error.

When debugging, we first summarize the obtained traceback of error using the above prompt, since the traceback may be too lengthy.

```

f'''# Code with an error:
```python
{code}
```

# Error:
{bug}

# Your task
- Please revise the code to fix the error.
- Provide the improved, self-contained Python script again.
- There should be no additional headings or text in your response.
- Do not include dummy contents since we will debug if error occurs.
- All files/documents are in `data/` directory.''

```

Listing 55: Prompt used for debugging when analyzing data files.

When generating $\{d_i\}_{i=1}^N$ using s_{desc} obtained from $\mathcal{A}_{\text{analyzer}}$, our debugging agent iteratively update the script using only the summarized traceback, using the above prompt.

```

f'''# Given data: {filenames}
{filenames #1}
{summaries #1}
...
{filenames #N}
{summaries #N}

# Code with an error:
```python

```

```

3132 {code}
3133 ``
3134
3135 # Error:
3136 {bug}
3137
3138 # Your task
3139 - Please revise the code to fix the error.
3140 - Provide the improved, self-contained Python script again.
3141 - Note that you only have {filenames} available.
3142 - There should be no additional headings or text in your response.
3143 - Do not include dummy contents since we will debug if error occurs.
3144 - All files/documents are in `data/` directory.''
3145

```

Listing 56: Prompt used for debugging when generating a solution code.

Once DS-STAR obtains  $\{d_i\}_{i=1}^N$ , our debugging agent utilizes such information with the above prompt when generating a solution Python script.

## I LARGE LANGUAGE MODEL USAGE FOR WRITING

In this paper, Gemini was utilized as a general-purpose writing tool. When draft text was provided to Gemini, it improved the writing (*e.g.*, by correcting grammatical errors). The generated text was then manually revised by the authors.

```

3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185

```

3186 **J IN-DEPTH ANALYSIS OF THE VERIFIER AGENT**  
31873188 **J.1 SENSITIVITY ANALYSIS VARYING LLM**  
31893190 **Table 10: Sensitivity analysis varying the underlying LLM used for the verifier agent.**  
3191

3192 <b>Framework</b>	3192 <b>Model for the verifier agent</b>	3192 <b>Easy-level accuracy</b>	3192 <b>Hard-level accuracy</b>
3193 Model-only	3193 -	3193 66.67	3193 12.70
3194 Amity DA Agent	3194 -	3194 80.56	3194 41.01
3195 DS-STAR (Variant)	3195 Llama 3.1 8B	3195 83.33	3195 42.59
3196 DS-STAR	3196 Gemini-2.5-Pro	3196 87.50	3196 45.24

3197 Here, we substitute the base LLM of DS-STAR’s verifier agent from Gemini-2.5-Pro to Llama 3.1  
3198 8B, which is open-sourced and much more lightweight compared to Gemini-2.5-Pro. The rest of the  
3199 configurations are set as the same as in our current manuscript (*i.e.*, other sub-agents, such as the  
3200 planner agent, still use Gemini-2.5-Pro as the base LLM).

3201 As shown in Table 10, we found that using a less capable LLM for the verifier agent leads to a marginal  
3202 performance decrease on the DABStep benchmark. However, it still shows better performance than  
3203 Model-only (which achieved a hard-level accuracy of 12.70% using Gemini-2.5-Pro) and also  
3204 outperforms the best baseline (Amity DA Agent). This indicates that small, open-sourced models like  
3205 Llama 3.1 also possess the capability to effectively verify plan sufficiency. Therefore, DS-STAR’s  
3206 verifier agent is generalizable and robust regarding the LLM choice.

3207 **J.2 SENSITIVITY ANALYSIS VARYING THE SPECIFIC PROMPT TEMPLATE**  
32083210 **Table 11: Sensitivity analysis varying the specific prompt template used to instruct the verifier agent.**  
3211

3212 <b>Framework</b>	3212 <b>Input for the verifier agent’s prompt</b>	3212 <b>Easy-level accuracy</b>	3212 <b>Hard-level accuracy</b>
3213 DS-STAR (Variant)	3213 User’s Query, Plan	3213 83.33	3213 43.92
3214 DS-STAR	3214 User’s Query, Plan, Code, Execution result of the code	3214 87.50	3214 45.24

3215 Here, we investigate the sensitivity of the prompt design for DS-STAR’s verifier agent. Specifically,  
3216 in our original setup (see Listing 51), the verifier agent takes the user’s query, the current plan, the  
3217 current code, and the corresponding execution result as inputs. To analyze prompt sensitivity, we  
3218 removed the code and its execution result and instead guided the verifier agent to check the sufficiency  
3219 of the plan compared to the user’s query. Here, we used Gemini-2.5-Pro for all sub-agents in this  
3220 analysis.

3221 As shown in Table 11, we found that additional information such as the code and its execution results,  
3222 is indeed helpful for the verifier agent, leading to consistently better performance on the DABStep  
3223 benchmark. However, even without these extra inputs, the verifier agent still works quite well,  
3224 showing only a small degradation in performance. This result indicates that the design of the verifier  
3225 agent’s prompt template is not sensitive (*i.e.*, does not significantly impact overall performance).  
3226 Moreover, we would like to emphasize that the prompt is not overfitted to the benchmark we  
3227 used, since information about the benchmark is not needed for the input, and therefore it is well  
3228 generalizable to any kinds of data science tasks.

3229  
3230  
3231  
3232  
3233  
3234  
3235  
3236  
3237  
3238  
3239

3240 **K DS-STAR WITH OPEN-SOURCED LLMs**  
 3241  
 3242  
 3243  
 3244

Table 12: DS-STAR with DeepSeek-V3.

Framework	Model	Easy-level accuracy	Hard-level accuracy
ReAct	DeepSeek-V3	66.67	5.56
Open Data Scientist	DeepSeek-V3	<b>84.72</b>	16.40
<b>DS-STAR</b>	DeepSeek-V3	79.17	<b>28.57</b>

3245  
 3246 Here, we conducted an additional experiment: running DS-STAR with DeepSeek-V3 on the DAB-  
 3247 Step benchmark. As shown in Table 12, our framework is also effective with open-source LLMs,  
 3248 demonstrating a significant outperformance on hard-level tasks over DeepSeek-based frameworks on  
 3249 the validated leaderboard of the DABStep benchmark. This result indicates that our framework is  
 3250 generalizable with respect to the choice of LLM.  
 3251  
 3252  
 3253  
 3254  
 3255  
 3256  
 3257  
 3258  
 3259  
 3260  
 3261  
 3262  
 3263  
 3264  
 3265  
 3266  
 3267  
 3268  
 3269  
 3270  
 3271  
 3272  
 3273  
 3274  
 3275  
 3276  
 3277  
 3278  
 3279  
 3280  
 3281  
 3282  
 3283  
 3284  
 3285  
 3286  
 3287  
 3288  
 3289  
 3290  
 3291  
 3292  
 3293