AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework

Meihao Fan
Renmin University of China
fmh1art@ruc.edu.cn

Ju Fan
Renmin University of China
fanj@ruc.edu.cn

Nan Tang
HKUST (GZ)
nantang@hkust-gz.edu.cn

Lei Cao
University of Arizona/MIT
lcao@csail.mit.edu

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

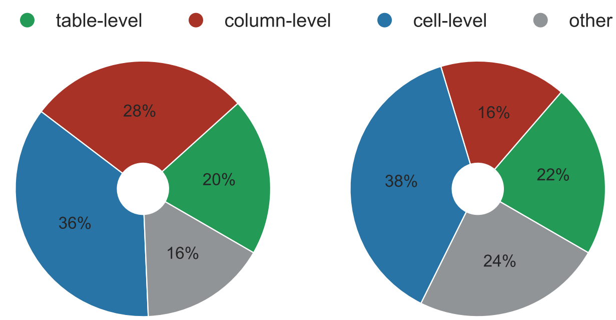
Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Answering natural language (NL) questions about tables, known as Tabular Question Answering (TQA), is crucial because it allows users to quickly and efficiently extract meaningful insights from structured data, effectively bridging the gap between human language and machine-readable formats. Many of these tables are derived from web sources or real-world scenarios, which require meticulous data preparation (or data prep) to ensure accurate responses. However, preparing such tables for NL questions introduces new requirements that extend beyond traditional data preparation. This question-aware data preparation involves specific tasks such as column augmentation and filtering tailored to particular questions, as well as question-aware value normalization or conversion, highlighting the need for a more nuanced approach in this context. Because each of the above tasks is unique, a single model (or agent) may not perform effectively across all scenarios. In this paper, we propose **AutoPrep**, a large language model (LLM)-based multi-agent framework that leverages the strengths of multiple agents, each specialized in a certain type of data prep, ensuring more accurate and contextually relevant responses. Given an NL question over a table, AutoPrep performs data prep through three key components. **Planner**: Determines a logical plan, outlining a sequence of high-level operations. **Programmer**: Translates this logical plan into a physical plan by generating the corresponding low-level code. **Executor**: Executes the generated code to process the table. To support this multi-agent framework, we design a novel Chain-of-Clauses reasoning mechanism for high-level operation suggestion, and a tool-augmented method for low-level code generation. Extensive experiments on real-world TQA datasets demonstrate that AutoPrep can significantly improve the state-of-the-art TQA solutions through question-aware data preparation.

PVLDB Reference Format:
Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(a) WikiTQ (b) TabFact

Figure 1: An error analysis of LLM-based TQA (using GPT-4) on two well-adopted datasets, with statistics calculated from a sample of 500 instances from each dataset.

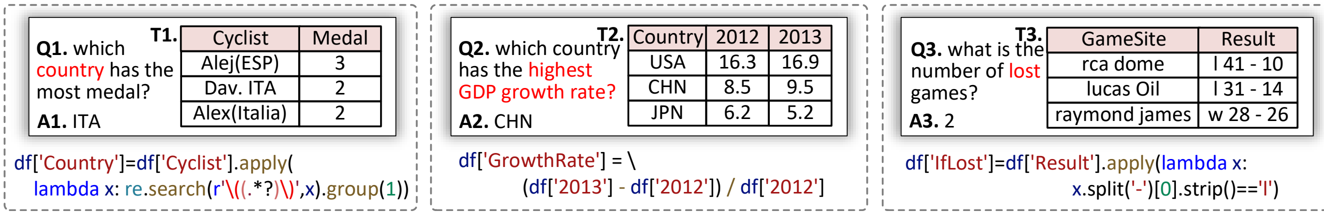
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fmh1art/AutoPrep.

1 INTRODUCTION

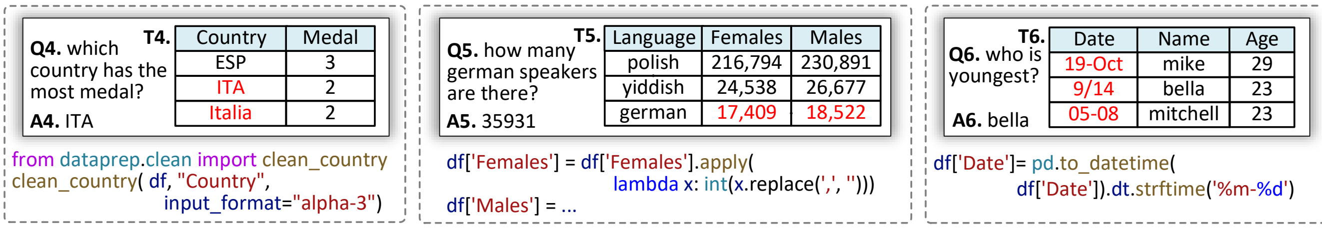
Tabular Question Answering (TQA) refers to the task of answering natural language (NL) questions based on provided tables [14, 15, 25, 29]. TQA empowers non-technical users such as domain scientists to easily analyze tabular data and has a wide range of applications, including table-based fact verification [6, 13] and table-based question answering [27, 30]. As TQA requires NL understanding and reasoning over tables, state-of-the-art solutions [15, 38, 39, 43–45] rely on large language models (LLMs), either utilizing LLMs as black-boxes or leverageing them for code generation.

Question-Aware Data Preparation. As many tables in TQA originate from web sources or real-world data, they demand meticulous *data preparation* (or data prep) to produce accurate answers. Figure 1 shows an error analysis of an LLM-based approach (using GPT-4) across two TQA tasks: table-based question answering on the WikiTQ dataset [30] and table-based fact verification on the TabFact dataset [13]. The results show that 84% and 76% of the errors arise from insufficient data prep at various granularities (table-level, column-level, and cell-level) in relation to the NL questions. This highlights the critical role of thorough data prep in ensuring accurate responses for TQA.

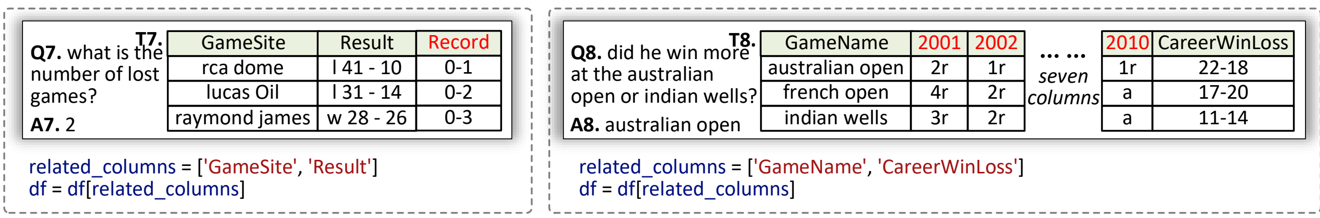
However, although data prep has been extensively studied in the database community [9], preparing tables to answer NL questions requires a new requirement of **question-aware data preparation**,



(a) Question-Aware Column Augmentation: Necessary information for QA is not explicitly available in tables.



(b) Question-Aware Value Normalization: Heterogeneous data representations or formats in tables, and how to normalize is query-dependent.



(c) Query-Aware Column Filtering: Irrelevant columns increase complexity and cost of LLMs, which also lead to errors in TQA methods.

Figure 2: Examples of Question-Aware Data Preparation.

which goes beyond traditional data prep. Unlike conventional methods, question-aware data prep must *tailor tables to the specific needs of the question, e.g.,* performing tasks such as question-specific column augmentation and filtering, as well as question-aware value normalization or conversion. To illustrate these scenarios more concretely, we provide the following examples.

Example 1. *Figure 2 shows several examples of question-aware data prep tasks that are needed for TQA.*

(a) **Column-Level:** Question-Aware Column Augmentation. *Figure 2a illustrates the necessity of augmenting the tables with new columns, as the information that NL questions require is not explicitly available. For example, to answer Q_1, a new* `Country` *column should be created by extracting country information from* `Cyclist`.

(b) **Cell-Level**: Question-Aware Value Normalization: *Figure 2b highlights the need for normalizing data types or formats in tables (*e.g., *"ITA" vs. "Italia" in T_4, and "19-Oct" vs. "9/14" in T_6). Note that normalizing tables in TQA requires converting data types or formats based on the need of questions, meaning different questions may result in various operations. For example, as Q_5 asks for summing numbers in T_5 but the corresponding columns are of string type, we need to remove commas from numbers and convert strings to integers.*

(c) **Table-Level**: Question-Aware Column Filtering. *As shown in Figure 2c, selecting* question-relevant *columns from tables is important, as irrelevant columns add complexity and can lead to errors in code generation. For example, the* `Result` *and* `Record` *columns in table T_7 have similar formats and ambiguous meanings, which may mislead the LLM. In addition, although T_8 contains 12 columns, only two are relevant to Q_8. Providing all columns to the LLM can thus result in long-context understanding challenges [26].*

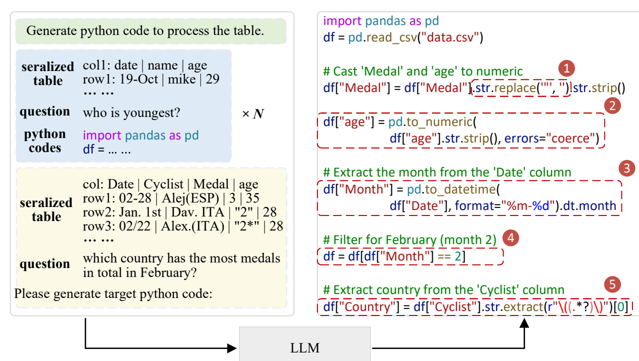


Figure 3: An LLM-based method with few-shot prompting for question-aware data prep.

The above example illustrates that different types of data prep operations are required to meet the needs of various NL questions. Therefore, traditional data prep, which prepares tables without considering the specific requirements of NL questions, may not effectively address these question-specific challenges.

Key Challenges. A straightforward approach to question-aware data prep is to prompt an LLM to prepare tables, leveraging its ability to interpret the specific requirements of NL questions. Specifically, given the input token constraints of LLMs, it is often infeasible to supply an entire table and request a fully prepared table as output. Therefore, a more effective strategy employs few-shot prompting to generate data prep programs. For instance, consider the table in Figure 3 with columns `Date`, `Cyclist`, `Medal` and `Age`, and question: "*Which country has the most medals in total in February*". The few-shot prompting strategy prompts an LLM with a task description and a few demonstrations, and requests the LLM to generate Python programs as shown in Figure 3.

However, this LLM-based solution may encounter the following challenges when performing question-aware data prep for TQA.

(C1) Question-Specific Data Prep Operations. Given the difficulties of understanding both NL questions and tables, it is challenging to determine which data prep operations are specifically required to meet the needs of each NL question, thus often resulting in false negatives (FNs) and false positives (FPs). Take the results in Figure 3 as an example: converting `Age` to a numerical format in code block ② is an FP, as it is irrelevant to the question. Conversely, the method fails to normalize the `Date` column before extracting the month in code block ③, which constitutes an FN, because it does not recognize the inconsistent `Date` formats.

(C2) Customized Data Prep Implementations. Due to input token limitations and challenges in long-context understanding [26], it is not easy to fully understand all possible issues in a table, and thus may struggle to generate customized programs to correct the issues. For example, in Figure 3, the normalization of `Medal` in code block ① overlooks certain corner cases (*e.g.,* "2*"), and the country extraction in code block ⑤ fails to handle "Dav.ITA", which is formatted differently from other values.

Recent methods, such as CoTable [38] and ReAcTable [45], can improve few-shot prompting by employing techniques like Chain-of-Thoughts (CoT) and ReAct. However, these methods remain insufficient to tackle the challenges, as they combine all diverse tasks, such as determining operations and implementing them at various granularities (table-level, column-level, and cell-level), within a single LLM agent. Existing studies [23] have shown that a single LLM agent is often ineffective when tasked with handling a diverse range of operations, due to limited context length in LLMs and decreased inference performance with more input tokens.

AutoPrep: A Multi-Agent LLM Framework. To address the challenges, we propose **AutoPrep**, an LLM-based multi-agent framework that decomposes the question-aware data prep task into smaller sub-tasks, and then leverages multiple agents specialized in different sub-tasks to ensure accurate and contextually relevant responses. Specifically, inspired by the design principles of modern DBMS, AutoPrep decomposes the data prep task for a given NL question over a table into three stages.

The Planning Stage: This stage analyzes the NL question and the structure of the given table, and generates a logical plan that outlines the sequence of high-level data prep operations. These operations may include tasks such as column augmentation, value normalization, or filtering irrelevant data, as shown in Figure 2.

The Programming Stage: This stage translates the logical plan into a physical plan by generating low-level executable code, which is responsible for selecting the appropriate programming constructs (e.g., Python functions or APIs) for each operation, customizing the code to match the table's structure and data formats, etc.

The Executing Stage: This stage executes the generated code for each operation and returning any errors encountered to the Programming stage for debugging.

Our AutoPrep framework features decoupling and extensibility, enabling it to overcome the limitations of single-agent methods. (1) *Decoupling*: AutoPrep separates the complex question-aware data prep task into distinct stages, allowing each specialized agent to focus on a single sub-task. This decoupling improves accuracy and maintainability of each sub-task. (2) *Extensible*: AutoPrep is designed to be extensible, *e.g.,* developing specialized implementations w.r.t. each type of data prep operations, or adding new operation types. This flexibility makes it adaptable to a wide range of data prep scenarios and evolving requirements.

Technical Solutions in AutoPrep. We design technical solutions in AutoPrep to effectively address the aforementioned challenges.

To address the first challenge (C1), we design a Planner agent in the Planning Stage, which suggests a tailored sequence of high-level operations that meet the specific needs of these questions. The core idea behind the Planner agent is a novel *Chain-of-Clauses (CoC) reasoning* method. This method translates the natural language question into an *Analysis Sketch*, outlining how the table should be transformed to produce the answer and guiding the agent's reasoning based on this sketch. Compared with existing CoT methods [39], which decompose questions into sub-questions, our approach more effectively captures the relationships between questions and tables.

To tackle the second challenge (C2), AutoPrep designs a specialized Programmer agent for each type of data prep operations. Existing LLM-based code synthesis often produces overly generic code that fails to effectively address the *heterogeneity* challenges of tables. For example, values may have diverse syntactic formats (*e.g.,* "19-Oct" and "9/14" in T_6) or or semantic representations (*e.g.,* "ITA" and "Italia" in T_4), making it difficult to generate code tailored to such variations. To address this, we develop a *tool-augmented* approach that improves the LLM's code generation capabilities by utilizing predefined API functions. This approach effectively resolves the issue of overly generic code by allowing the LLM to leverage specialized functions tailored to variations in table values.

Contributions. Our contributions are summarized as follows.

(1) We introduce a novel problem of question-aware data preparation for TQA, which is formally defined in Section 2..

(2) We propose AutoPrep, an LLM-based multi-agent framework for question-aware data prep (Section 3). We develop effective techniques in AutoPrep for the Planner agent (Section 4) and the Programmer agents (Section 5).

(3) We conduct a thorough evaluation on data prep in TQA (Section 6). Extensive experiments show that AutoPrep achieves new SOTA accuracy, outperforming TQA methods without data prep by 12.22 points on WikiTQ and 13.23 points on TabFact, and surpassing TQA methods with data prep by 3.05 points on WikiTQ and 1.96 points on TabFact.

2 QUESTION-AWARE DATA PREP FOR TQA

2.1 Tabular Question Answering

Let Q be a natural language (NL) question, and T a table consisting of m columns (*i.e.,* attributes) $A_1, A_2, \ldots, A_m$ and n rows $r_1, r_2, \ldots, r_n$, where v_{ij} denotes the value in the i-th row and j-th column of the table. The problem of **tabular question answering (TQA)** is to generate an answer, denoted as A, in response to question Q based on the information in table T. By the purposes of the questions, there are two main types of TQA problems: (1) table-based fact verification [6, 13], which determines whether Q can be *entailed* or *refuted* by T, and (2) table-based question answering [27, 30], which extracts or reasons the answer to Q from T.

EXAMPLE 2. *Figure 2 provides several examples of TQA. For instance, consider table T_1, which contains medal information for cyclists from different countries, with two columns:* `Cyclist` *and* `Medal`. *Given the question Q_1, "Which* ***country*** *has the most medals?", the answer A_1 should be ITA, as two Italian cyclists, "Dav" and "Alex", have won a total of 4 medals, more than the ESP cyclist "Alej".*

TQA often requires complex reasoning over tables. For instance, to answer question Q_2, we first need to calculate the "GDP growth rate" for all countries, then sort the countries by growth rate, and finally identify the country with the highest GDP growth rate, i.e., *CHN.*

2.2 Data Prep for TQA

Unlike traditional data prep, for **question-aware data prep for TQA**, we need to *tailor table T to specific needs of question Q*.

Data Prep Operations. To meet the new requirements of data prep for TQA, this paper defines high-level **data prep operations** (or *operations* for short) to formalize the *question-aware* data prep tasks, such as column augmentation, question-aware value normalization or conversion, and column filtering. Formally, an operation, denoted as o, encapsulates a specific question-aware data prep task that transforms table T into another table T', *i.e.,* $T' = o(T)$. Specifically, this paper considers the following three types of operations.

- `Augment`: a data prep task that augments a new column for table T based on existing columns, to better answer Q. This task typically involves column combinations via arithmetic operations, value extraction and inference, etc.
- `Normalize`: a data prep task that normalize types or formats of the values in a column of T based on the needs of Q. This task typically involves value representation or format normalization, type conversion, etc.
- `Filter`: A data prep task that filters out columns in T that are not relevant to answering question Q. This is crucial for handling large tables to address the input token limitations and challenges in long-context understanding of LLMs [26].

Given a high-level operation o_i, we define f_i as its low-level *implementation*, either by calling a well-established algorithm from a known Python library or using a customized Python program to meet the requirements of o_i.

EXAMPLE 3. *Figure 2 shows examples of question-aware data prep operations for TQA, along with their implementations in Python.*

(a) The `Augment` operation: *Figure 2a shows three examples of* `Augment` *operations,* i.e., *extracting* `Country` *information from column* `Cyclist` *in T_1, computing new column* `GrowthRate` *using two columns in T_2, and inferring a status of* `IfLost` *by analyzing the scores in column* `Result`. *Note that the corresponding Python implementations are provided alongside the operations.*

(b) The `Normalization` operation: *Figure 2b shows three examples of* `Normalize` *operations,* i.e., *normalizing country formats in T_4, converting strings into integers in T_5, and normalizing value formats in* `Date` *in T_6. We can see that, to implement a specific operation (*e.g., *in T_4), we sometimes need to call external tools, like a function* `clean_country` *from an external library.*

(c) The `Filter` operation: *Figure 2c illustrates two examples of* `Filter` *operations in tables T_5 and T_6. The* `Result` *and* `Record` *columns in table T_7 have similar formats and ambiguous meanings, potentially misleading the LLM. Additionally, although T_8 has 12 columns, only two are relevant to Q_8, and providing all columns may cause long-context understanding challenges of LLMs [26].*

The above three types of operations address most of the errors caused by insufficient data prep, as shown in our error analysis in Figure 1. In fact, the set of data prep operations is extensible, in order to support new or domain-specific data prep tasks. Note that we do not consider operations like groupings or selecting rows in data prep, because these operations are more related to question answering. Instead, we design an ANALYZER agent to handle these operations, which will be introduced later.

Question-Aware Data Prep for TQA. Given an NL question Q posed over table T, *question-aware data prep for TQA* is to generate a sequence of high-level operations $O = \{o_1, o_2, \ldots, o_{|O|}\}$ as a *logical plan*. Then, it generates a *physical plan*, where each operation o_i is implemented by low-level code f_i, such that these operations transform T into a new table T' that meets the needs of Q.

EXAMPLE 4. *To better answer the example question Q_1 over T_1 in Figure 2a, we first generate an* `Augment` *operation to transform T_1 into T_1' with three columns:* `Cyclist`, `Medal`, *and* `Country`. *Next, we generate a* `Normalize` *operation, similar to the one in T_4, to standardize the values in* `Country` *into a unified format, producing table T_1''. Finally, we leverage LLMs, either as black-box or for code generation, to compute the answer A_1 to Q_1 from the prepared table T_1''.*

2.3 Related Work

Tabular Question Answering. Most of the SOTA solutions for TQA rely on LLMs [8, 19], as TQA requires NL understanding and reasoning over tables. Specifically, direct Prompting approaches [12, 39], which prompt an LLM to directly generate answers through next token prediction, may struggle when processing large tables due to the input token limitation and long-context understanding challenges [26]. Code Generation approaches [33] leverage LLMs to generate programming code (*e.g.,* Python) that extracts answers from tables, providing better explainability and adaptability.

Previous TQA methods, like Dater [44], Binder [15], CoTable [38] and ReAcTable [45], also consider data prep in their question answering process. Specifically, Dater [44] prompts LLMs to select relevant columns related to answering the question, targeted at solving filtering tasks. Binder [15] proposes to integrate SQL with an LLM-based API to incorporate external knowledge, which may partly address augmentation tasks. Similarly, CoTable [38] addresses the filtering tasks and augmentation tasks by designing operators which are implemented by LLM completion. ReAcTable [45] uses few-shot demonstrations to instruct LLMs to generate python code or SQL to address the augmentation and filtering tasks.

Table 1: Comparing AutoPrep with existing TQA methods.

Method	Multiple Agents	Supported Data Prep Tasks		
		Filter	Augment	Normalize
Dater [44]	✗	✓	✗	✗
Binder [15]	✗	✗	✓	✗
CoTable [38]	✗	✓	✓	✗
ReAcTable [45]	✗	✓	✓	✗
AutoPrep	✓	✓	✓	✓

However, previous TQA methods have the following limitations when handling data prep, as summarized in Table 1. First, all these methods do not address the cell-level *normalization* tasks, which account for the most significant error types, as indicated in Figure 1. Second, all these methods utilize a single LLM agent for both data prep and answer reasoning. Given that data prep is a complex challenge, these one-size-fits-all solutions still fail to achieve satisfactory performance as demonstrated in our experiments.

In this paper, we split the TQA task into two phases, data prep and answer generation. Based on this, we propose a multi-agent framework AutoPrep and design specific agents for all the common error types, namely filtering, augmentation and normalization.

Traditional Data Preparation. Data prep techniques have been widely adopted in various tasks. Auto-Weka [36] leverages Bayesian optimization to identify data prep operations. Auto-Sklearn [18, 22] and TensorOBOE [42] apply meta-learning to discover promising operations. Alpine Meadow [34] introduces an exploration-exploitation strategy, while TPOT [28] uses a tree-based representation of data prep and genetic programming optimization techniques. Several studies [7, 17, 20] explore reinforcement learning techniques. HAIPipe [11] integrates both human-orchestrated and automatically generated data prep operations.

In this paper, we propose AutoPrep to generate question-specific data prep operations for TQA tasks. The key difference between these methods and AutoPrep is that the data prep operations generated by AutoPrep are tailored to specific user questions and customized for the tables, as different questions may have different data prep requirements, even for the same table.

LLMs-based Multi-Agent Framework. A multi-agent LLM framework refers to a well-designed hierarchical structure consisting of multiple LLM-based agents and scheduling algorithms [37]. Compared with single-agent methods based on prompting techniques, such structures are better suited for handling complex tasks like software development, issue resolution, and code generation [10, 21, 23, 24, 32, 35]. AutoTQA [46] propose a multi-agent framework supporting Tabular Question Answering.

While AutoTQA improves table analysis, this paper focuses on the performance bottleneck caused by data prep issues in TQA. Our proposed framework, AutoPrep, addresses question-aware data prep for TQA and can be integrated as a plugin into current TQA approaches to further improve the overall performance.

3 AN OVERVIEW OF AUTOPREP

To address the limitations, we propose AutoPrep, a *multi-agent* LLM framework that automatically prepares tables for given NL questions. Figure 4 provides an overview of our framework. Given an NL question Q posed over table T, AutoPrep decomposes the data prep process into three stages:

(1) Planner Agent: the **Planning** stage. It guides the LLM to suggest *high-level* data prep operations $O = \{o_1, o_2, \ldots, o_{|O|}\}$, which are tailored to specific question Q,

(2) Multiple Programmer Agents (*i.e.,* Augmenter): the **Programming** stage. It directs the LLM to generate *low-level* implementation f_i (*e.g.,* Python code) for each operation o_i customized for the table T. Besides, it is also tasked for code debugging if any execution errors occur.

(3) An Executor Agent: the **Executing** stage. It executes the given low-level code and reports error messages if any code bugs occur.

After that, an Analyzer agent extracts the answer from the prepared table. This agent can either use LLMs as black-boxes or leverage them for code generation, which is *orthogonal* to the question-aware data prep problem studied in this paper. For simplicity, we use a Text-to-SQL strategy that translates the question into an SQL query over the prepared table to obtain the final answer, as shown in Figure 4. Note that other strategies could also be used by the agent in a "plug-and-play" manner, which will be discussed in the experiments.

Example 5. *Figure 4 illustrates how AutoPrep supports data prep for question "Which country has the most medals in total in February?" posed over a table with columns* `Date`, `Cyclist`, `Medal` *and* `Age`.

(a) The `Planning` stage: *The Planner suggests the following high-level operations to address the specific NL question:*

- T'`[Country]=Augment(`*"Extract country code"*, `Cyclist)` *that extracts the country information from column* `Cyclist`, *producing a new* `Country` *column, in response to the "which country" part of the question.*
- `Normalize(`*"Case to INT"*, `Medal)` *that standardizes the value formats in the* `Medal` *column (*e.g., *removing quotation marks and asterisks) and then converts the strings to integers, as the question requires aggregating the "medals in total";*
- `Normalize(`*"Format date as %m-%d"*, `Date)` *standardizes the values in the* `Date` *column into a unified format to support the 'in February' condition in the question.*
- T' `=Filter(`T`, [Date, Country, Medal])` *that filters out column* `Age`, *which is irrelevant to the question;*

(b) The `Programming` stage: *AutoPrep designs specialized Programmer agents for each operation type,* i.e.,, *Augment, Normalizer and Filter, corresponding to the* `Augment`, `Normalize` *and* `Filter` *operations, respectively. Each specialized Programmer focuses on generating executable code for its assigned operations.*

(c) The `Executing` stage: *an Executor agent iteratively refines the generated code if any error occurs.*

After these stages, AutoPrep generates a prepared table T^*, *which is then fed into an Analyzer agent to produce the answer* `ITA`.

The Planner Agent. The key challenge here is how to suggest *question-aware operations* that address specific NL questions. Specifically, even for the same table, different NL questions may require not only different data prep operations but also varying sequences

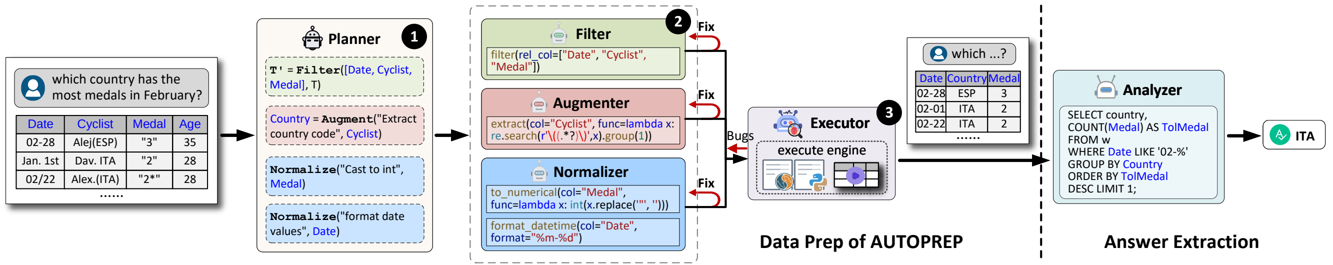


Figure 4: A Multi-agent Data Prep Framework for TQA (Left), which could be plugged to any other TQA methods (Right).

of those operations. For example, the second `Normalize` operation might not be necessary if "in February" is not part of the question. To address this challenge, we propose a novel *Chain-of-Clauses (CoC) reasoning* method for the Planner agent. This method translates the natural language question into an *Analysis Sketch*, outlining how the table should be transformed to produce the answer, thereby guiding the agent's reasoning based on this sketch. Compared to existing CoT methods [39], which break down questions into sub-questions, our approach better captures the relationships between questions and tables. More details of the CoC method are given in Section 4.

The Programmer Agents. The key challenge is that a given high-level operation can have multiple executable code alternatives (e.g., Python functions), and the difference in outcomes between the best and worst options can be substantial. For example, the Augmenter agent may generate an overly generic regular expression that extracts countries based on parentheses. Unfortunately, this code fails to correctly process "Dav. ITA", which is formatted differently from other values. To tackle this challenge, we develop a *tool-augmented* approach that enhances the LLM's code generation capabilities by utilizing predefined API functions. We establish a search space of API functions for all high-level operations, create an algorithm to identify the most suitable function for a given operation, and optimize it using a trial-and-error mechanism. More details of our tool-augmented approach are discussed in Section 5.

Note that our proposed AutoPrep framework is extensible. When additional question-aware data prep operations are required, more specialized Programmer agents can be designed to handle them. The central Planner agent can then determine which operations should be performed and assign them accordingly.

4 THE PLANNER

Given a question Q posed over a table T, Planner aims to generate a *logical plan* that outlines a sequence of high-level operations needed to prepare the table for answering the question.

4.1 A Direct Prompting Method

The most common way to generate a logical plan is to directly prompt an LLM using a typical in-context learning approach. The inputs are a question Q, a table T, a set Σ of specifications for each operation type, and an LLM θ. Here, each specification $\sigma \in \Sigma$ describes the purpose of an operation type, e.g., "*an* `Augment` *operation augments a new column for a table based on existing columns, in response to the specific needs of a question*". The output of the algorithm is a set O of high-level operations like those shown in Figure 4, *e.g.,* `Normalize("Cast to INT", Medal)`.

Example 6. *Figure 5(a) shows an example of the aforementioned direct prompting method, which produces two high-level operations,* `Filter` *and* `Normalize`*. However, this logical plan might not be accurate as discussed as follows.*

(a) Incorrect operations: *The* `Filter` *operation incorrectly filters out the* `Cyclist` *column because the column is not explicitly mentioned in the question. This operation is clearly disastrous, as the country information is implicitly embedded in the* `Cyclist` *column.*

(b) Missing operations: *Observing the ground-truth in Figure 4, we see that the* `Augment` *operation on* `Cyclist` *is not generated, as the column has already been filtered out. Additionally, although the* `Normalize` *operation on* `Medal` *is generated, the* `Normalize` *operation on* `Date` *is missing. This is because the phrase "the most" in the question suggests the need for type conversion of* `Medal`*, but there are no clues from the question to normalize* `Date`*, unless we observe from the table that the values are inconsistent.*

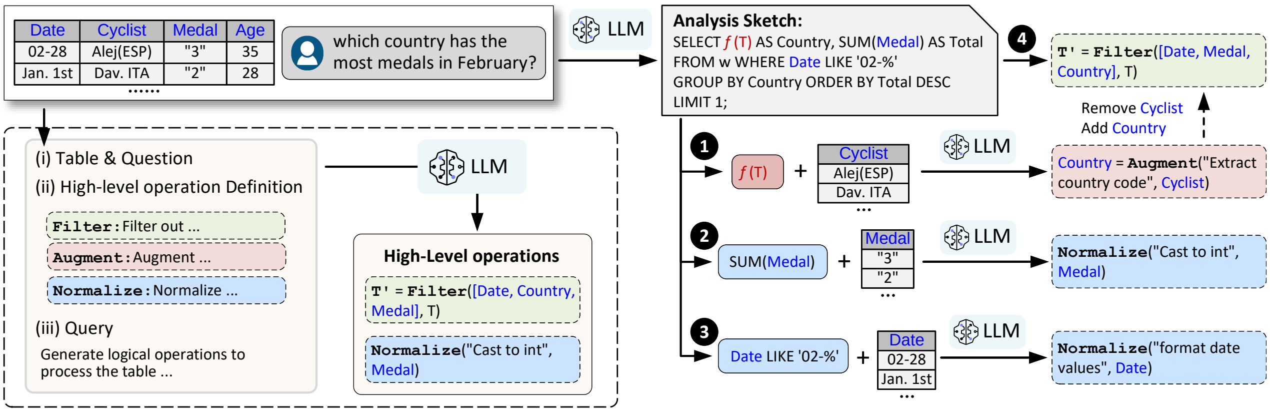
Key Challenge. The above example clearly demonstrates that the key challenge in designing the Planner agent, as discussed earlier, is to capture *the relationships between different parts of question Q and the columns in table T*. For instance, as shown in Figure 4, the `Normalize` operation on the `Date` column is generated by considering two key factors: (1) the question contains "in February", and (2) the values in the `Date` column are inconsistent.

4.2 A Chain-of-Clauses Method

To address this challenge, we propose a *Chain-of-Clauses (CoC) reasoning* method that decomposes the entire process of logical plan generation into two phases, as shown in Figure 5(b).

- The first phase leverages an LLM to generate an SQL-like *Analysis Sketch*, outlining how the table should be transformed to produce the answer.
- The second phase iteratively examines different clauses in the Analysis Sketch, *e.g.,* `Date LIKE '02-%'`. For each clause, we associate the corresponding data in T (*e.g.,* values in column `Date`) with it to prompt the LLM for generating possible high-level operations (*e.g.,* `Normalize`).

Compared with existing CoT methods [39], which simply break down questions into sub-questions, our approach is more effective for logical plan generation. First, our method decomposes the question into a set of analysis steps over table, formalized as an SQL-like Analysis Sketch, which simplifies the task of logical plan



(a) Direct Prompting **(b) Chain-of-Clauses Reasoning**

Figure 5: Our proposed approaches to logical plan generation in the PLANNER agent: (a) A straightforward direct prompting method, and (b) a more effective Chain-of-Clauses (CoC) reasoning method.

generation. More importantly, our method *jointly* considers each analysis step along with the corresponding relevant columns (instead of the whole table) to prompt the LLM, effectively capturing the relationships between the question and the data.

SQL-like Analysis Sketch. An Analysis Sketch S is an SQL-like statement, which can be represented as follows.

$$\texttt{SELECT}\ A\ |\ \mathrm{agg}(A)\ |\ f(\{A_i\})\ \texttt{FROM}\ T$$
$$\texttt{WHERE}\ \mathrm{Pred}(A_i)\ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathrm{Pred}(A_j)$$
$$\texttt{GROUP BY}\ A\ \texttt{ORDER BY}\ A\ \texttt{LIMIT}\ n$$

where agg(A) is an aggregation function (*e.g.,*SUM and AVG) over column A, Pred(A_i) denotes a predicate (*i.e.,* filtering condition) over column A_i, such as `Date LIKE '02-%`. Note that $f(\{A_i\})$ is a *user-defined function* (UDF) that maps existing columns $\{A_i\}$ in the table into a new column. Figure 5(b) shows an example Analysis Sketch with a UDF function that specifies a new column `Country`, *i.e.,* f(`Cyclist`) `AS Country`. Obviously, this UDF is introduced because the Analysis Sketch contains a `GROUP BY Country` clause.

EXAMPLE 7. *Figure 5(b) shows our proposed CoC reasoning method for the example question and table in Figure 4. Specifically, the method generate a set O of operations via the following two phases.*

(a) Phase I - Analysis Sketch Generation: *In this phase, the algorithm prompts the LLM θ with several exemplars $\{(Q_i, T_i, s_i)\}$ to generate an Analysis Sketch S, as shown in Figure 5(b).*

(b) Phase II - Operation Generation: *In this phase, the algorithm iteratively examines the clauses in Analysis Sketch S as follows.*

- f(`Cyclist`) `AS Country`*: this clause and relevant columns are used to prompt the LLM to generate an* `Augment` *operation.*
- `SUM(Medal)`*: this clause and the values in column* `Medal` *are used to prompt the LLM to generate a* `Normalize` *operation.*
- `Date LIKE '02-%'`*: this clause and the values in column* `Date` *are used to prompt the LLM to generate a* `Normalize`.

Finally, the algorithm generates a `Filter` *operation that only selects the columns relevant to the Analysis Sketch.*

5 THE PROGRAMMER & EXECUTOR

PROGRAMMER agents translate a high-level logical plan into a *physical plan* by generating low-level code, which is then passed to an EXECUTOR agent for code execution and interactive debugging. A straightforward approach is to prompt an LLM with the logical plan using in-context learning with several exemplars, asking the LLM to generate code for each high-level operation in the plan. The code is then executed iteratively, and if any runtime errors occur, the PROGRAMMER is prompted with the error messages for debugging.

However, the above method may have limitations, as the generated code may be overly generic and unable to effectively address the *heterogeneity* challenge of the tables. Specifically, many tables originate from web sources or real-world scenarios, where values have diverse syntactic (*e.g.,* "19-Oct" and "9/14" in T_6) or semantic formats (*e.g.,* "ITA" and "Italia" in T_4), making it difficult to generate code customized to these tables. For example, consider table T_4 in Figure 2b. Given a `Normalize` operation to standardize country formats, a generic function from an existing Python library may not be sufficient to transform country names (*e.g.,* "Italia") to their ISO codes (*e.g.,* "ITA"). In such cases, customized functions, such as the `clean_country` function from an external library `dataprep` [31], are needed to address specific requirements of code generation.

To address these limitations, we introduce a *tool-augmented* method for the PROGRAMMER agents, enhancing the LLM's code generation capabilities by utilizing pre-defined API functions, referred to as the *function pool* in this paper. Specifically, we first introduce the key idea of our tool-augmented method in 5.1, then present the function pool in Section 5.2.

5.1 Tool-Augmented Method: Key Idea

Figure 6 provides an overview of our *tool-augmented* method for physical plan generation and execution. Given a table T and a set $O = \{o_1, \ldots, o_{|O|}\}$ of high-level operations, the method *iteratively* generates and executes low-level code for each individual operation $o_i \in O$, thus generating a sequence of intermediate tables T_1 (*i.e.,* the input T), $T_2, \ldots T_{|O|+1}$.

Specifically, in the i-th iteration, given a high-level operation o_i (*e.g.,* `Augment`) and an intermediate table T_i, the method generates

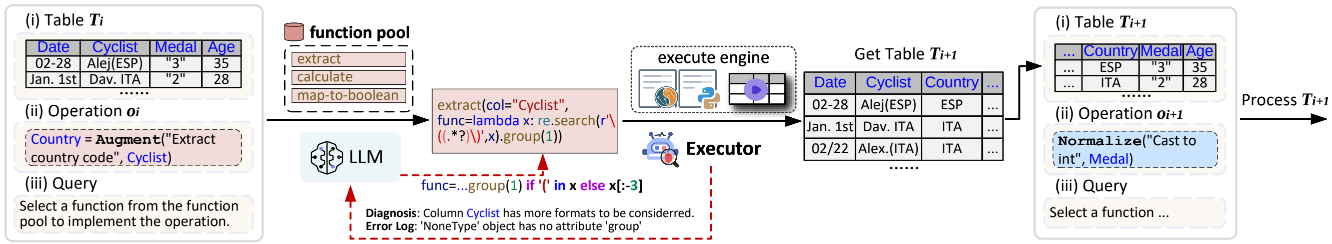


Figure 6: Our proposed tool-augmented method for physical plan generation and execution.

low-level code and executes it to produce an updated table T_{i+1} through the following two steps.

Function Selection and Argument Inference. The method first prompts the LLM to select a specific function from a function pool $\mathcal{F}$ corresponding to the operation type (e.g., F_{Aug}) and infer the arguments (*e.g.,* regular expression) for the selected function.

For instance, given the `Augment` operation over table T_i shown in Figure 6, the LLM selects a function from the pool $\mathcal{F}_{\text{Aug}}$ designed specifically for `Augment`, obtaining an `extract` function with two arguments: `column` and `func`. The LLM then generates preliminary code for these arguments, assigning `Cyclist` to the `column` argument and generating a lambda function with a specific regular expression for argument `func`.

Note that, in addition to selecting functions from the corresponding pool, our method may also prompt the LLM to write specific code for an operation if no existing function is suitable to meet the operation's requirements.

Code Execution and Debugging. Given the function generated in the previous step, the Executor agent then applies the function over table T_i. If any bugs occur during execution, it captures and summarizes error messages and returns to corresponding Programmer agent. For instance, as shown in Figure 6, the first lambda function for the argument `func` only extracts countries based on parentheses, which may produce incorrect results for the value "Dav.ITA", as it is formatted differently from other values. In this case, based on the execution results, the Executor agent records the error log and examines relevant data to summarize reasons, which will be passed to the corresponding Programmer to modify the code. After the execution and debugging process, the method produces a new intermediate table T_{i+1} and proceeds to the next operation, o_{i+1}, *i.e.,* a `Normalize` operation in Figure 6.

5.2 Search Space of Function Pools

This section presents a search space of function pools for different types of high-level operations formalized in the current AutoPrep.

Function pool for `Augment`. We define a specific Programmer agent, *i.e.,* `Augmenter`, to generate low-level code for the operation type `Augment` from the following function pool $\mathcal{F}_{\text{Aug}}$.

- `extract`. This function extracts a substring from a column to generate a new column. It has two arguments: `column`, representing the name of the source column, and `func`, representing the substring extraction function. Using table T_1 from Figure 2a as an example, to extract the country code, we would set `column` to `Cyclist` and `func` to a lambda function `lambda x: re.search(r'(.*?)',x).group(1)`.
- `calculate`. This function generates a new column through arithmetic operations of columns such as addition, subtraction, multiplication, or a combination of these. It has two arguments: `columns`, representing a list of source columns and `func`, representing a lambda function with input to be a dictionary of values, since we may perform calculations on multiple columns. For example, given a table with two columns `A` and `B`, recording the scores of team A and team B, we need to generate a new column `ScoreDifference`. In this case, we set `columns` to `['A', 'B']` and `func` to the specific lambda function `lambda x: x['A']-x['B']`.
- `map-to-boolean` and `concatenate`. Both functions generate a new column from multiple columns. The difference is that one generates boolean values, while the other concatenates strings. Both operators take two arguments, `columns` and `func`, similar to those in the `calculate` operator.
- `infer`. This function directly uses LLMs to deduce values that could not be processed by the functions above. It takes a list of source columns and the failed target values as input, learning from demonstrations of successfully processed values to directly output the target values.

Function pool for `Normalize`. We define a specific Programmer agent, *i.e.,*`Normalizer`, to generate low-level code for the operation type `Normalize` from the following function pool $\mathcal{F}_{\text{Norm}}$.

- `to-numerical`. This function standardizes a column to a numerical type, such as int or float. To use it, two arguments need to be specified: `column`, representing the source column, and `func`, representing a lambda function. For example, given the table T_5 in Figure 2b, we use this function to standardize the `Females` column to integers, with `column` as `Females` and `func` as `lambda x: int(x.replace(',', ''))`.
- `format-datetime`. The function standardizes the format of columns with `DateTime` values. It has two arguments: `column` and `format`. The `format` argument specifies the desired format for standardization. Given the table T_6 in Figure 2b, to make the values in the `Date` column comparable, we use the `format-datetime` function with `column` set to `Date` and `format` set to "%m-%d".
- `clean-string`. To clean string values in tables, we define the function `clean-string`. It has two arguments: `column` and `trans_dict`. The `trans_dict` is a dictionary where each key k and its corresponding value v represent that k in `column` should be replaced with v. Given the original table T_4 in Figure 2b, we set `trans_dict` to `{'Italia': 'ITA'}`.

- `infer`. Similar to the one for the `Augment` operation, this function uses LLMs to deduce values that could not be processed by the functions above.

Function pool for `Filter`. We design the `Normalizer` agent to utilize a function pool $\mathcal{F}_{\texttt{Filter}}$ with a single function, `filter-columns`. This function has one argument, `rel_columns`, which represents a list of question-related column names.

Discussions. Note that the search space is extensible, allowing additional functions or external APIs to be easily incorporated into the function pools, such as specific data prep functions for various semantic types developed in the `dataprep` library [31].

6 EXPERIMENTS

6.1 Experimental Setup

Datasets. We evaluate our proposed framework AutoPrep using two well-adopted benchmarking datasets WikiTQ [30] and TabFact [13] and a very recent benchmark TableBench [40]. The statistics of the datasets are shown in Table 2.

(1) **WikiTQ** contains complex questions annotated by crowd workers based on diverse Wikipedia tables. WikiTQ comprises 17,689 question-answer pairs in the training set and 4,344 in the test set with significant variation in table size. The answers in WikiTQ can be categorized into three forms: (i) string from a cell in the table, (ii) string from analyzing, and (iii) list of cells in the table.

(2) **TabFact** provides a collection of Wikipedia tables, along with manually annotated NL statements. One requires to deduce the relations between statements and tables as “true” (a statement is entailed by a table) or “false” (a statement if refuted by a table). To reduce the experimental cost without losing generality, we choose the small test set provided by [44] which contains 2024 table-statement pairs. The tables in TabFact are much smaller than those in WikiTQ. The answers are binary with nearly equal proportions.

(3) **TabBench** provides an evaluation of table question answering capabilities within four major categories, including multi-hop fact checking (FC), multi-hop Numerical Reasoning (NR), Trend Forecasting and Chart Generation. We evaluate AutoPrep on TabBench-FC and TabBench-NR, where one requires to conduct more complex and multi-hop reasoning over tables for answering a question. The answers in TabBench could be a string or list of items.

Baselines. We consider the following two categories of baselines:

(1) **TQA Methods w/o Data Prep** directly generate the answer or utilize programs to extract the answer from the table without considering data prep. We implement four primary TQA baselines.

End2EndQA (End2End) *[12]* utilizes the in-context learning abilities of LLMs to generate the answer for TQA task based on the supervision of human-designed demonstrations. We implement End2End method with prompt and demonstrations provided by [15].

Chain-of-Thought (CoT) *[39]* prompts LLMs to generate the reasoning process step-by-step before generating the final answer. We implement CoT with the prompt provided by [12].

NL2SQL *[33]* first translates the question into a SQL program and then executes it to get the final answer from the table for the question. We use the prompt from [15] to implement NL2SQL.

Table 2: Statistics of Datasets.

Dataset	# Rec.	# Row.	# Col.	Ans. Types
WikiTQ	22, 033	4∼753	3∼25	string / list (3.05%)
TabFact	2, 024	5∼47	5∼14	true / false (49.60%)
TableBench	886	2∼212	2∼20	string / list (31.49%)

NL2Py uses Python code to process and reason over the tables. To construct the prompt for NL2Py, we use the TQA instances in NL2SQL prompt and manually write the Python code to process the table and generate the final answer.

(2) **TQA Methods with Data Prep**. We also investigate four SOTA TQA methods that consider data prep tasks in their question answering process, as discussed in Table 1.

ICL-Prep. uses few-shot in-context learning (ICL) demonstrations to guide LLMs in generating programs for data prep, as shown in Figure 3. Subsequently, we employ NL2SQL to extract answers from the cleaned tables.

Dater [44] addresses the TQA task by decomposing both the table and the question. It first selects relevant columns and rows to obtain a sub-table and then decomposes the origin question into sub-questions. Dater answers these sub-questions based on the sub-tables to generate the final answer. We use code in [2] for implementation.

Binder [15] enhances the NL2SQL method by integrating LLMs into SQL programs. It uses LLMs to incorporate external knowledge bases and directly answer questions that are difficult to resolve using SQL alone. We utilize the original code provided by [1].

ReAcTable [45] uses the ReAct paradigm to extract relevant data from the table using Python or SQL code generated by LLMs. Once all relevant data is gathered, it asks the LLMs to predict the answer. We run the original code from [4] and keep all settings as default. Notice that the original code does not include prompts for TabFact, we generate it based the WikiTQ prompt.

Chain-of-Table (CoTable) [38] enhances the table reasoning capabilities of LLMs by predefining several common atomic operations (including data prep operations) that can be dynamically selected by the LLM. These operations form an “operation chain” that represents the reasoning process over a table and can be executed either via Python code or by prompting the LLM. We implement CoTable using the original code from [3].

Evaluation Metrics. We adopt the evaluator in Binder [15] to avoid the situation where program executions are likely to be semantically correct but fail to match the golden answer exactly. For example, when the SQL outputs 1/0 for yes/no questions, the Binder evaluator will conduct a pre-matching check and translate it to boolean values first. It will avoid mismatches of correct prediction and golden answer, such as pairs like 7.45 and 7.4499, 4 years and 4. Afterwards, we adopt accuracy as our evaluation metric.

Backbone LLMs. We evaluate our method using representative LLMs as backbones. For closed-source LLMs, we select DeepSeek [19] (DeepSeek-V2.5-Chat) and GPT3.5 [8] (GPT3.5-Turbo-0613). For open-source LLMs, we choose Llama3 [16] (Llama-3.1-70B-Instruct) and QWen2.5 [41] (QWen2.5-72B-Instruct) for evaluation.

Experiment Settings. We provide **detailed prompts** of each

Table 3: Improvement of data prep for TQA (the best results are in bold and the second-best are underlined).

Method	DeepSeek		GPT3.5		Llama3		QWen2.5	
	WikiTQ	TabFact	WikiTQ	TabFact	WikiTQ	TabFact	WikiTQ	TabFact
End2End	56.65	81.77	52.56	71.54	58.72	81.27	60.01	81.17
+ AutoPrep	63.14 ↑ 6.49	82.11 ↑ 0.34	61.21 ↑ 8.65	71.79 ↑ 0.25	61.23 ↑ 2.51	84.19 ↑ 2.92	63.42 ↑ 3.41	83.05 ↑ 1.88
CoT	54.95	82.02	53.48	65.37	40.75	80.93	59.67	82.31
+ AutoPrep	61.12 ↑ 6.17	82.26 ↑ 0.24	60.01 ↑ 6.53	74.36 ↑ 8.99	56.01 ↑ 15.26	83.65 ↑ 2.72	62.02 ↑ 2.35	<u>85.67</u> ↑ 3.36
NL2Py	59.35	68.13	53.59	66.15	50.12	76.24	53.02	72.63
+ AutoPrep	<u>65.86</u> ↑ 6.51	<u>87.35</u> ↑ 19.22	<u>64.69</u> ↑ 11.1	<u>84.83</u> ↑ 18.68	<u>62.55</u> ↑ 12.43	<u>85.42</u> ↑ 9.18	<u>68.65</u> ↑ 15.63	**85.72** ↑ 13.09
NL2SQL	52.83	70.21	52.90	64.71	51.80	75.15	56.86	80.09
+ AutoPrep	**66.09** ↑ 13.26	**87.85** ↑ 17.64	**64.75** ↑ 11.85	**84.19** ↑ 19.48	**63.72** ↑ 11.92	**85.72** ↑ 10.57	**68.72** ↑ 11.86	85.33 ↑ 5.24

component in AutoPrep in our technical report [5] due to the space limit. For a fair comparison, we set the maximum token input of all methods as 8192. Moreover, we set the temperature parameter of all methods to 0.01 for reproducibility.

6.2 Improvement of Data Prep for TQA

Exp-1: Impact of question-aware data prep on TQA performance. We integrate AutoPrep into our four TQA baselines w/o data prep, in order to investigate the impact of question-aware data prep on TQA performance. The results are reported in Table 3.

As demonstrated, integrating AutoPrep significantly improves the performance of all evaluated methods. Notably, NL2SQL shows the most substantial gains, achieving an average accuracy improvement of **12.22** on WikiTQ and **13.23** on TabFact across all LLM backbones. Similarly, NL2Py also shows notable improvements in its performance, after being integrated with AutoPrep. This significant improvement is attributed to the sensitivity of NL2SQL and NL2Py to data incompleteness and inconsistency, which can lead to erroneous outcomes when performing operations on improperly formatted data. Thus, data prep operations, such as `Augment` and `Normalize` can solve these cases and improve the overall results.

Moreover, End2End and CoT methods also show considerable performance gains. For example, End2End shows an average accuracy gain of **5.27** and **1.35** on WikiTQ and TabFact respectively. These improvements are largely due to AutoPrep 's filtering mechanism, which removes irrelevant columns, thereby simplifying the reasoning process for extracting answers directly from tables.

Finding 1: The integration of AutoPrep improves the accuracy of TQA methods w/o data prep by 12.22 points on WikiTQ and 13.23 points on TabFact, respectively.

Since "NL2SQL + AutoPrep" achieves the best accuracy in most cases, we take its results as default for further comparison.

6.3 Data Prep Method Comparison

Exp-2: Comparison of AutoPrep with previous SOTA TQA methods with data prep. We compare AutoPrep with TQA methods that consider data prep tasks in their question answering process, using a single LLM agent. The results are reported in Table 4.

The results clearly demonstrate that AutoPrep achieves the new SOTA performance on both the WikiTQ and TabFact datasets across all LLM backbones. In particular, AutoPrep outperforms ICL-Prep with a significant accuracy improvement by **11.00** and **8.33** on WikiTQ and TabFact on average, respectively. Other baselines, except Dater, generally outperform ICL-Prep by introducing specific technical optimizations to data prep. For instance, ReAcTable applies the ReAct paradigm to iteratively generate data prep operations rather than producing all code at once, and it explicitly considers data prep tasks (e.g., filtering and augmentation). Despite this, AutoPrep still outperforms the second best method by considerable improvement. Specifically, when compared with CoTable, AutoPrep achieves better performance, with an average accuracy improvement by **3.05** and **1.96** on WikiTQ and TabFact, respectively.

These improvements are primarily attributed to our multi-agent framework, which effectively tackles the question-aware data prep challenge. The experimental results indicate that data prep is inherently complex and is hard to be resolved by a single, one-size-fits-all solution; instead, a more effective approach is to develop specialized methods for each type. A centralized module can then determine which operations to perform and assign them accordingly. Moreover, AutoPrep covers a broader range of data prep operations than these state-of-the-art TQA methods, thereby addressing gaps (*e.g.*, normalization tasks) not fully explored by previous solutions.

Finding 2: AutoPrep outperforms the the previous SOTA TQA methods with data prep by 3.05 points on WikiTQ and 1.96 points on TabFact, respectively,

In addition, we find that DeepSeek achieves the better overall performance at lower API costs. Thus, we select DeepSeek as our default backbone LLM for subsequent experiments.

Exp-3: Evaluation of AutoPrep on tables with various sizes. To further investigate AutoPrep, we analyze its performance across tables of varying sizes. While all tables in TabFact are small, we categorize the tables in WikiTQ into Small (fewer than 2048 tokens), Medium (2048 to 4096 tokens), and Large (more than 4096 tokens), resulting in 4040 Small tables, 200 Medium tables, and 104 Large tables. The results are reported in Figure 7.

We observe that all baselines exhibit unstable performance, particularly on larger tables. For instance, although CoTable achieves the highest average accuracy among previous SOTA methods, it suffers an accuracy drop of **11.78** when reasoning over large tables. Likewise, ReAcTable shows relatively stable performance on medium tables (dropping by **2.15**), yet its performance on large tables remains unsatisfactory (dropping by **17.53**).

In contrast, AutoPrep achieves the highest and most stable performance across tables of varying sizes. When processing medium and large tables, the accuracy of AutoPrep drops by **2.29** and **3.79**. The key to its stability is that it employs specialized agents for each

Table 4: Experimental results of AutoPrep and TQA methods with data prep.

Method	DeepSeek		GPT3.5		Llama3		QWen2.5	
	WikiTQ	TabFact	WikiTQ	TabFact	WikiTQ	TabFact	WikiTQ	TabFact
ICL-Prep	56.54	80.53	55.71	73.91	50.05	75.20	57.00	80.14
Dater	48.32	83.05	52.81	72.08	43.53	74.01	58.78	79.84
Binder	56.81	82.81	56.74	79.17	50.51	78.16	55.43	81.72
ReAcTable	64.13	85.71	51.80	72.80	58.01	80.00	60.15	81.67
CoTable	64.53	86.22	59.94	80.20	62.22	85.62	64.41	83.20
AutoPrep	**66.09**	**87.85**	**64.75**	**84.19**	**63.72**	**85.72**	**68.72**	**85.33**

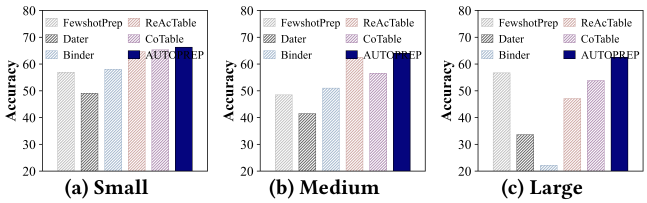


Figure 7: Comparison of AutoPrep and other data prep methods on tables with different sizes on the WikiTQ dataset.

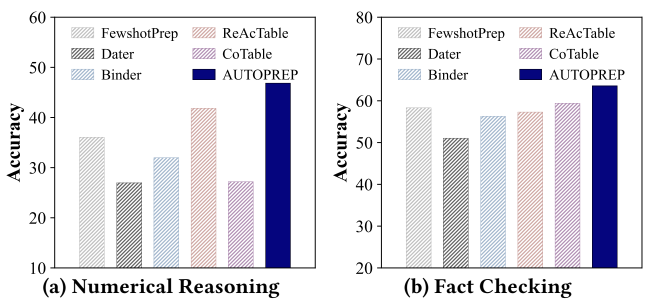


Figure 8: Evaluating Generalization on the TabBench dataset.

data prep task, mitigating the issue of exceeding prompt length limits. Furthermore, each agent generates program-based operators to handle data prep on the entire table, thereby avoiding information loss that occurs when large tables are cropped.

Finding 3: By employing multiple agents and program-based operations, AutoPrep maintains stable performance as table size grows, ensuring that each data prep task is handled effectively without overwhelming a single model.

6.4 Evaluation on Generalization

Exp-4: Evaluating generalization on unseen datasets. As discussed previously, we evaluate the generalization capabilities of AutoPrep on two TabBench datasets. Specifically, we directly use the designed prompting strategies on the WikiTQ dataset, and examine whether these strategies can be generalized to TabBench.

As shown in Figure 8, AutoPrep achieves the highest overall accuracy among all methods. Specifically, compared with the second best method ReAcTable, AutoPrep improves by **5.28**, indicating the strong generalization capability of our method. The main reason is that AutoPrep utilizes tool-augmented method for physical plan generation and execution, generalizing well on unseen datasets.

Finding 4: Utilizing tool-augmented execution enhances the generalization capabilities of AutoPrep on unseen datasets.

6.5 Ablation Studies

Exp-5: Evaluation on the Planner agent. We compare two Planner variants with different high-level operation suggestion methods, namely Direct Prompting and our proposed Chain-of-Clauses method, and report the results in Table 5a.

We observe that Chain-of-Clauses outperforms Direct Prompting by 7.92, 4.50, 5.27 in accuracy on WikiTQ, TabFact and TabBench respectively. This indicates the superiority of our proposed method in generating more accurate high-level operations. Moreover, we find that the performance improvement on WikiTQ is more significant than that on other datasets. This is because the WikiTQ dataset has relatively large tables and complex questions, which could make the high-level operation suggestion problem more challenging to be solved using Direct Prompting.

Finding 5: Our proposed Chain-of-Clauses method is more effective in suggesting high-level operations.

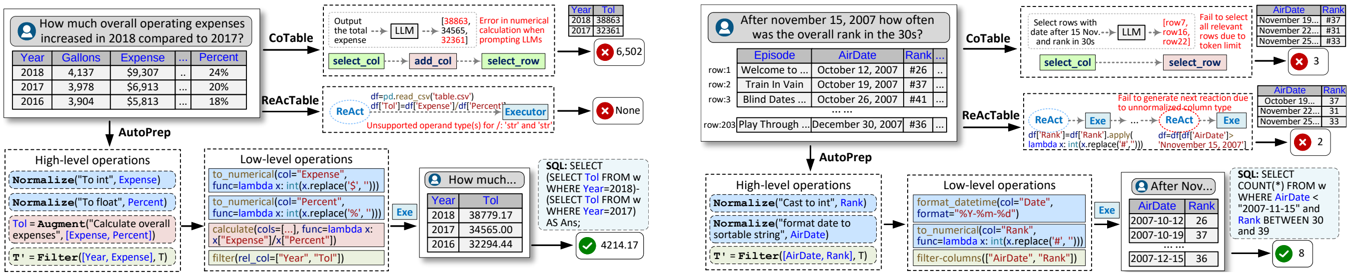
Exp-6: Evaluation on the Programmer Agents. We change the low-level operation generation methods in the Programmer agents and keep other settings of AutoPrep as default.

As illustrated in Table 5b, our proposed tool-augmented method achieves better performance compared with the code generation method. Considering the high-level operations input to the Programmer agents are the same, we can conclude that selecting a function from a function pool and then completing its arguments can generate more accurate and high-quality programs to implement the high-level operations. Moreover, when using demonstrations constructed from same table-question pairs for the prompt of programmer agents, Tool-augmented method can save 18.07% input tokens for low-level operation generation. We also record the error ratio of these two methods, as shown in Table 5b. The probability of bugs in our tool-augmented method is greatly reduced (e.g., from 4.51% to 1.50%), demonstrating its effectiveness.

Finding 6: Our tool-augmented method performs better in both accuracy and cost, with fewer program errors.

Exp-7: Contribution of each agent in AutoPrep. We ablate each Programmer agent including Filter, Augmenter and Normalizer and compare the performance with AutoPrep.

As shown in Table 5c, for WikiTQ, without column augmentation, the accuracy drops the most (4.30). For TabFact and TabBench, the Normalizer matters the most with an accuracy drop by 4.06 and 8.52. This is because that WikiTQ has more instances requiring string extraction or calculation to generate new columns for answering the question, while normalization is a primary issue



(a) Case one (b) Case two

Figure 9: Two case studies selected from TabBench and WikiTQ to qualitatively analyze effectiveness of AutoPrep.

Table 5: Experimental Results of Ablation Studies.

(a) Evaluation on the Planner agent.

Method	WikiTQ	TabFact	TabBench
Direct Prompting	58.17	83.35	44.83
Chain-of-Clauses	**66.09**	**87.85**	**50.10**

(b) Evaluation on the Programmer Agents.

Method	Metric	WikiTQ	TabFact	TabBench
Code Generation	Acc ↑	62.82	81.97	47.26
	Err ↓	4.51%	4.50%	4.73%
Tool Augmented	Acc ↑	**66.09**	**87.85**	**50.10**
	Err ↓	**1.50%**	**0.05%**	**1.01%**

(c) Contribution of Each Programmer Agent in AutoPrep.

Method	WikiTQ	TabFact	TabBench
AutoPrep	**66.09**	**87.85**	**50.10**
- Filter	62.78 (−3.31)	84.98 (−2.87)	47.67 (−2.43)
- Augmenter	61.79 (−4.30)	85.67 (−2.18)	45.44 (−4.66)
- Normalizer	62.02 (−4.07)	83.79 (−4.06)	41.58 (−8.52)

in TabFact and TabBench. Moreover, for all datasets, each agent plays an essential role in data preparation for TQA, which brings accuracy improvement by at least 2.43, 2.18 and 4.06.

Finding 7: Solving three types of data preparation tasks in TQA methods in a specialized manner improves the downstream TQA performance, which demonstrates the rationality of the multi-agent design in AutoPrep.

6.6 Case Studies

This section presents two illustrative examples from TabBench and WikiTQ to qualitatively analyze effectiveness of AutoPrep.

To answer the question in Figure 9a, CoTable generates an operator chain with an "add_col" operator, which is used to generate a new column `Tol` based on the division results of two existing columns `Expense` and `Percent`. This operator directly prompts an LLM to output a list containing all values of the new column, which is a challenging task for previous LLMs. Thus, CoTable generates a new column with two wrong values which leads to a wrong answer. For ReAcTable, although it generates a logically correct Python program, it ignores the types of existing columns which do not support for numerical calculation. Thus, ReAcTable also fails to augment the `Tol` column. AutoPrep addresses this issue by first utilizing a Planner agent to generate logical operators specifying data prep requirements, i.e., two `Normalize` to normalize column `Expense` and `Percent`, a `Augment` to generate the overall expenses `Tol` and a `Filter` to select related columns `Year` and `Tol`. These logical operators are passed to Programmer agents to generate physical operators consisting of a pre-defined Python function. For example, we call `to-numerical` function two times to cast `Expense` and `Percent` to numerical values. All physical operators are executed to process the original table and output a prepared table. Finally, an Analyzer agent based on NL2SQL method generates a SQL query to extract the correct answer "4214.17".

Figure 9b illustrates a TQA instance involving large tables and inconsistent values. To answer the question, CoTable selects all relevant rows for the question. However, it loses related information due to the maximum input token limitation. Specifically, the table is cropped to include only the first 100 rows to fit within the prompt. Thus, all data in the last 103 rows is lost. Consequently, it generates a sub-table table with incomplete data, resulting in an incorrect answer "3". For ReAcTable, it reacts in multiple steps to process the table. When filtering records with `Date` after "november 15, 2007", it does not transform the `Date` into comparable format before using operator ">" to filter the records. Thus, ReAcTable also generates an error answer. AutoPrep solves this by splitting the TQA question into two phases, i.e., data prep and data analysis. It first generates three logical operators including two `Normalize` and one `Filter`. Next, the Programmer agents implement them with corresponding physical operators, which are executed to produce a prepared table. Based on this, the Analyzer agent extracts the final answer "8".

7 CONCLUSION

In this paper, we have introduced AutoPrep, an LLM-based multi-agent framework designed to support automatic data prep for TQA tasks. AutoPrep consists of three stages: (1) the Planning stage, which suggests high-level data prep operations; (2) the Programming stage, which generates low-level implementations for each high-level operation; and (3) the Executing stage, which runs the Python code and iteratively debugs. We also propose a Chain-of-Clauses method to generate high-quality logical plans and a Tool-augmented method for effective physical plan generation. Additionally, we introduce an effective method to ensure successful execution of the physical plan. Extensive experiments on WikiTQ and TabFact demonstrate that: (1) AutoPrep achieves state-of-the-art performance in data prep for TQA, and (2) integrating AutoPrep's data prep improves the overall performance of TQA methods.

REFERENCES

[1] 2023. *Code of Binder.* https://github.com/xlang-ai/Binder
[2] 2023. *Code of Dater.* https://github.com/AlibabaResearch/DAMO-ConvAI
[3] 2024. *Code of Chain-of-Table.* https://github.com/google-research/chain-of-table
[4] 2024. *Code of ReAcTable.* https://github.com/yunjiazhang/ReAcTable
[5] 2025. *Technical Report.* https://github.com/fmh1art/AutoPrep/tree/main/pdf/report.pdf
[6] Rami Aly, Zhijiang Guo, Michael Schlichtkrull, James Thorne, Andreas Vlachos, Christos Christodoulopoulos, Oana Cocarascu, and Arpit Mittal. 2021. FEVEROUS: Fact Extraction and VERification Over Unstructured and Structured information. In *35th Conference on Neural Information Processing Systems, NeurIPS 2021.* Neural Information Processing Systems foundation.
[7] Laure Berti-Équille. 2019. Learn2Clean: Optimizing the Sequence of Tasks for Web Data Preparation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 2580–2586. https://doi.org/10.1145/3308558.3313602
[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS 2020*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).
[9] Chengliang Chai, Nan Tang, Ju Fan, and Yuyu Luo. 2023. Demystifying Artificial Intelligence for Data Preparation. In *SIGMOD*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 13–20. https://doi.org/10.1145/3555041.3589406
[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024).
[11] Sibei Chen, Nan Tang, Ju Fan, Xuemi Yan, Chengliang Chai, Guoliang Li, and Xiaoyong Du. 2023. HAIPipe: Combining Human-generated and Machine-generated Pipelines for Data Preparation. *Proc. ACM Manag. Data* 1, 1 (2023), 91:1–91:26. https://doi.org/10.1145/3588945
[12] Wenhu Chen. 2022. Large language models are few (1)-shot table reasoners. *arXiv preprint arXiv:2210.06710* (2022).
[13] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2019. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164* (2019).
[14] Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. HiTab: A Hierarchical Table Dataset for Question Answering and Natural Language Generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 1094–1110.
[15] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2023. Binding Language Models in Symbolic Languages. In *International Conference on Learning Representations (ICLR 2023)(01/05/2023-05/05/2023, Kigali, Rwanda).*
[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[17] Ori Bar El, Tova Milo, and Amit Somech. 2020. Automatically Generating Data Exploration Sessions Using Deep Reinforcement Learning. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1527–1537. https://doi.org/10.1145/3318464.3389779
[18] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074* 24 (2020), 8.
[19] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024). https://doi.org/10.48550/ARXIV.2401.14196 arXiv:2401.14196
[20] Yuval Heffetz, Roman Vainshtein, Gilad Katz, and Lior Rokach. 2020. DeepLine: AutoML Tool for Pipelines Generation using Deep Reinforcement Learning and Hierarchical Actions Filtering. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 2103–2113. https://doi.org/10.1145/3394486.3403261
[21] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).
[22] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated machine learning: methods, systems, challenges.* Springer Nature.
[23] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183* (2024).
[24] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv preprint arXiv:2405.11403* (2024).
[25] Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: recent advances. In *China Conference on Knowledge Graph and Semantic Computing.* Springer, 174–186.
[26] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context LLMs Struggle with Long In-context Learning. *CoRR* abs/2404.02060 (2024). https://doi.org/10.48550/ARXIV.2404.02060 arXiv:2404.02060
[27] Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, et al. 2022. FeTaQA: Free-form Table Question Answering. *Transactions of the Association for Computational Linguistics* 10 (2022), 35–49.
[28] Randal S Olson and Jason H Moore. 2016. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning.* PMLR, 66–74.
[29] Vaishali Pal, Andrew Yates, Evangelos Kanoulas, and Maarten de Rijke. 2023. MultiTabQA: Generating Tabular Answers for Multi-Table Question Answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 6322–6334.
[30] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* 1470–1480.
[31] Jinglin Peng, Weiyuan Wu, Brandon Lockhart, Song Bian, Jing Nathan Yan, Linghao Xu, Zhixuan Chi, Jeffrey M. Rzeszotarski, and Jiannan Wang. 2021. DataPrep.EDA: Task-Centric Exploratory Data Analysis for Statistical Modeling in Python. In *SIGMOD.* ACM, 2271–2280. https://doi.org/10.1145/3448016.3457330
[32] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* 6 (2023).
[33] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498* (2022).
[34] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1171–1188. https://doi.org/10.1145/3299869.3319863
[35] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. *arXiv preprint arXiv:2403.17927* (2024).
[36] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.* 847–855.
[37] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.
[38] Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, et al. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv preprint arXiv:2401.04398* (2024).
[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[40] Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xinrun Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. 2024. TableBench: A Comprehensive and Complex Benchmark for Table Question Answering. *arXiv preprint arXiv:2408.09174* (2024).
[41] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* (2024).
[42] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang,

and B. Aditya Prakash (Eds.). ACM, 1446–1456. https://doi.org/10.1145/3394486.3403197

[43] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[44] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 174–184.

[45] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1981–1994.

[46] Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models. *Proc. VLDB Endow.* 17, 12 (2024), 3920–3933. https://www.vldb.org/pvldb/vol17/p3920-zhu.pdf