TEXT2DB : Integration-Aware Information Extraction with Large Language Model Agents

Yizhu Jiao, Sha Li, Sizhe Zhou, Heng Ji, Jiawei Han
University of Illinois Urbana-Champaign
yizhuj2@illinois.edu

Abstract

The task of information extraction (IE) is to extract structured knowledge from text. However, it is often not straightforward to utilize IE output due to the mismatch between the IE ontology and the downstream application needs. We propose a new formulation of IE TEXT2DB that emphasizes the integration of IE output and the target database (or knowledge base). Given a user instruction, a document set, and a database, our task requires the model to update the database with values from the document set to satisfy the user instruction. This task requires understanding user instructions for *what to extract* and adapting to the given DB/KB schema for *how to extract* on the fly. To evaluate this new task, we introduce a new benchmark featuring common demands such as data infilling, row population, and column addition. In addition, we propose an LLM agent framework OPAL (Observe-Plan-Analyze LLM) which includes an Observer component that interacts with the database, the Planner component that generates a code-based plan with calls to IE models, and the Analyzer component that provides feedback regarding code quality before execution. Experiments show that OPAL can successfully adapt to diverse database schemas by generating different code plans and calling the required IE models. We also highlight difficult cases such as dealing with large databases with complex dependencies and extraction hallucination, which we believe deserve further investigation.

1 Introduction

Text has always been seen as a rich source of information, and information extraction (IE) is defined as the task of extracting knowledge from unstructured text. However, a long-overlooked question is what counts as "relevant knowledge": the entity, relation, and event types that require extraction (Ding et al., 2021; Wan et al., 2023; Li et al., 2021). Current methods sidestep this question by either assuming that "relevant knowledge" is given by the ontology (Weischedel et al., 2013) in the closed domain setting or assuming that all knowledge is relevant in the OpenIE setting (Muhammad et al., 2020). We argue that the scope of relevant knowledge is highly dependent on the downstream task, especially when IE output needs to be ingested into databases or knowledge bases. We call such a setting *integration-aware information extraction*, where we take a holistic view and consider both the source of IE and also the consumer of IE results. In the database community, integration refers to the alignment between schemas of different databases. We borrow this term to refer to the alignment of IE output and the target database, or in other words, the integration of structured and unstructured information. Data integration of IE results is critical as (1) a database system provides the infrastructure to support large-scale data management and execution of complex analytical queries; and (2)

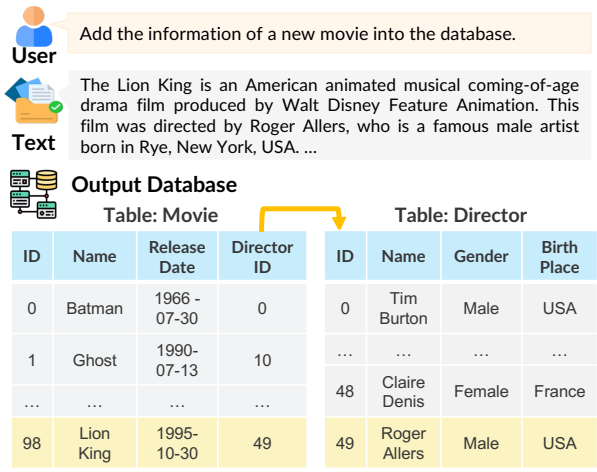

Figure 1: Our TEXT2DB task is defined over a database, a user instruction, and a document set. The model aims to fulfill the user instruction by updating the database with values (shown in yellow) extracted from text. In this example, the input database has two tables linked with the foreign key constraint (`DirectorID` in the Movie table refers to `ID` of the Director table).

Findings of the Association for Computational Linguistics: ACL 2024, pages 185–205

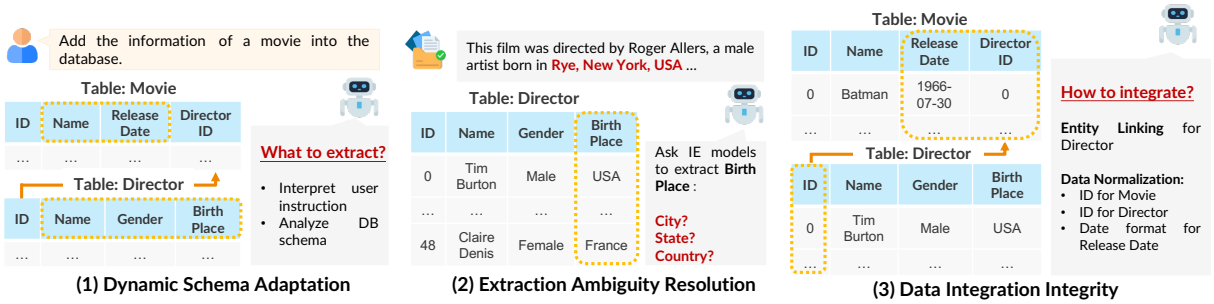

Figure 2: Three major challenges of the TEXT2DB task: (1) dynamically decide what to extract by analyzing complex database schemas and interpreting user instructions; (2) resolve extraction ambiguity to ensure extracted values match the semantics and granularity of existing database content; (3) integrate the extracted data into the database while maintaining integrity and consistency.

real-life applications often involve structured data stored in databases that complement IE results (*e.g.*, an e-commerce website has a proprietary product database but might wish to extract user feedback from comments).

As an instance of this *integration-aware IE* setting, we propose a new task TEXT2DB. Specifically, each instance includes a target database (with existing data), a document set, and a user instruction (Figure 1). The user instruction will provide high-level guidance on which type of action to perform ("Add information about new movies") and the model is required to extract the relevant information from the documents and update the database accordingly. Since each instance deals with a different database, the model must be able to automatically infer which fields (entities, relations, events, attributes) are relevant according to the user instruction and database schema. This is not possible with conventional IE models since the ontology is usually built into the model and each model typically can only handle one type of extraction task. In addition, the granularity of extraction could be ambiguous when examining the user instruction alone, and analysis of the database content is key to coming up with a precise plan of action. Finally, even after the values are extracted from the document, the IE output might need to be normalized before being added to the database. We summarize these key challenges in Figure 2.

To benchmark the TEXT2DB task, we introduce an annotated dataset. The dataset includes three high-level categories of instructions: data infilling, row addition, and column addition. It incorporates two sources of databases – simple schemas derived from Wikipedia tables and more complex schemas manually selected from BIRD (Li et al., 2023). Additionally, the dataset spans various domains, to test the generalization ability across different areas. The dataset is classified into three difficulty levels – easy, medium, and hard – based on the complexity of the database schema, the length of the source document, and the number of values to update.

While large language models (LLMs) show strong capability of instruction-following and extraction, directly performing the complex task of TEXT2DB results in unsatisfactory performance, and scaling up to actual databases would be prohibitively expensive. We propose a large language model-based agent framework OPAL that incorporates multiple IE models as tools. At its core, a Planner agent decomposes the user instruction into a code-based plan, which involves data transformations and external calls to specialized models (*e.g.* named entity recognition, relation extraction, entity linking). The Analyzer checks the syntax and logic in the program and provides feedback to the planner for iterative improvement. Simultaneously, the Observer agent is also incorporated to interface with the databases, aiding in database schema analysis, tool selection, and test case generation, thereby ensuring the system's robustness and efficiency.

Our experiments demonstrate the effectiveness of our overall framework and individual components. We find that equipping the Planner with feedback for self-revision is critical and IE demonstrations from the database help resolve extraction ambiguity and eventually boost extraction quality.

To conclude, our contributions include (1) we define a new task TEXT2DB which is an example of *integration-aware information extraction*, (2) we introduce a new benchmark for TEXT2DB with diverse databases and instructions of different difficulty, (3) we design a new LLM agent framework OPAL (Observe-Plan-Analyze LM).

2 The TEXT2DB Task

Our task is defined over a set of (user instruction I, database B, documents $\mathcal{D}$) instances. The goal is to automatically update the database ($B \rightarrow B^*$) with new information extracted from a set of text documents to fulfill the user's request. The user instruction I is a natural language sentence indicating the high-level scope and the type of operation. The database B contains multiple tables $\mathcal{T}$ that can be filled with pre-existing data entries. The database schema is available to the model, which outlines the data types, constraints, relationships, and integrity rules among different tables.

Unlike creating a database from scratch, our task focuses on enriching an existing database B with a collection of text documents $\mathcal{D}$. This setting is more realistic but also more challenging since information to be extracted from $\mathcal{D}$ must be aligned with the schema and data in B.

3 TEXT2DB Benchmark

	#DB	#Table	#Row	#Col.	ΔValue	#Ins.
Wiki	191	1.0	116.3	5.7	6.2	195
Bird	12	9.1	297K	55.3	17.7	45

Table 1: Database Comparison. The WikiTable subset of our dataset emphasizes schema diversity whereas the BIRD subset emphasizes database size and complexity. "Ins." denotes the number of instances. More details are provided in the Appendix.

Our benchmark construction starts with selecting a set of databases $\mathcal{B}$ to work with, then finding relevant documents $\mathcal{D}$ and annotating instructions $\mathcal{I}$ and the updated databases $\mathcal{B}'$.

3.1 Database Selection

We use tables from Wikipedia and databases from an existing dataset BIRD.

We outline the selection criteria and preprocessing procedure for the two sources below:

- **WikiTables.** The advantage of this data source is that we have a board domain coverage and a natural matching between the tables and documents. We transform these tables into databases by specifying the primary key for each table and performing data cleaning. This involves standardizing column names, removing rows with incomplete values, and excluding descriptive columns that cannot be directly extracted from text.
- **BIRD** (Li et al., 2023). Databases in BIRD feature multiple tables and complex schemas which introduce dependencies between tables. We exclude databases that do not contain any column that can be found in public text, focusing instead on those with accessible licenses and real-world applicability.

3.2 Annotation Process

Data	Task Types DI	RP	CA	Total
User Instruction				
# Avg. Words	24.4	21.2	48.0	31.1
Source Text				
# Avg. Docs	1.0	2.0	5.2	2.7
# Avg. Words	1,099.0	1,541.9	2,739.1	1786.5
Database				
# Databases	73	72	73	203
# Avg. Tables	2.6	2.6	2.4	2.5
# Avg. Rows	105K	33K	29K	56K
# Avg. Columns	16.2	12.7	16.1	15.0
Δ Values	1.9	11.2	12.1	8.4
Difficulty				
# Easy	39	10	32	81
# Medium	17	38	27	82
# Hard	25	32	20	77
Overall				
# Data Instances	81	80	79	240
# Domains	23	32	29	45

Table 2: Statistics of our dataset. "DI", "RP", and "CA" correspond to three task types, data infilling, row population, and column addition, respectively. "#" indicates the count, "Avg." stands for the average value per instance, and "Δ Values" represents the number of value changes in the process of database population.

We include three general categories of database updates in our benchmark: Data Infilling, Row Population, and Column Addition, detailed below.

- **Data Infilling** aims to fill in missing values for existing rows. The rows to update are specified by the user or automatically decided by the system. In these rows, the system updates all columns with missing values by default if not specified by users.
- **Row Population** typically adds 1-3 new rows, with the most difficult cases adding up to 10 rows. For each new row, the model should populate as many columns as possible, based on the information available in texts. Otherwise, the default values (as defined in the DB schema) should be inserted.

Difficulty	Criteria	Number
Easy	# Table = 1 and ΔValues $\leq$ 10 and Avg. Words $\leq$ 1k	81
Medium	# Table = 1 and 10 < ΔValues $\leq$ 20 and 1k < Avg. Words $\leq$ 2k	82
Hard	# Table > 1 or ΔValues > 20 or Avg. Words > 2k	77

Table 3: Criteria of three difficulty levels. "Avg. Words" represents the average number of words per document.

- **Column Addition** generally adds one to three new columns to a specific table. The instruction should specify the name, meaning, and default value for the new columns, with any special formatting requirements if applicable. The system needs to decide which rows the new values should be linked to.

After selecting a database and the operation category, the human annotator will write a clear and concrete instruction, find related documents, and modify the values in the database to serve as the ground truth for evaluation. For detailed guidelines, see Appendix A.

3.3 Statistics

Our evaluation benchmark includes 240 data instances across 203 databases, showcasing a variety of schemas with an average of 2.5 tables (including 56K rows and 15 columns per database on average). The complexity spans from databases with a single table to those with up to 21 tables. Each task in the dataset aims to populate an average of 8.4 values, based on instructions averaging 31 words in length. The overall statistics are shown in Table 2. The domain distribution within the dataset is well-rounded, featuring significant representations from entertainment (15.4%), sport (9.2%), art (8.3%) and other areas, ensuring comprehensive domain coverage.

The dataset is categorized into three difficulty levels, easy, medium, and hard based on the schema complexity, the size of the required update, and the length of input texts. The criteria for determining the difficulty level and difficulty distribution across categories are shown in Table 3. Note that the size of the required update also positively correlates with the difficulty of the level.

4 The OPAL Framework

We introduce the OPAL (Observe-Plan-Analyze Language Model) framework, which starts with observing the target database, then generates the plan of action in code by referring to the database and IE tools and catches errors by both static and dynamic code analysis. The OPAL framework is an instance of an LM agent framework that interacts with a database environment, utilizes IE models as tools, executes actions as code, and improves its plan and tools using feedback from the database.

```
# Determine the task based on the user instruction
task_type = 'data infilling'
# List the attributes mentioned by the user instruction.
attribute = "Country"
# Extract the blizzard name
blizzard_names = NER(text, "blizzard")
all_data = []
for _id, blizzard_name in enumerate(blizzard_names):
    # Extract the essential attributes
    blizzard_attributes = AE(text, blizzard_name, attribute)
    blizzard_attributes['Name'] = blizzard_name
    all_data.append(blizzard_attributes)
# Normalize the extracted attributes
norm_data = Norm(all_data, database, table_name='blizzard')
# Migrate the new values into the database
DI(norm_data, database, table_name='blizzard')
```

Figure 3: Example of the generated code in one pass, which misconfigures the attribute extraction tool since the tool expects a list of attributes rather than a string.

4.1 PLANNER: The Function-Calling Agent

Since updating a database per user instruction typically requires extracting multiple related fields, instead of asking a model to perform the task end-to-end, following the idea of ViperGPT (Sur'is et al., 2023) and VisProg (Gupta and Kembhavi, 2023), we first decompose the task into a series of steps represented as code. Each of these steps can either be directly executable code or external API calls to a set of IE models (*i.e.* function calls).

Concretely, the input context C consists of the system prompt C_0, the user instruction I and potential input O from the Observer (introduced in Sec. 4.3). The system prompt C_0 includes the code APIs for the available tools and in-context examples. In the output, inspired by ReAct (Yao et al., 2022), we allow the model to generate both code actions and natural language thoughts. The model is free to choose when to output an action and when to output a thought (represented as a comment).

In our work, we define 10 different tools for the Planner agent to use as shown in Table 4, spanning standard IE tasks and database primitives. More

Tools	API Signature
Information Extraction	
Named Entity Recognition	NER (text: str, type: str) → list[str]
Relation Extraction	RE (text: str, head_e:str, relation: str) → list[str]
Attribute Extraction	AE (text: str, entity: str, attribute_list: list) → dict
Text Classification	Classify (text: str, label_list: list) → str
Database Integration	
Entity Linking	Link (data_entries: list, database: dict, table_name: str) → list
Data Normalization	Norm (data_entries: list, database: dict, table_name: str) → list
Data Infilling	DI (data_entry: list, database: dict, table_name: str) → dict
Row Addition	PR (data_entries: list, database: dict, table_name: str) → dict
Column Addition	AC (data_entry: list, database: dict, table_name: str, new_columns: list) → dict

Table 4: Tools available to the Planner Agent.

tool descriptions can be found in Appendix B. Figure 3 showcases how the generated code calls for these tools.

4.2 ANALYZER: The Code Feedback Agent

In more complicated cases, the Planner often fails to generate the correct actions (code plan) in a single pass (as shown in Figure 3). If the plan execution attempt is unsuccessful, the error message from the code compiler will be provided to the Planner to guide its self-repair process. While this self-repair process has shown to provide some benefit (Chen et al., 2023; Wang et al., 2023), it is limited by the quality of the feedback (how well the error message explains the mistake) (Olausson et al., 2024) and comes at the cost of executing the code multiple times. In particular, since some of our function calls invoke external models, this process can be very time-consuming. To mitigate this problem and improve code quality, we designed the CODE ANALYZER component, which sits between the Planner and the code compiler, aiming to provide more informative feedback early on.

The input to the Analyzer is the plan of action A (written as code) and the output is the natural language feedback F. The Analyzer provides feedback of three different types:

- **Syntax error feedback**. Syntax errors might appear in the generated native code (in our case, we use the Python language), or in the API calls. These errors can be directly detected by the interpreter and the Analyzer aims to supplement the error messages with natural language feedback.
- **Runtime and logic error feedback**. Similar to how human programmers debug their code with unit tests, the Analyzer takes a few data samples provided by the Observer and generates test cases by mocking the output from external models. If the output does not match the data samples, the Analyzer will generate another piece of feedback to the Planner.
- **Database integrity feedback**. Even when the extraction results are correct, we might not be able to update the database successfully due to database constraints. Thus, we implement the functions for the analyzer to checks against duplicating entries and violating database dependency constraints.

4.3 OBSERVER: The Database Expert Agent

In our setting, each database can consist of one or multiple tables, each with its own schema and data entries. Directly providing the whole database as text (by converting to JSON or Markdown code) could dilute the model's attention to other parts of the input prompt. We introduce the OBSERVER agent which serves as a bridge between the database environment and other components, including the Planner, Code Analyzer, and the IE models. Speficially:

- **Observer → Planner.** The Observer analyzes the schema and content of the database, identifying crucial aspects behind different columns, such as the format, value range, and semantic meaning. Such insights are summarized into a summary observation O, which becomes part of the Planner's input context. The observation informs the Planner on selecting the right API call, and whether data normalization is required. For instance, for a movie database, the Observer recommends an attribute extractor to extract the movie budget and gross. It might suggest a text classifier to categorize

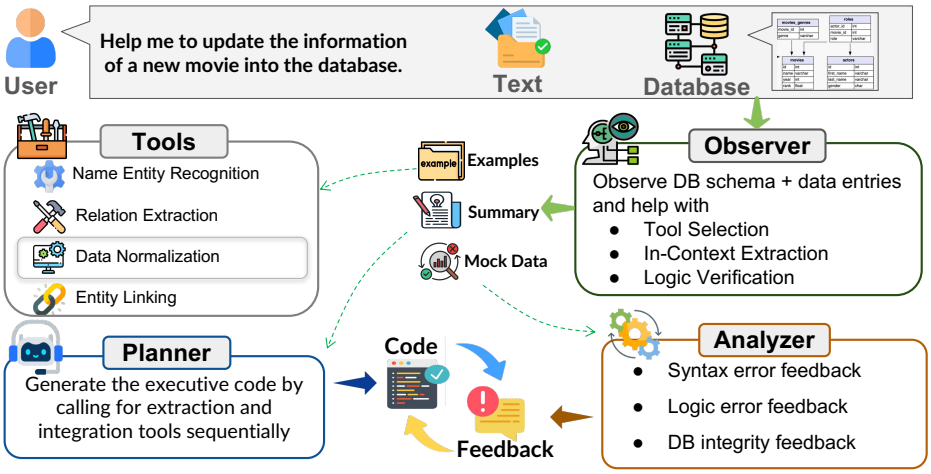

Figure 4: Framework architecture.

the movie after identifying predefined genres (such as action, comedy, and drama) in the database.

- **Observer → IE Models.** The type name derived from the database schema alone could be ambiguous for IE models. For instance, "location" could refer to a country, city, or a specific area, leading to extractions that may misalign with the database contents. To handle this issue, the Observer selectively fetches relevant entries from the database as few-shot demonstrations. These demonstrations guide IE tools to understand what to extract and enable in-context learning if applicable. For example, if the target column `Loc` includes values such as `New York, Los Angeles, Boston`, IE tools can prioritize similar levels of detail (US cities) in the input document.

- **Observer → Analyzer.**The Observer selects data for test cases that simulate the user request to help identify logical errors before running the code. For instance, if the plan misses the mark on how different tables relate to each other, these simulated tests can highlight those errors early on.

5 Experiments

5.1 Experiment Setting

Evaluation Metric We evaluate the models by comparing the database before and after updating, checking the difference ΔB. We represent the database entries in ΔB as a set of structured tuples T, following the form of *(table name, primary key, primary key value, column name, value of that column)*. Each updated entry is ruled as correct if all the fields match the ground truth (Exact Match). Then we compute F1 over all entries in the database update ΔB. The reported metric is **macro-F1**, in other words, F1 averaged over each (instruction, document set, database) instance.

Implementation The planner and observer agents in our OPAL framework are powered by the GPT4 language model `gpt-4-1106-preview`. The maximum number for the Planner to revise plans is 10. The whole process can restart for 2 times at most after failure. For the tool library, we emulate the models with GPT for named entity recognition, relation extraction, attribute extraction, text classification and data normalization. In addition, we adopt an existing entity linking model GENRE (De Cao et al., 2022). More details and prompts can be found in Appendix C.

5.2 Experiment Results

We show our evaluation results in Table 5. We have the following observations:

(1) OPAL vs Template: Dynamic plans are necessary. Our first baseline replaces the planner with a static template which first uses an entity extraction tool to extract the primary key of the table (selected by the Observer) and then goes on to extract each column by using the attribute extraction tool. In our TEXT2DB setting, the diversity of schemas and instruction led to a sharp decline in performance over all slices of the dataset.

(2) OPAL vs One-Shot: Feedback improves plan quality. The plan generated by the Planner in its first attempt is often error-prone. By utilizing the Analyzer and allowing the Planner to make

Models	Difficulty			Task Type			DB Source		Overall
	Easy	Medium	Hard	DI	RP	CA	Wiki	Bird	
Planner									
Template	11.19	8.21	0.35	7.23	12.88	0.00	8.07	0.0	6.73
One-shot	18.77	14.45	2.51	16.25	12.83	7.04	14.85	0.04	12.08
IE tools									
GPT → Small Models	20.63	21.24	13.44	24.21	9.84	21.42	19.10	15.27	18.50
Observer									
- Observer	23.24	22.87	18.49	27.05	14.72	22.95	23.17	14.74	21.59
- DB Analysis	38.55	26.78	22.70	30.55	32.48	24.76	31.83	18.25	29.29
- IE Demonstration	25.96	28.85	15.38	26.18	20.00	24.30	24.15	20.24	23.50
- Simulated Test	42.42	33.79	22.00	38.93	32.77	26.30	36.90	11.81	32.72
Full Model									
OPAL	**42.44**	**36.91**	**23.21**	**38.85**	**37.01**	**26.34**	**36.97**	**21.74**	**34.11**

Table 5: Experiment results. The metric is F_1 (%) of the exact matching score.

multiple rounds of revision, we can achieve a large gain in performance.

(3) Smaller IE models are less capable of zero-shot/few-shot learning. We defined our IE tools as few-shot learners without a fixed ontology since the type names are provided as part of the input. In this setting, we find that LLM-emulated IE models work better than fine-tuned smaller-scaled models (Li et al., 2022; Sainz et al., 2021; Lyu et al., 2021; Gera et al., 2022), due to the strong in-context learning ability of LLMs. However, since many of the errors are from the extraction stage (see Section 5.4), perhaps a better choice would be to automatically route the API call to an LLM or specialized model based on the requested type.

(4) The Observer is most helpful by providing IE demonstrations. Among the multiple functions of the Observer component, we see that selecting a few values from the target table (or target column) to serve as few-shot demonstrations significantly contribute to the final performance. This partially resolves the challenge of *extraction ambiguity*.

(5) The Observer is more useful when the database is large and complex. When the number of tables is larger and there are more dependencies between tables, using the Observer to generate simulated test cases helps improve the plan quality.

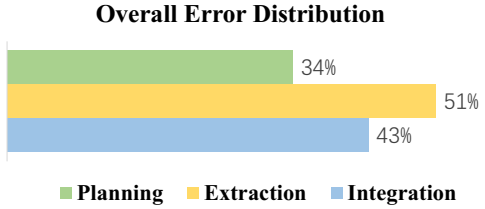

Figure 5: Error distribution of the whole framework.

5.3 Case study

Figure 6 shows a case of updating a movie database. The database includes three tables (`Movie`, `Actor`, `Character`) with two foreign key dependencies. To fulfill the user's instruction, the model must add new entries in all three tables while ensuring database integrity. The planner effectively identified extraction targets across tables and generated appropriate function calls, for instance, using an attribute extractor for the `Gross` column and employing a classifier for the `Genre` column. However, it mistakenly attempted to find `Character.ActorID` by the character name, suggesting a need for better database understanding. Extraction tools accurately identify the desired data from text, though they erroneously make up the actors' "birthplace", pointing to the need to improve extraction faithfulness, possibly through verification mechanisms. Integration is successful, effectively normalizing data formats like release dates and movie gross, and correctly assigning primary keys to new entries.

5.4 Error Analysis

We analyzed 100 wrong cases by classifying them based on the stage of error: planning, extraction, or integration. Our findings in Figure 5 show that 34% of errors occurred during the planning phase, particularly in databases with complex schemas and multiple tables, aligning with our findings in Table 5. Errors in planning were notably more frequent in row addition tasks due to their likelihood of involving multiple tables. Extraction errors are most common, driven by dependencies in the extraction process, such as the need for named entity recognition before relation extraction, leading to error accumulation. In the integration phase, the

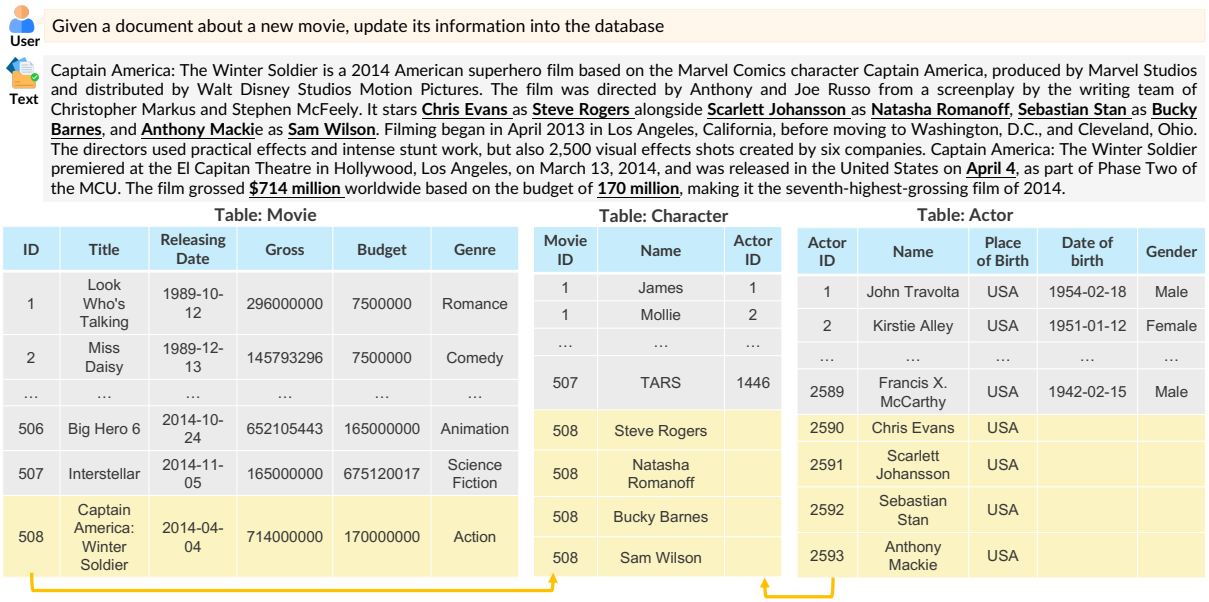

Figure 6: Case study of a row population task on a movie database with three tables. The model successfully extracts relevant information from the document and normalizes the data to conform with the database (`Movie.Releasing Date, Movie.Gross`). However, the model still struggles to deal with table dependencies (`Character.ActorID` should refer to `Actor.ActorID`) and occasionally hallucinates values (`Actor.PlaceOfBirth`).

majority of errors are related to data infilling and column addition, with the bottleneck being entity linking. A more detailed analysis of each stage can be found in Appendix D.

6 Related Work

LLMs for Databases The majority of work involving the application of LLMs to databases focuses on the text-to-SQL task (Yu et al., 2018; Li et al., 2023; Liu et al., 2023) which does not involve extraction. Recently, LLMs with few-shot examples have shown to outperform fine-tuned smaller models (Pourreza and Rafiei, 2023; Zhang et al., 2023; Sun et al., 2023).

The structured view generation task in (Arora et al., 2023) is closest to our TEXT2DB setting. However, their task only requires extraction from semi-structured documents, without the need for integration with existing tables. We note that when the target database is of the form of (entity name, entity attribute), our task resembles knowledge base population (KBP) (Getman et al., 2018). The key difference lies in the fact that our task requires dynamic adaptation to diverse databases.

Tool Learning in LLMs Tool learning, or function-calling, has emerged as a promising approach to extend the capability of large language models. In the tool learning paradigm, certain tools (such as internet search, a calculator, and image generation models) are provided to the LLM. Tool learning can be enabled by prompting with function definitions or specialized fine-tuning (Schick et al., 2023; Tang et al., 2023; Patil et al., 2023; Zeng et al., 2023)[1]. We refer the reader to (Qin et al., 2023) for a comprehensive survey. In the OPAL framework, IE models act as external tools to the Planner. Unlike prior work that leaves all the work to the Planner and assumes that tools as provided as-is, our Observer provides essential insight into the database to support tool selection and selects demonstrations to assist the IE tools.

7 Conclusion and Future Work

In this paper, we propose the new task TEXT2DB which updates a given database using values extracted from a document set following user instructions. TEXT2DB presents unique challenges in dynamic adaptation , extraction ambiguity and integrity requirements. We present a new benchmark and a new LLM agent framework OPAL for this task. In particular, OPAL features 3 components: the Planner, Analyzer, and Observer. We show that OPAL substantially improves update effectiveness over directly generating the plan, with significant gains coming from using demonstrations from the Observer for IE models.

[1]It is speculated that GPT-4 has been fine-tuned to support function calling.

At this point, we have only considered inserting values that do not previously exist into the database. However, conflict resolution has been a long-standing issue in the integration of databases, and we also foresee similar challenges in the integration of documents and databases. In this case, we need to consider the reliability of the document and the confidence of the extraction model.

Limitations

The success of our agent framework OPAL relies heavily on the instruction-following and tool-using ability of the language model. As a proof-of-concept for our new task, we used the most capable LLM GPT4 at the time of writing. Benchmarking different base LLMs would provide extra insight to how the framework generalizes. In addition, We have only experimented with a small set of IE models as tools (one for each API call) and could have been extended to a larger repository of open-source models such as that in Shen et al. (2023).

Acknowledgements

Thanks to Yu Su at the Ohio State University and Tanay Dixit at Universtiy of Illinois at Urbana-Champaign for their enlightening discussions. Research was supported in part by US DARPA KAIROS Program No. FA8750-19-2-1004 and INCAS Program No. HR001121C0165, National Science Foundation IIS-19-56151, and the Molecule Maker Lab Institute: An AI Research Institutes program supported by NSF under Award No. 2019897, and the Institute for Geospatial Understanding through an Integrative Discovery Environment (I-GUIDE) by NSF under Award No. 2118329.

References

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language models enable simple systems for generating structured views of heterogeneous data lakes. *ArXiv*, abs/2304.09433.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128.

Nicola De Cao, Ledell Wu, Kashyap Popat, Mikel Artetxe, Naman Goyal, Mikhail Plekhanov, Luke Zettlemoyer, Nicola Cancedda, Sebastian Riedel, and Fabio Petroni. 2022. Multilingual autoregressive entity linking. *Transactions of the Association for Computational Linguistics*, 10:274–290.

Ning Ding, Guangwei Xu, Yulin Chen, Xiaobin Wang, Xu Han, Pengjun Xie, Haitao Zheng, and Zhiyuan Liu. 2021. Few-nerd: A few-shot named entity recognition dataset. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3198–3213.

Ariel Gera, Alon Halfon, Eyal Shnarch, Yotam Perlitz, Liat Ein Dor, and Noam Slonim. 2022. Zero-shot text classification with self-training. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1107–1119.

Jeremy Getman, Joe Ellis, Stephanie Strassel, Zhiyi Song, and Jennifer Tracey. 2018. Laying the groundwork for knowledge base population: Nine years of linguistic resources for tac kbp. In *International Conference on Language Resources and Evaluation*.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14953–14962.

Bangzheng Li, Wenpeng Yin, and Muhao Chen. 2022. Ultra-fine entity typing with indirect supervision from natural language inference. *Transactions of the Association for Computational Linguistics*, 10:607–622.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Chenhao Ma, Kevin C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *ArXiv*, abs/2305.03111.

Sha Li, Heng Ji, and Jiawei Han. 2021. Document-level event argument extraction by conditional generation. In *Proc. The 2021 Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL-HLT2021)*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Yuxian Gu, Hangliang Ding, Kai Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Shengqi Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2023. Agentbench: Evaluating llms as agents. *ArXiv*, abs/2308.03688.

Qing Lyu, Hongming Zhang, Elior Sulem, and Dan Roth. 2021. Zero-shot event extraction via transfer learning: Challenges and insights. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 322–332.

Iqra Muhammad, Anna Kearney, Carrol Gamble, Frans Coenen, and Paula Williamson. 2020. Open information extraction for knowledge graph construction. In *Database and Expert Systems Applications - DEXA 2020 International Workshops BIOKDD, IWCFS and MLKgraphs, Bratislava, Slovakia, September 14-17, 2020, Proceedings*, volume 1285 of *Communications in Computer and Information Science*, pages 103–113. Springer.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation?

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *ArXiv*, abs/2305.15334.

Mohammad Reza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Neurips*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shi Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bo Li, Ziwei Tang, Jing Yi, Yu Zhu, Zhenning Dai, Lan Yan, Xin Cong, Ya-Ting Lu, Weilin Zhao, Yuxiang Huang, Jun-Han Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023. Tool learning with foundation models. *ArXiv*, abs/2304.08354.

Oscar Sainz, Oier Lopez de Lacalle, Gorka Labaka, Ander Barrena, and Eneko Agirre. 2021. Label verbalization and entailment for effective zero and few-shot relation extraction. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 1199–1212.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Neurips*.

Yongliang Shen, Kaitao Song, Xu Tan, Dong Sheng Li, Weiming Lu, and Yue Ting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *ArXiv*, abs/2303.17580.

Ruoxi Sun, Sercan Ö. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language model adaptation for text-to-sql. *ArXiv*, abs/2306.00739.

D'idac Sur'is, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11854–11864.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *ArXiv*, abs/2306.05301.

Zhen Wan, Fei Cheng, Zhuoyuan Mao, Qianying Liu, Haiyue Song, Jiwei Li, and Sadao Kurohashi. 2023. Gpt-re: In-context learning for relation extraction using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3534–3547.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2023. Leti: Learning to generate from textual interactions. In *arxiv*.

Ralph Weischedel, Martha Palmer, Mitchell Marcus, Eduard Hovy, Sameer Pradhan, Lance Ramshaw, Nianwen Xue, Ann Taylor, Jeff Kaufman, Michelle Franchini, et al. 2013. Ontonotes release 5.0 ldc2013t19. *Linguistic Data Consortium, Philadelphia, PA*, 23.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *ArXiv*, abs/2310.12823.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. ACT-SQL: In-context learning for text-to-SQL with automatically-generated chain-of-thought. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3501–3532, Singapore. Association for Computational Linguistics.

A Benchmark Annotation Guidelines

Specifically, user instructions generally should concise (not exceeding 200 words) and directly relevant to the schema and data of the database. To simulate real-world scenarios, instructions should vary in the description style, ranging from casual chats to formal requests. To pair with each user instruction, the annotators then retrieve a set of real texts online (such as wiki articles or news reports). Each set contains no more than 10 documents with 3000 words. The preferred documents are relevant to the specific databases and user instructions, and

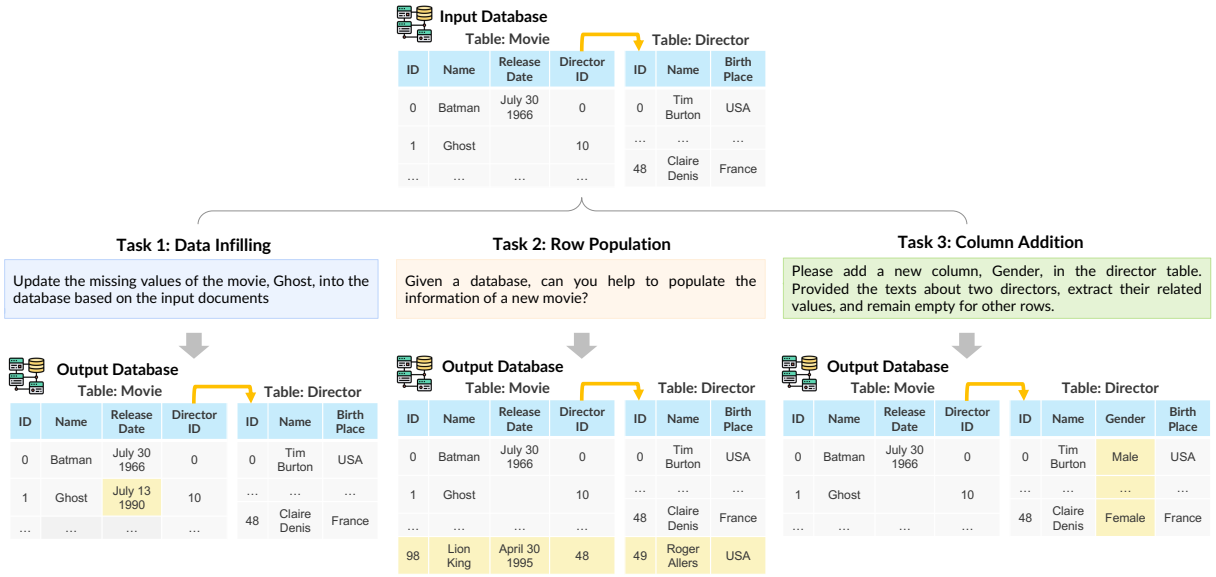

Figure 7: Examples of three task types including data infilling, row population, and column addition.

Statistics	Task Types Wiki	Task Types Bird	Total
User Instruction			
# Avg. Words	29.7	37.5	31.1
Source Text			
# Avg. Documents	2.8	2.4	2.7
# Avg. Words	2,123.8	325.0	1786.5
Database			
# Databases	191	12	203
# Avg. Tables	1.0	9.1	2.5
# Avg. Rows	116.3	297K	56K
# Avg. Columns	5.7	55.3	15.0
ΔValues	6.2	17.7	8.4
Overall			
# Domains	42	9	45
# Easy	81	0	81
# Medium	82	0	82
# Hard	32	45	77
# Instances	195	45	240

Table 6: Statistics of our dataset divided by database source. "Wiki" and "Bird" correspond to different database sources. "#" indicates the count, "Avg." stands for the average value per instance, and "ΔValues" represents the number of value changes in the database following the completion of integration.

provide suitable data for extraction and population. The next core step is updating the databases by manually extracting values from texts according to the user instructions. The newly-added values should strictly keep to all constraints or format requirements of the databases. After that, the annotators map the databases and corresponding instructions to various domains, annotated according

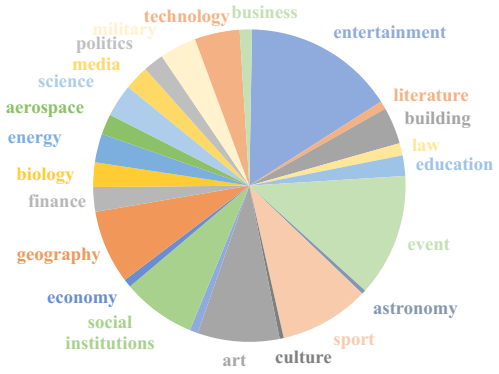

Figure 8: Domain distribution of our dataset.

to Wikipedia's taxonomy[2]. This ensures that our dataset tests the generalization ability of the proposed framework across different fields and types of information.

Throughout the annotation process, we engage three Computer Science PhD students, each with research backgrounds in information extraction, to carry out the data annotation tasks. Every data instance is initially annotated by one of these annotators and subsequently reviewed by another. The annotator and reviewer discuss for any necessary adjustments until a consensus is reached on the annotation. After finalizing the annotation, all annotators convene to assign a difficulty level—easy, medium, or hard—to each data instance. This categorization is based on multi-facet criteria, including the complexity of the database schema, the length

[2] https://en.wikipedia.org/wiki/List_of_lists_of_lists

of the source texts, and the number of values required to be populated.

B Tool Description

The tools can be categorized into two types:

- **Information Extraction**: We include tools corresponding to standard IE tasks including (1) Named Entity Recognition, (2) Relation Extraction, (3) Attribute Extraction, and (4) Text Classification to pinpoint the entities, their attributes, relations, and categories. They set the foundation for a structured information framework necessary for database population.
- **Database Integration**: To materialize database updates, we employ three Database Integration Functions: (5) Data Normalization then adjusts the format of the extracted information to meet database requirements, while (6) Entity Linking connects identified entities with existing entries in the database. (7) Data Infilling fills in missing values by linking extracted entities with their corresponding database entries. (8) Row Population involves adding new rows that adhere to the database schema and constraints. (9) Column Addition introduces new columns, linking extracted entities to existing rows and populating these new columns with relevant values.

C Implementation of OPAL

The planner and observer agents in our OPAL framework are powered by the GPT4 language model `gpt-4-1106-preview`. The maximum number for the Planner to revise plans is 10. The whole process can restart for 2 times at most after failure. Specifically, the planner utilizes code from three different task types as demonstrations. The analyzer then verifies the generated code from perspectives of syntax, logic, and integrity, immediately returning any detected errors to the planner. The observer generates a summary for each table in the database to ensure the quality of observation; it selects 20 pieces of data as IE demonstrations and generate a piece of mock data for the specific task type, which may include values from multiple tables. For the tool library, we emulate the models with GPT for named entity recognition, relation extraction, attribute extraction, text classification and data normalization. They utilizes the demonstrations produced by the observer to achieve in-context learning. Table 19 - 22 shows the prompts for two agents (the planner and the observer), and four tools (named entity recognition, relation extraction, attribute extraction, text classification and data normalization). In addition, we adopt an existing entity linking model GENRE (De Cao et al., 2022). We transforms the extracted results into sentences using a manually designed template, to calculate the probability of linking with sentences transformed from the database's data entries.

D Extra Error Analysis

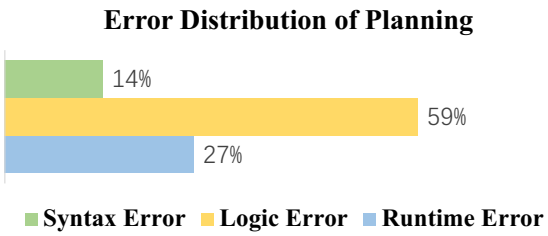

Figure 9: Error distribution of the planning step.

In the planning stage, we identified three main error subtypes: syntax errors, logic errors, and runtime errors. Syntax errors are often fixable through multiple revisions, but with more than three tables in a database, the complexity increases, and not all syntax errors can be resolved quickly. Logic errors usually stem from using incorrect tools (e.g., choosing attribute extraction over text classification for movie genres) or neglecting the interdependencies between tables (like missing foreign key relationships). Runtime errors typically occur due to tool misconfiguration (such as expecting a list instead of a string for input) or mismatches between the extracted data and the database schema (like overlooking necessary column values).

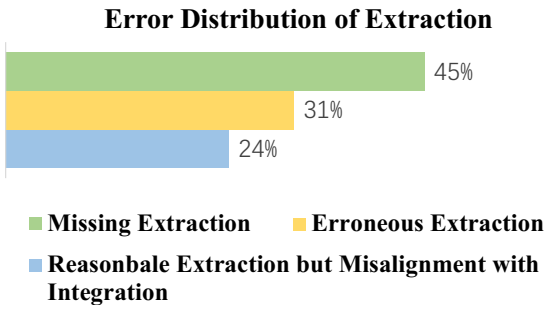

Figure 10: Error distribution of the extraction step.

In the extraction phase, the typical error subtypes are Missing Extraction, Erroneous Extrac-

tion, and Reasonable Extraction but Misaligned with Integration Requirements. Missing Extraction often occurs in long documents where IE tools may overlook sections, especially where named entity recognition models fail to capture all possible entities, leading to errors. Erroneous Extraction arises in cases with closely related information, causing confusion, such as mixing up movie premiere and release dates. The third error type involves extractions that are accurate but don't meet the specific needs for integration, like varying granularity levels in location data for an earthquake's epicenter. Demonstrations via data entry have relieved this, but there's still room for further improvement.

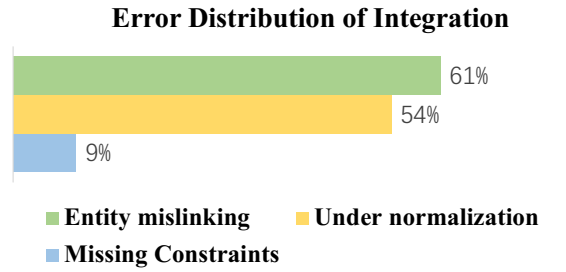

Figure 11: Error distribution of the integration step.

For integration, the specific error subtypes are Entity Mislinking, Under-normalization, and Data Integrity Issues. Entity Mislinking, a major bottleneck for data infilling and column addition, involves linking extracted data to the appropriate database entry, a task that becomes challenging with large databases or when database entries lack sufficient information. Under-normalization typically deals with inconsistencies in data formats, such as varying date formats, where normalization efforts might miss specific values or sacrifice efficiency for accuracy. Data Integrity Issues often arise in databases with multiple tables, where problems like missing foreign key constraints lead to the insertion of unrelated values.

E Case Study

In Figure 12, we show the code for case study in Section 5.3. Meanwhile, we also present more cases for three task types, including the updated databases and the corresponding generated codes, in Figure 13-18.

```
# Step 1. Determine the task type
task_type = 'row population'


# Step 2. Perform operations on the movie table
movie_titles = Named_Entity_Recognition(text, "movie")

all_movie_data = []
for _id, movie_title in enumerate(movie_titles):
    movie_attributes = Attribute_Extraction(text, movie_title, ["MPAA Rating", "Budget", "Gross",
"Release Date", "Runtime", "Rating", "Rating Count", "Summary"])
    movie_attributes['Genre'] = Text_Classification(text, ["Romance", "Comedy", "Crime", "War",
"Drama", "Family", "Action"])
    movie_attributes['Title'] = movie_title
    movie_attributes['MovieID'] = int(database['movie'][-1]['MovieID']) + _id + 1
    all_movie_data.append(movie_attributes)

normalized_movie_data = Data_Normalization(all_movie_data, database, table_name='movie')
Populate_Row(normalized_movie_data, database, table_name='movie')


# Step 3. Perform operations on the actor table
actor_names = Named_Entity_Recognition(text, "actor")
actor_ids = {}

all_actor_data = []
for _id, actor_name in enumerate(actor_names):
    actor_attributes = Attribute_Extraction(text, actor_name, ["Date of Birth", "Birth City", "Birth
Country", "Height (Inches)", "Biography", "Gender", "Ethnicity", "NetWorth"])
    actor_attributes['Name'] = actor_name
    actor_attributes['ActorID'] = int(database['actor'][-1]['ActorID']) + _id + 1
    actor_ids[actor_name] = actor_attributes['ActorID']
    all_actor_data.append(actor_attributes)

normalized_actor_data = Data_Normalization(all_actor_data, database, table_name='actor')
Populate_Row(normalized_actor_data, database, table_name='actor')


# Step 4. Perform operations on the characters table
characters_names = Named_Entity_Recognition(text, "characters")

all_character_data = []
for _id, character_name in enumerate(characters_names):
    character_attributes = Attribute_Extraction(text, character_name, ["creditOrder", "pay",
"screentime"])
    character_attributes['Character Name'] = character_name
    # Identify associated ActorID and MovieID for each character based on existing actor and movie data
    character_attributes['ActorID'] = actor_ids[character_name] if character_name in actor_ids else
None
    character_attributes['MovieID'] = movie_attributes['MovieID']
    all_character_data.append(character_attributes)

normalized_character_data = Data_Normalization(all_character_data, database, table_name='characters')
Populate_Row(normalized_character_data, database, table_name='characters')
```

Figure 12: The complete of the generated code for the example in Figure 6. By checking the part corresponding to the `Character` table, interestingly, we observe that the model is aware of the table dependencies but the wrong key is used to find the ActorID (should be actor name instead of character name).

```
# a. Determine the task type based on the user instruction
task_type = 'data infilling'
# List the targeted attributes explicitly mentioned by the user instruction. Otherwise, infer based on
database schema.
target_attributes = ["Deaths", "Injuries"]

# b. Extract earthquake information
# (1) Extract earthquake event name, which is the table name or the primary key, using
Named_Entity_Recognition
event_names = Named_Entity_Recognition(text, "earthquake")

all_event_data = []
for _id, event_name in enumerate(event_names):
    # (2) Extract the attributes using Attribute_Extraction
    event_attributes = Attribute_Extraction(text, event_name, target_attributes)
    event_attributes['Event'] = event_name
    all_event_data.append(event_attributes)

# (3) Normalize the extracted attributes to fit the earthquake table schema using Data_Normalization
normalized_event_data = Data_Normalization(all_event_data, database, table_name='earthquake')
print(normalized_event_data)

# (4) Populate the new rows into the database
Infill_Data(normalized_event_data, database, table_name='earthquake')
```

Figure 13: Example code for the data infilling task.

User

I am maintaining a database of the largest earthquakes by year. Given the latest document of the peru earthquake, please update the numbers of deaths and injuries in this disaster.

Text

An earthquake measuring Mw 8.0 struck Peru and the surrounding areas on 26 May 2019 at 02:41 local time. It had a maximum perceived intensity of VII (Very strong) on the Modified Mercalli intensity scale in the towns of Yurimaguas and Lagunas. Two people died and a further 30 were injured. It was the strongest earthquake in 2019 by magnitude.\n\nTectonic setting\nPeru lies above the destructive plate boundary where the Nazca Plate subducts beneath the South American Plate. The plates converge at a rate of 70 mm per year. The country has been affected by many large megathrust earthquakes caused by slip along the plate interface, such as the 1868 Arica earthquake.

	Year	Magnitude	Location	Depth (km)	MMI	Deaths	Injuries	Event	Date
57	2009	8.1	Samoa, Offshore	18.0	VII	189	7	2009 Samoa earthquake and tsunami	September 29
58	2010	8.8	Chile, Concepción	22.9	IX	550	12,000	2010 Chile earthquake	February 27
59	2011	9.1	Japan, Honshu	29.0	IX	19,747	6,000	2011 Tōhoku earthquake and tsunami	March 11
60	2012	8.6	Indonesia, Indian Ocean	20.0	VII	10	12	2012 Indian Ocean earthquakes	April 11
61	2013	8.3	Russia, Sea of Okhotsk	598.1	VI	0	0	2013 Okhotsk Sea earthquake	May 24
62	2014	8.2	Chile, Iquique	25.0	VIII	6	9	2014 Iquique earthquake	April 1
63	2015	8.3	Chile, Coquimbo	22.4	IX	21	34	2015 Illapel earthquake	September 16
64	2016	7.9	Papua New Guinea, New Ireland	94.5	VII	0	0	2016 Solomon Islands ...	December 17
65	2017	8.2	Mexico, Chiapas	47.4	IX	98	250	2017 Chiapas earthquake	September 7
66	2018	8.2	Fiji, Offshore	600.0	V	0	0	2018 Fiji earthquake	August 19
67	2019	8.0	Peru, Loreto	122.6	VIII	2	30	2019 Peru earthquake	May 26
68	2020	7.8	United States, Alaska Peninsula offshore	28.0	VII	0	0	July 2020 Alaska Peninsula earthquake	July 22
69	2021	8.2	United States, Alaska Peninsula offshore	32.2	VII	0	0	2021 Chignik earthquake	July 28
70	2022	7.6	Papua New Guinea, Morobe	61.4	VIII	21	42	2022 Papua New Guinea earthquake	September 11
71	2022	7.6	Mexico, Michoacán	25.5	VIII	2	35	2022 Michoacán earthquake	September 19

Figure 14: Case study for the data infilling task. The newly added values are framed in orange.

```
# a. Determine the task type based on the user instruction
task_type = 'row population'

# b. Extract flood event information
# (1) Extract flood event name, which is the table name or the primary key, using
Named_Entity_Recognition
flood_event_names = Named_Entity_Recognition(text, "flood")

all_flood_data = []
for _id, flood_event_name in enumerate(flood_event_names):
    # (2) Extract attributes for the flood event using Attribute_Extraction
    flood_attributes = Attribute_Extraction(text, flood_event_name, ["Death count", "Location",
"Year"])
    flood_attributes['Event'] = flood_event_name

    # (3) Normalize the extracted attributes to fit the flood table schema using Data_Normalization
    normalized_flood_data = Data_Normalization([flood_attributes], database, table_name='flood')

    all_flood_data.append(normalized_flood_data[0])

# (4) Populate the new rows into the database
Populate_Row(all_flood_data, database, table_name='flood')
```

Figure 15: Example code for the row population task.

User

Given a document about one of the deadliest floods, can you add its related information into the database.

Text

The 1935 Yangtze flood struck China during a decade of flooding, famine and social turmoil. It is considered to be the fifth deadliest flood in recorded history, with a death toll of 145,000 and displacement of millions. As a result of the flood, millions of survivers were faced with hardship due to displacement, injury, loss of property as well as food shortages and famine.Four years earlier in 1931, after three years of drought, both the Yangtze and Yellow Rivers experienced significant flooding. Known as the 1931 China Floods, they were considered to be the worst non-pandemic disaster of the century because of the millions of deaths they led to indirectly.

Death count	Event	Location	Year
85+	January 2010 Rio de Janeiro floods and mudslides	Rio de Janeiro, Brazil	2010
81+	Valencia flood	Valencia, Spain	1957
81	Holmfirth floods—Bilberry Reservoir dam failure	United Kingdom	1852
81	2021 Turkey floods	Turkey	2021
80–100	1852 Gundagai flood	Australia	1852
80+	2014 Southeast Europe floods	Serbia, Bosnia and Herzegovina, Croatia	2014
80	1988 Sudan floods	Sudan	1988
80	Failure of Laurel Run Dam and flash flooding, Johnstown, ...	United States	1977
78	Austin Dam failure	United States	1911
77+	2019 Iran floods	Iran	2019
73	1993 Kagoshima Heavy Rain, mudslide and debris flow	Japan	1993
72+	Nigeria floods	Nigeria	2012
72	Gudbrandsdalen flood and landslides	Norway	1789
69	2005 levee failures in Greater New Orleans	United States	2005
68	2019 South Sulawesi floods	Indonesia	2019
67	2003 Santa Fe flood	Argentina	2003
66	2020 Jakarta floods	Indonesia	2020
62	2021 Niger floods	Niger	2021
61	Clermont and Peak Downs flood	Australia	1916
145000	1935 Yangtze flood	China	1935

Figure 16: Case study for the row population task. The newly added values are framed in orange.

```
# a. Determine the task type based on the instruction
task_type = 'column addition'

# b. Extract leader information
# (1) Extract leader's name, which is the table name or the primary key, using Named_Entity_Recognition
leader_names = Named_Entity_Recognition(text, "state_leader")

# (2) Extract attributes for the leader using Attribute_Extraction. Here the attributes are the new
columns according to the user instruction.
all_leader_data = []
new_columns = ['Born']
for _id, leader_name in enumerate(leader_names):
    leader_attributes = Attribute_Extraction(text, leader_name, new_columns)
    leader_attributes['Name'] = leader_name
    all_leader_data.append(leader_attributes)

# (4) Normalize the extracted attributes to fit the state_leader table schema using Data_Normalization
# for column addition, the data format for the new column should be defined, including the data
description, the data type and any special requirement if needed.
data_format = {"Born": "date of birth of a leader, Text, DD Month YYYY"}
normalized_leader_data = Data_Normalization(all_leader_data, database, table_name='state_leader',
data_format=data_format)
print(normalized_leader_data)

# (5) Populate the new columns into the database
Add_Column(normalized_leader_data, database, table_name='state_leader', new_columns=new_columns)
```

Figure 17: Example code for the column addition task.

User

Given five documents about five state leaders, can you add a new column of the dates they born on into the database? This column should be named as \"Born\" in the format of DD Month YYYY (such as 27 September 1940).

Text

Seyyed Ali Hosseini Khamenei (Persian: سید علی حسینی خامنه‌ای, romanized: Ali Hoseyni Xāmene'i, pronounced [ʔæ'li: hosej'ni: xɒ:mene'ʔi:] ; born 19 April 1939) is an Iranian Twelver Shia marja' and politician who has been the second supreme leader of Iran since 1989. He previously served as third president of Iran from 1981 to 1989. Khamenei is the longest-serving head of state in the Middle East, as well as the second-longest-serving Iranian leader of the last century, after Shah Mohammad Reza Pahlavi....

Rank	Name	Position	Assumed office	Age	Born
1	Paul Biya	President of Cameroon	1982	90 years, 313 days	
2	Mahmoud Abbas	President of the Palestinian National...	2005	88 years, 38 days	
3	Salman bin Abdulaziz Al Saud	King of Saudi Arabia	2015	87 years, 357 days	
4	Francis	Pope of the Holy See, Sovereign of ...	2013	87 years, 6 days	
5	Harald V	King of Norway	1991	86 years, 305 days	
6	Ali Khamenei	Supreme Leader of Iran	1989	84 years, 248 days	19 April 1939
7	Margrethe II	Queen of Denmark	1972	83 years, 251 days	16 April 1940
8	Mishal Al-Ahmad Al-Jaber Al-Sabah	Emir of Kuwait	2023	83 years, 87 days	27 September 1940
9	Michael D. Higgins	President of Ireland	2011	82 years, 249 days	18 April 1941
10	Sergio Mattarella	President of Italy	2015	82 years, 153 days	23 July 1941

Figure 18: Case study for the column addition task. The newly added values are framed in orange.

I am working on developing an automatic system for dataset population. Specifically, given a database schema, a user instruction, and the background text, the system aims to populate database with the desired information extracted from the text according to the user instruction, including three task types, data infilling, row population and column addition.

Currently, you need to act as a code generation model, that considering the user instruction and the database schema (especially the interdependency between the tables), determines the sequence of the modules and output the executive python code which calls for these modules sequentially to solve the dataset population task.

The modules are defined as follows:

- Named_Entity_Recognition(text: str, type: str) -> list
- For the given background text and the entity type, this module is to extract all the entities of the specified type from the text.

- Relation_Extraction(text: str, head_e:list, relation: str) -> list
- Given a text, a list of entities and a relation type, this module is to extract all the tail entities for each head entity considering this relation type from the provided text.

- Attribute_Extraction(text: str, entity: str, attribute_list: list) -> dict
- Given the background text, an entity, and a list of attribute names, this module is to extract the attribute values for each attribute name from the provided text.

- Text_Classification(text: str, label_list: list) -> str
- Given a text and a list of labels, this module is to classify the text into one label.

- Data_Normalization(data_entries: list, database: dict, table_name: str, data_format: dict=None) -> list
- Given a list of data entries whose keys are the column name of the values, and an existing database, transform these data entries to match the schema of one table in the database. The function output is the normalized data entries.

- Entity_Linking(data_entries: list, database: dict, table_name: str) -> list
- Given an existing database, a table name, and a list of data entries, this module is to link these entities with some existing rows in the database. Finally, this function outputs a list of data index.

- Infill_Data(data_entries: list, database: dict, table_name: str) -> dict
- Given some data entries, an existing database, and a table name in this database, this module is to infill missing values in one table of database with the data entries. The output is the updated database.

- Populate_Row(data_entries: list, database: dict, table_name: str) -> dict
- Given some data entries, an existing database, and a table name in this database, this module is to populate the table with the data entries by add new rows. The output is the updated database.

- Add_Column(data_entries: list, database: dict, table_name: str, new_columns: list) -> dict
- Given some data entries, an existing database, a table name, and a list of new column names in this database, this module is to add new columns into a table of the database with the data entries. The output is the updated database.

Below are three examples of different task types using the same database:

[Example 1]
[Example 2]
[Example 3]

Using the format provided above, generate the plan for database population given the inputs. The code can directly use four parameters, including instruction, text, database_schema, and database. But don't assume the value for any other parameter. Importantly, don't output any other information but the code.

Figure 19: Prompt for the Planner.

Given a database including its schema and some existing data entries, summarize the database into a brief description for each column in every table. This description should include:
column_name
column_description
data_format
value_description (data unit, special data format, label space for the values)
Recommanded extraction tool (including name entity recognition, relation extraction, attribute extraction, text classification)

For example:

Input:

Database schema:
CREATE TABLE "movie" (
"MovieID" INTEGER,
"Title" TEXT,
"MPAA Rating" TEXT,
"Budget" INTEGER,
"Gross" INTEGER,
"Release Date" TEXT,
"Genre" TEXT,
"Runtime" INTEGER,
"Rating" REAL,
"Rating Count" INTEGER,
"DirectorID" INTEGER,
CONSTRAINT "movie_pk" PRIMARY KEY("MovieID")
FOREIGN KEY("DirectorID") REFERENCES "director"("DirectorID"),
);

Existing data entries:
1 Look Who's Talking PG-13 7500000 296000000 1989-10-12 Romance 93 5.9 73638 1
2 Driving Miss Daisy PG 7500000 145793296 1989-12-13 Comedy 99 7.4 91075 2
3 Turner & Hooch PG 13000000 71079915 1989-07-28 Crime 100 7.2 91415 100
4 Born on the Fourth of July R 14000000 161001698 1989-12-20 War 145 7.2 91415 1000
5 Field of Dreams PG 15000000 84431625 1989-04-21 Drama 107 7.5 101702 1234
6 Uncle Buck PG 15000000 79258538 1989-08-16 Family 100 7.0 77659 1231
7 When Harry Met Sally... R 16000000 92800000 1989-07-21 Romance 96 7.6 180871 129

Output:
MovieID: Unique identifier for each movie. Format: Integer. Should use
Title: Name of the movie. Format: Text. Use name entity recognition tool.
MPAA Rating: Motion picture association of america rating. Content suitability rating. Format: Text. Label space includes G, PG, PG-13, R, NC-17. Use text classification tool.
Budget: Production budget in dollars. Format: Integer. Use attribute extraction tool.
Gross: Box office revenue in dollars. Format: Integer. Use attribute extraction tool.
Release Date: Date of release. Format: Text (yyyy-mm-dd). Use attribute extraction tool.
Runtime: Duration in minutes. Format: Integer. Use attribute extraction tool.
Rating: Audience rating, 0.0 to 10.0 where higher is better. Format: Real. Use attribute extraction tool.
Rating Count: Total number of ratings received. Format: Integer. Use attribute extraction tool.
DirectorID: Identifier for the director, linking to the "director" table. Format: Integer.

Following the example above, generate a brief description for the following database:

Figure 20: Prompt for the Observer.

Named Entity Recognition	For the given entity type and text, extract all the entities of the specified type from the text. Here is an example: Input: Entity Type: product Text: I just bought the new iPhone 13 and MacBook Pro. My friend recommended the Samsung Galaxy S21 and the Amazon Echo Dot. Output: iPhone 13 MacBook Pro Samsung Galaxy S21 Amazon Echo Dot Using the format provided above, extract the entities for the given entity type from the text. Please strictly follow the output format like the given example and do not output any extra words.
Relation Extraction	Given a list of entities and a relation type, your task is to extract all the tail entities for each head entity considering this relation type from the provided text. Here is an example: Input: Text: The song 'Shape of You' by Ed Sheeran was released on January 6, 2017. 'Thinking Out Loud', another hit by him, was released on September 24, 2014. Entity: Shape of You, Thinking Out Loud Relation Type: Releasing date Output: January 6, 2017 September 24, 2014 Using the format provided above, extract the tail entities for each head entity according to the relation type from the text. Please strictly follow the output format like the given example and do not output any extra words.
Attribute Extraction	Given a entity and a list of attribute names, your task is to extract the attribute value for each attribute name from the provided text. If the value of some attribute is not mentioned in the text, please output "None" for this attribute. Here is an example: Input: Text: The song 'Shape of You' by Ed Sheeran was released on January 6, 2017. 'Thinking Out Loud', another hit by him, was released on September 24, 2014. Entity: Shape of You Attribute Names: Releasing date Singer Genre Output: January 6, 2017 Ed Sheeran None Using the format provided above, extract attribute values for the given entity and attribute names from the text. Please strictly follow the output format like the given example and do not output any extra words.

Figure 21: Prompts for the information extraction tools.

Text Classification	Given a text and a list of labels, your task is to classify the text into one label. Here is an example: Input: Text: The song features distorted electric guitars, aggressive drums, and powerful vocals. The lyrics discuss themes of rebellion and personal freedom. Label list: Rock, Pop, Hip Hop, R&B, Jazz, Blues Output: Rock Using the format provided above, classify the following text to one label from the provided list. Please strictly follow the output format like the given example and do not output any extra words.
Data Normalization	Given a few rows of new data, please normalize the data according to some existing rows and data format requirement (if exists) in a database so as to match with database schema. Note that each data value is splited by '; ' in one row. Here is two examples: Example 1 Input: New data: 1939; 7; Erzincan Province in Turkey; 20.34; 27 Dec; two 1968; 5.2; China, Shanghai; 10.3; December 2 2000; 10 Existing data: 1940; 7.7; Romania, Vrancea County; 133.0; November 10; 31 1941; 5.8; Yemen, Razih District; 35.0; January 11; 3 1942; 7.0; Turkey, Erbaa; 10.0; December 20; 9 Output: 1939; 7.0; Turkey, Erzincan Province; 20.3; December 27; 2 1968; 5.2; China, Shanghai; 10.3; December 2; 10 Example 2 Input: New data: December 2 2000 June 10 2004 1889 09 27 Data format requirement: Birth of Data: date of birth of a single, Text, MM-DD-YYYY Output: 12-02-2000 06-10-2024 09-27-1889 Using the format provided above, normaliza a new row given some existing rows as the examples. Please strictly follow the output format like the given example and do not output any extra words.

Figure 22: Prompts for the information extraction tools.