A APPENDIX

A.1 Detailed Annotation Pipeline

We describe the detailed annotation pipeline of ELT-Bench, which consists of following six steps:

Step 1: data sources convertion. To simulate integrating various data sources in a real-world ELT pipeline, we convert the collected data into different formats based on its characteristics and the classifications in Table 1. The original BIRD data is stored in `Sqlite`, while `Fivetran` data is in CSV format. Data is typically stored in multiple formats in real-world scenarios, depending on its intended use. For example, as shown in Table 1, relational databases are commonly used for transactional data storage. The selection of a target format follows these criteria:

(1) We identify the potential data sources based on Table 1.
(2) We select the format that maximizes the source diversity within the instance database if a data source can be represented in multiple formats.
(3) We ensure that the selected format is compatible with `Airbyte`.

Step 2: data source and environment setup. The second step involves setting up the environment for storing data in different formats. As mentioned, we use Docker containers to deploy various data sources. We write the necessary scripts to convert data into multiple formats during environment setup to reflect the diversity of real-world data storage, including table creation and data insertion for `PostgreSQL` and `MongoDB`, `REST API` implementation, and data upload scripts for cloud storage. Furthermore, because the existing `Airbyte` extractor does not support local APIs, we have developed a custom extractor that can retrieve data from a `REST API` running inside a Docker container. We only need to perform all of these setup steps once before running the experiments.

Step 3: configuration annotation. After converting data formats and setting up the environment, we annotate the necessary connection information for data storage platforms and tools, which serve as one of the inputs in ELT-Bench. Notably, the annotated configurations do not strictly match the field names in the `Airbyte` documentation. This setting increases complexity and requires the agent's reasoning capabilities. For example, when extracting data from `MongoDB`, we specify the connection string as follows:

```
mongodb:
  config:
    connection_string: 'mongodb://elt-mongodb:27017/?
        directConnection=true'
```

The agent must determine that `MongoDB` is self-managed and configure `Airbyte` based on the provided configuration:

```
configuration = {
  database_config = {
    self_managed_replica_set = {
      connection_string = "mongodb://eltmongodb:27017/
          directConnection=true"}}}
```

Step 4: data model definition. In this step, we annotate the columns and their corresponding descriptions of data models in each database. Each data model consists of descriptive attributes from dimension tables and quantitative metrics from fact tables, representing specific entities. We categorize the columns in the data model based on the transformations applied:

(1) *Derived columns* come from either direct copies of the columns in source tables or transformations through basic operations (e.g., renaming, concatenation, and mathematical operations).
(2) *Aggregated columns* summarize data from the fact table using aggregation functions such as `SUM`, `AVG`, `COUNT`, `MAX` and `MIN`.
(3) *Categorical columns* classify data into predefined categories based on thresholds or conditions.
Example: The `has_more_than_10_movies_1960_to_1985` is set to 1 if the director has directed at least 10 movies between 1960 to 1985; otherwise set to 0.

```
SELECT director_id, CASE WHEN COUNT(*) > 10 THEN 1
  ELSE 0 END AS has_more_than_10_movies_1960_to_1985
FROM movies
WHERE movie_release_year BETWEEN 1960 AND 1985
GROUP BY director_id;
```

(4) *Ranked Columns* apply ranking functions (e.g., `RANK()`) to establish an order based on specific criteria and extract a value at a particular rank position.
Example: The `highest_average_score_film` column shows the film directed by a director with the highest average score, with ties broken by the ascending order of the movie title.

```
WITH rated_film_ranks AS (
  SELECT director_id, movie_title,
    RANK() OVER (PARTITION BY director_id
      ORDER BY AVG(rating_score) DESC,
      movie_title ASC) AS rated_film_rank
  FROM movie_platform.movies AS T1
  JOIN movie_platform.ratings AS T2
    ON T1.movie_id = T2.movie_id
  GROUP BY director_id, movie_title)
SELECT director_id,
  movie_title AS highest_average_score_film
FROM rated_film_ranks
WHERE rated_film_rank = 1
```

We now describe how we extract these four types of columns for our data models based on the natural language questions of BIRD. While the original questions are typically designed for a specific entity, such as a single movie or person, we extract features for all entities instead. For example, consider the question, "Which film directed by Abbas Kiarostami has the highest average score?" This corresponds to the extracted feature `highest_average_score_film` in the `Directors` data model, which represents the film with the highest average score for each director. After extracting features, we add text descriptions for these columns to help the agent better understand the desired data model and reduce ambiguity.

To make the data transformation stage in ELT-Bench more challenging, we rank the original SQL queries by complexity and prioritize features that involve more SQL components, conditions, and table joins [79]. Each data model typically consists of three derived columns from the source tables and five additional columns (i.e., aggregated columns, categorical columns, and ranked columns) extracted from BIRD questions. Since the databases from `Fivetran` already contain predefined data models, we directly refine these models by removing columns generated by utility functions and those that contain only null values, unless they are required by another data model (a data model may incorporate another data model as an intermediate stage).

Table 6: ELT-Bench evaluation results for all tested agents with open-source LLMs.

Agent Framework	LLM	SRDEL (%)	SRDT (%)	Average Cost ($)	Average Steps
Spider-Agent [36]	DeepSeek-R1	0	0	0.38	18.4
	Llama-3.1-405B-Instruct	0	0	0.39	22.0
	Qwen2.5-Coder-32B-Instruct	0	0	0.50	37.3
SWE-Agent	DeepSeek-R1	0	0	3.16	66.9
	Llama-3.1-405B-Instruct	0	0	2.90	73.9
	Qwen2.5-Coder-32B-Instruct	0	0	0.48	39.1
Augment Agent*[12]	Llama-3.1-405B	0	0	2.80	65.1
OpenHand CodeActAgent[66, 67]	DeepSeek-R1	1%	0	4.05	62.6
	Llama-3.1-405B-Instruct	0	0	1.48	65.2
	Qwen2.5-Coder-32B-Instruct	0	0	0.57	47.2

* Since Augment Agent only supports function calling, we do not run it with DeepSeek-R1 and Qwen2.5-Coder-32B-Instruct.

Step 5: ground-truth annotation. We annotate the ground truth using the configuration file and the defined data models. First, we review the official documentation to understand each configuration field and implement the necessary code accordingly. Next, we annotate the SQL queries used to generate the defined data models in Step 4. To achieve this, we initially validate the existing queries provided by the Bird benchmark and the `Fivetran` repository. We modify those valid queries to conform to our defined data models; otherwise, we write gold queries from scratch.

Step 6: execution-based verification. To ensure the quality and correctness of ELT-Bench, we manually execute and verify each ELT pipeline, thoroughly confirming the accuracy of environment configurations and annotations.

In the first stage, we validate whether `Airbyte` can correctly extract data from diverse sources and load it into the data warehouse based on the provided configurations. Since `Airbyte` is an actively developing project, we encounter several issues:

(1) *Table name case sensitivity in* `PostgreSQL`: `Airbyte` automatically converts table names in `PostgreSQL` to lowercase, which can cause a table not found error if the provided configuration contains table names with uppercase letters.
(2) *Schema detection for API data sources*: `Airbyte`'s automatic schema detection fails when the source data exceeds the maximum allowed string length.

To address these issues, we standardize table names in `PostgreSQL` to only use lowercase letters and manually define schemas for affected API data sources.

In the data transformation stage, we verify our annotated SQL transformation queries. Specifically, for each defined data model, we write ten additional representative testing queries on average and then execute them against both the original source tables and corresponding data models. These testing queries encompass: (1) aggregation queries, (2) limit queries, and (3) queries obtained from Bird corresponding to entity-specific questions.

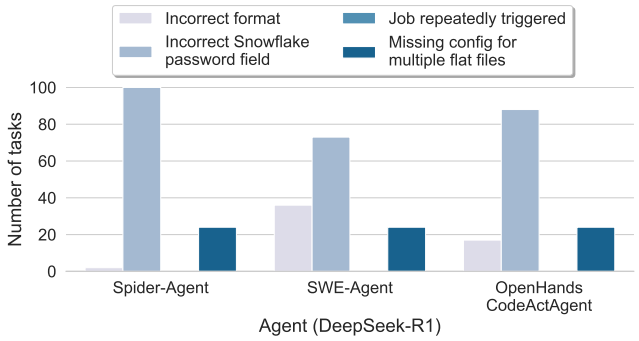
For any discrepancies or mismatches observed during query execution, we carefully review the annotated transformation queries, correct identified errors, and revise the transformation queries accordingly.

A.2 Performance of Open-Sourced LLMs

We evaluated the performance of four agents using three open-source LLMs (DeepSeek-R1, Llama-3.1-405B-Instruct, and Qwen2.5-Coder-32B-Instruct) on the ELT-Bench benchmark. As illustrated in Table 6, only OpenHands CodeActAgent DeepSeek-R1 achieves an SRDEL of 1%. We first analyzed each agent based on the four main issues identified in Section 5.1. Subsequently, we examined the trajectories of these agents and identified a recurring error pattern in Spider-Agent DeepSeek-R1, characterized by the omission of intermediate action steps during task execution. We discuss the details in the following paragraphs.

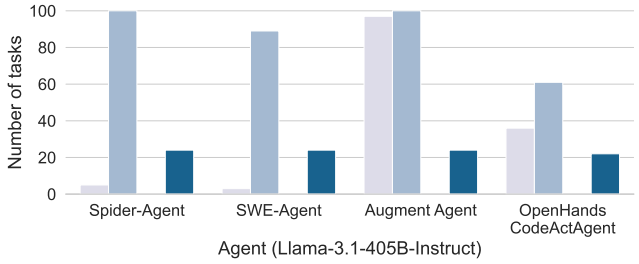
Error analysis of Agents with open-source LLMs across four major issues. We analyzed the low SRDEL exhibited by agents using open-source LLMs, focusing on the four major error types outlined in Section 5.1. We describe them one by one as follows.

(1) *Failure to generate action or code in the required format:* We assessed the agents' ability to generate actions or code in the correct format. As shown in Figure 18, Augment-Agent Llama-3.1-405B-Instruct produced incorrectly formatted code in 97% of tasks, while Spider-Agent Qwen2.5-Coder-32B-Instruct generated incorrectly formatted actions in 71% of tasks.
(2) *Failure to configure the* `Snowflake` *password field:* We analyzed the agents' capability to configure the `Snowflake` password field. Similar to GPT-4o, these open-source LLMs were trained on outdated versions of the `Airbyte Terraform` documentation, which impaired their ability to correctly configure `Airbyte Terraform`. Despite providing the correct documentation in the codebase, the agents failed to set the `Snowflake` password correctly in 61% to 100% of cases.
(3) *Incorrect loaded table size due to the synchronization job being repeatedly triggered:* Because most agents lacked the ability to correctly configure the `Airbyte Terraform`, synchronization jobs could not be triggered successfully. Therefore, we report zero instances of jobs being repeatedly triggered in Figure 18.
(4) *Missing configuration when having multiple flat files:* We further evaluated whether the agents correctly configured all necessary flat files when having multiple flat files. Out of 24 such tasks,



(a) Statistics of agents with DeepSeek-R1.

(b) Statistics of agents with Qwen2.5-Coder-32B-Instruct.



(c) Statistics of agents with Llama-3.1-405B-Instruct.

Figure 18: Stage 1 task failures by agents using open-source LLMs.

the agents failed in 22 to 24 cases. In summary, the agents' low success rate for data extraction and loading, underscore the limitations of current open-source LLMs for building ELT pipelines compared to GPT-4o and Claude-3.5-Sonnet.

Ignoring intermediate steps. We conducted an in-depth analysis of action trajectories of agents and investigated a common error pattern in Spider-Agent DeepSeek-R1. While Spider-Agent DeepSeek-R1 typically generated detailed contextual reasoning in explaining its approach to solve tasks, it frequently generated only a limited sequence of actions, ignoring critical intermediate steps necessary for task completion. A representative example is when the agent recognizes that it should use the `config.yaml` file to configure `Airbyte Terraform`. Rather than accessing the actual contents of the existing `config.yaml` file, Spider-Agent DeepSeek-R1 generates configuration values based on internal knowledge during the reasoning process. Consequently, when editing the `Airbyte Terraform` configuration, the agent populates required fields with these generated values instead of those present in the actual configuration file, resulting in configuration errors. We observed that the agent explicitly referenced the `config.yaml` file in only five tasks. In one of these instances, the agent accessed the file after initializing the `Airbyte Terraform` and encountering an error. In the remaining four cases, the agent consulted `config.yaml` when it required a password to monitor the synchronization job.

We futher compared the action trajectories of Spider-Agent DeepSeek-R1 with those of other agents, focusing on DeepSeek-R1 with different agents and Spider-Agent with different reasoing LLMs. We observed that only Spider-Agent DeepSeek-R1 frequently ignored intermediate steps.

(1) *DeepSeek-R1 with different agents:* We first compared Spider-Agent with SWE-Agent and OpenHands CodeActAgent using DeepSeek-R1. Similar to Spider-Agent, SWE-Agent leverages the ReAct framework [77], but differs by adopting a prompt template that instructs the agent to generate only a single command at each iteration. In contrast, OpenHands CodeActAgent adopts the CodeAct framework [66]. While Spider-Agent DeepSeek-R1 ignored intermediate steps, both SWE-Agent DeepSeek-R1 and OpenHands CodeActAgent DeepSeek-R1 solved tasks in a step-by-step manner.

(2) *Spider-Agent with different reasoning LLMs:* We compared Spider-Agent DeepSeek-R1 with Spider-Agent Claude-3.7-Sonnet. In contrast to Spider-Agent DeepSeek-R1, Claude-3.7-Sonnet produced more concise and efficient reasoning, and generated more effective actions to solve tasks. As a result, Spider-Agent Claude-3.7-Sonnet achieved a higher overall success rate.

A.3 Additional Ablation Studies

Modifying the prompt improves agent performance. In this section, we investigate the effect of prompt design on agent performance. By analyzing error cases, we discovered that GPT-4o, DeepSeek-R1, Llama-3.1-405B-Instruct, and Qwen2.5-Coder-32B-Instruct rely on outdated internal knowledge about configuring `Airbyte Terraform`. As a result, these agents often fail to use the documentation provided in the project base. To address this issue, we enhance the prompt by explicitly instructing the agents to disregard their internal knowledge and instead rely on the up-to-date documentation included in the project base.

As illustrated in Figure 19, Spider-Agent GPT-4o achieves a 133% relative improvement in SRDEL after the prompt modification, compared to its original performance. In contrast, the SRDEL of Spider-Agent models incorporating open-source LLMs remain unchanged following the same prompt adjustment. These results highlight the crucial impact of advanced prompt engineering on the effectiveness of ELT agents. Future research should investigate additional prompt design strategies to further improve the performance of ELT agents across a range of LLMs.

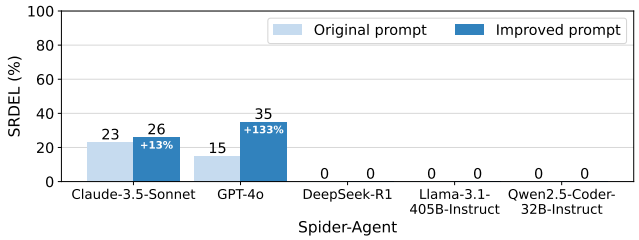


Figure 19: Modifying the prompt improves agent performance. The SRDEL of Spider-Agent GPT-4o increases by 133% in relative terms.

Solving each stage individually does not improve agent performance. To determine whether solving each ELT pipeline stage individually enhances agent performance, we evaluated the best-performing agent, OpenHands CodeActAgent Claude-3.5-Sonnet, by evaluating it separately on the data extraction and loading stage and the data transformation stage. We calculated the overall success rate as the proportion of data models that completed the transformation stage and whose corresponding databases also succeeded in the extraction and loading stage. OpenHands CodeActAgent Claude-3.5-Sonnet achieved an overall success rate of 12.3% when solving each stage individually, compared to 11.3% for the end-to-end pipeline. This small increase does not demonstrate that isolating pipeline stages can significantly improve agent performance.