SEED: Domain-Specific Data Curation With Large Language Models

ABSTRACT

Data curation tasks that prepare data for analytics are critical for turning data into actionable insights. However, due to the diverse requirements of applications in different domains, generic off-the-shelf tools are typically insufficient. As a result, data scientists often have to develop *domain-specific solutions* tailored to both the dataset and the task, e.g. writing domain-specific code or training machine learning models on a sufficient number of annotated examples. This process is notoriously difficult and time-consuming. We present SEED, an *LLM-as-compiler* approach that automatically generates domain-specific data curation solutions via Large Language Models (LLMs). Once the user describes a task, input data, and expected output, the SEED compiler produces a hybrid pipeline that combines LLM querying with more cost-effective alternatives, such as vector-based caching, LLM-generated code, and small models trained on LLM-annotated data. SEED features an optimizer that automatically selects from the four LLM-assisted modules and forms a hybrid execution pipeline that best fits the task at hand. To validate this new, revolutionary approach, we conducted experiments on 9 datasets spanning over 5 data curation tasks. In comparison to solutions that use the LLM on every data record, SEED achieves state-of-the-art or comparable few-shot performance, while significantly reducing the number of LLM calls.

1 INTRODUCTION

Data curation tasks [19, 37] that discover, extract, transform, clean, and integrate data are critical for a wide variety of organizations. Despite significant efforts from the data management community, many sources still report that data scientists still spend over 80% of their time on these tasks [14]. A key reason for this is that applications in different domains have diverse requirements, with no one-size-fits-all solution existing even for single data curation tasks. For example, for the task of data extraction, extracting monetary amounts can be effectively done by a regular expression such as that searches for a dollar sign followed by digits separated by commas and periods, i.e., `"$\d[\d|,|.]*"`, while extracting human names requires a totally different method such as searching for capitalized words near salutations like "Mr." or "Ms.". Because of cases like this, generic off-the-shelf tools are rarely sufficient. Instead, data scientists often have to develop application-specific solutions that are tailored to both the dataset and the problem domain, such as *domain-specific code* (like the regex above) or *machine learning models* trained on a large number of annotated examples to perform these types of tasks. As a result, devising a data curation solution for a particular scenario is time-consuming, with multiple rounds of requirement generation, training data collection, model/algorithm development, and testing with both data scientists and domain experts, and rarely reusable from one deployment to the next. This can be quite costly for enterprises – for example, Citadel employs over 50 data management experts to deliver high-quality cleaned data to their analysts – costing them tens of millions each year.

1.1 Our Approach: SEED

In this work, we propose SEED, an *LLM-as-compiler* approach which, allows users to describe a data curation task via a natural language specification, along with an input data. SEED automatically compiles this specification into an *instance-optimized* solution tailored for the data and application at hand.

The key insight is that LLMs – with their impressive ability to generate code, perform reasoning, understand the semantics of data, and encode common knowledge – will lead to a paradigm shift in data curation research and make it possible to automatically construct data curation solutions *on the fly*. Indeed, prior work has shown that LLMs can be remarkably effective at addressing specific data curation tasks [21, 36, 43]. Unlike these prior works, which rely directly on LLMs for processing every record in a data curation task, SEED instead aims to use LLMs to generate domain-specific *modules* for different data curation tasks, some of which may involve direct invocation of LLMs and some of which are LLM-generated but do not use the LLM once they have been produced.

Specifically, the SEED compiler generates an execution pipeline composed of *code*, *small machine learning models*, as well as direct invocations of the LLM itself on (some) individual data records. In this execution pipeline, modules use LLMs in a variety of ways. For example, code is synthesized by the LLM to provide a domain-specific solution (e.g., a regular expression for extracting monetary amounts) and small models are trained on labels generated by the LLM. If these modules are not confident about the results on some records, SEED will forward them to the *LLM module*, which, although expensive, is often able to perform complex, human-like reasoning tasks on data items. For each request, the LLM module may further employ tools that retrieve relevant information from a database or other user-supplied data to assist the LLM in solving the task. Here, SEED leverages the *reasoning ability* of LLMs to determine on a case-by-case basis what additional information and tools will be helpful in solving the specific task.

In this way, SEED leverages LLMs' synthesis, reasoning, semantics understanding abilities as well as the encoded common knowledge to construct a domain-specific solution. Ideally, users do not need to manually code modules or annotate a large number of training examples. Moreover, unlike prior work on using LLMs for data curation tasks, SEED does not require expensive LLM invocations on every data record, which suffers from scalability, efficiency, and cost issues when handling large datasets.

SEED consists of three key components: **modules**, **infrastructure**, and an **optimizer**.

The **modules** correspond to different types of physical operators which once linked together, offer a domain specific solution to each data curation task. In our current implementation, SEED supports

four types of modules: *CodeGen*, *CacheReuse*, *ModelGen*, and *LLM*. The *CodeGen* module uses LLM-generated code to replace the LLM in processing the data. The *CacheReuse* module reuses the previous exact or similar LLM query results to directly answer the new queries. The *ModelGen* module distills an LLM to a small machine learning model using the LLM as an annotator. The *LLM* module directly invokes an LLM to produce an answer. It also supports RAG-style data access with data curation tools.

SEED's **infrastructure** offers a set of generic functions serving all data curation tasks. It consists of a *scheduler*, *cache storage*, and some basic optimizations to make the modules more efficient and effective. The scheduler controls the dataflow, routes data to suitable modules, and aggregates the results of all modules to produce the final answer. The cache storage, a key element of SEED, caches the input queries and LLM query results and offers interfaces for the modules to efficiently leverage the cached data to directly answer queries, train small models, validate the generated code, and augment LLMs with a RAG style solution. It features some generic optimizations such as *query batching* that saves LLM cost by batching a bunch of LLM queries into one and *tools integration* that automatically invokes the tools supplied by users into the solution.

The **optimizer** constitutes the most interesting component of SEED. Similar to database optimizers, given a specification of a data curation task, SEED 's optimizer automatically decides what modules to use, configures the chosen modules, and orders these modules into an execution pipeline. The objective is to produce a data curation plan that minimizes the execution costs with guaranteed accuracy. Because the SEED optimizer has to take both the execution time and effectiveness into consideration, its search space is much larger than a classical database optimizer where the goal is to minimize the execution time. To efficiently produce a good plan, inspired by the classic cost-based optimizers, SEED collects some statistics and then iteratively constructs the plan by adding modules one by one and reusing the optimal subplans. In addition, we leverage the unique properties of the SEED optimization problem and data curation tasks to further reduce the search space, while not missing the good plans. Moreover, the SEED optimizer dynamically re-optimizes as SEED continuously accumulates more cached data and gradually improves the performance of the cache reuse and small model. In some cases, when SEED determines the small model's performance is good enough with the labelled data it has accumulated from the LLM at a certain point in the execution, the small model might replace the LLM completely.

1.2 Contributions

Our primary contribution is to demonstrate that LLMs enable a new, transformative approach to tackling data curation tasks. Unlike conventional data curation tools, SEED does not aim to provide pre-configured data curation models for specific data domains. Instead, based on the properties of the given dataset as well as user-specific requirements, SEED compiles an *instance-optimized* solution for the given dataset *on the fly*. During this process, users do not have to write any code or perform prompt engineering.

We have built an initial version of SEED to validate the idea. The results are encouraging: with a number of basic optimizations in the infrastructure and optimizer, which we describe below, SEED is able to produce efficient and effective solutions for multiple data curation problems using different modules, demonstrating its potential in practice. In particular, through experiments on 9 datasets spanning various data curation tasks including data imputation, data extraction, data annotation, entity resolution, and data discovery, we show that SEED produces solutions that significantly outperform their generic counterparts, often approaching the performance of solutions trained on thousands of examples. Moreover, in comparison to solutions that use LLMs on every data record, SEED achieves state-of-the-art or comparable few-shot performance, while significantly reducing the number of required LLM calls.

Our technical contributions, mainly focused on the SEED optimizer and effectively generating modules themselves, include:

- **SEED Optimizer.** Inspired by database query optimizers, we have built the SEED optimizer that, given any specific data curation task, automatically produces an execution plan with minimized cost and guaranteed effectiveness. We prove that the plan is optimal under reasonable assumption. To the best of our knowledge, SEED is the first database optimizer style solution that optimizes the use of the LLMs in solving large scale data problem.
- **SEED Modules.** By leveraging the cutting edge techniques in the field as well as inventing new techniques, SEED uses LLMs to effectively synthesize the SEED modules. In particular, to support scenarios with complex logic, the *CodeGen* module in SEED produces an ensemble of code snippets that jointly solve a task. We design an *evolutionary algorithm* to automatically build, filter and aggregate the ensemble of code snippets.
- **Infrastructure.** SEED's infrastructure includes a scheduler that dynamically routes the data along the execution pipeline at instance level. It also offers optimizations such as *query batching* and *tools integration* that are generally applicable to many different data curation tasks.

2 SEED OVERVIEW

We first describe SEED from the user's perspective in Sec. 2.1 and overview its internals in Sec. 2.2.

2.1 SEED Usage

User Configuration. Users interact with SEED via a simple configuration file. Fig. 1 shows the configuration of an entity resolution task. Given a set of products with attributes *"name"*, *"manufacturer"*, and *"price"*, the user seeks to determine whether two products refer to the same item. The configuration mainly involves two parts: (1) describing the task, inputs, and outputs in natural language and (2) (optionally) providing available data and tools. The user first specifies the task name, inputs and outputs. The name can be arbitrary text. Each input also includes its name, data type, and a natural language description, – for example, the description may incorporate the user's domain knowledge about the attribute. The user defines the output in the same way. Optionally, the user may provide additional domain knowledge. This can be done through natural language descriptions, custom tools in the form of APIs for SEED to use, or data files containing examples, either annotated or not.

For common data curation tasks including data extraction, data discovery, entity resolution, data imputation, and data discovery, SEED offers built-in configuration templates. These templates automatically populate the required fields in the configuration file, which

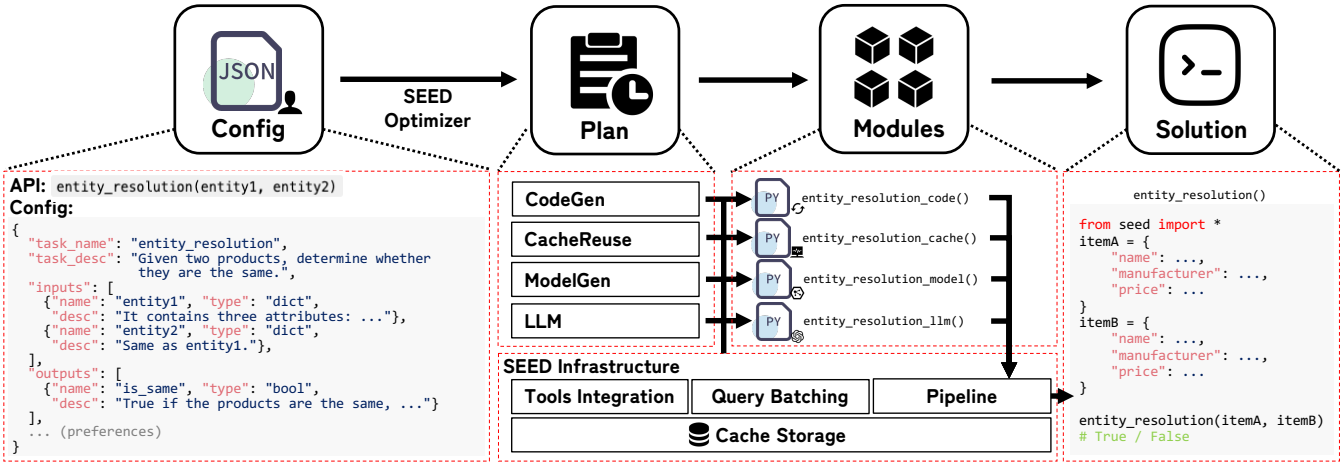

Figure 1: The Architecture of SEED, using an entity resolution task as example.

can be adjusted by users as needed. In the above entity resolution task, the template populates the input field with two entities and the output field with the following information: *"name: is_same; type: bool; description: 1 if the two titles are the same, 0 otherwise."*. After selecting the entity resolution template, the user only needs to optionally provide the hint and some examples.

Compilation & Execution. After the user completes the configuration, SEED compiles it into a deployable program as a functional API conforming to the user's specification, e.g., for the example in Fig. 1, one function, `entity_resolution(entity1, entity2)` → `is_same`, is generated. This function ingests the data in the format of data record pairs and outputs a boolean value indicating if one pair matches or not. As SEED runs, it may periodically regenerate new implementations of this function as, for example, new training data annotated by LLMs becomes available for ModelGen.

2.2 SEED internals

SEED compiles the solution in three steps: (1) using the SEED optimizer to produce a data curation plan that determines what modules to enable; (2) constructing the modules; (3) and linking the modules into an executable pipeline.

(1) Producing a Data Curation Plan. For different tasks and datasets, the SEED optimizer produces different plans that make use of different combinations of modules in different orders. This is because different modules apply to different tasks. Blindly using all the modules on every task tends to be neither effective nor efficient.

As shown in Fig. 1, SEED chooses and applies four modules to each task: CacheReuse, CodeGen, ModelGen, and raw LLM. The CodeGen module is suited to data curation tasks where most cases can be effectively addressed using rule-based methods, such as detecting errors in data that violate domain specific rules or extracting monetary amounts or human names as discussed in Sec. 1. The CacheReuse module performs well in the scenarios where similar or even identical instances frequently arise, while an LLM tends to generate similar or even identical results for these queries. Reusing the cached results thus avoids repeatedly invoking the LLM to answer these queries. The ModelGen module is suitable for data curation tasks that can be modeled as predictive or generative machine learning tasks. For example, entity resolution can be viewed as a classification problem which classifies a pair of objects as matched or unmatched. Such problems can be well-addressed with a machine learning model, trained by examples generated by LLMs or supplied by users. Finally, the LLM module, although universally applicable as advanced but expensive *data analysis tools*, is best suited for complex tasks that require semantic understanding ability. Some tasks might use the LLM modules alone; for example, data discovery – where reasoning whether a table is relevant to a user's question – is an area where LLMs show a clear advantage over simpler methods.

Instead of leaving users to decide what modules to use, the SEED optimizer automatically produces a plan. Moreover, rather than always produce a plan with the highest possible accuracy regardless of the execution cost, SEED can trade-off accuracy to minimize execution cost by for example setting a performance gap tolerance of 10%, meaning that the user accepts a 10% reduction in accuracy versus the maximum attainable for an increase in efficiency. Guided by this objective, the SEED optimizer produces a plan that specifies what modules to use, configures their hyperparameters, and decides the execution order.

(2) Constructing the Modules SEED uses LLMs to help construct the modules on the fly.

For plans that use the CodeGen module, SEED first translates the config file into a task description prompt using a predefined template. This prompt is then sent to an LLM to generate a series of code snippets, which are automatically evaluated and refined (Sec. 4.1). This produces a callable method, e.g., `entity_resolution_code()`. Similarly, for plans with the *ModelGen* module, SEED uses prompt templates to request an LLM to annotate data for use in model training. Then SEED generates a callable method, e.g., `entity_resolution_model()`, which corresponds to a small machine learning model trained on the LLM-generated training data (Sec. 4.2). For the *CacheReuse* module, given a new LLM query, SEED searches for its nearest neighbors in the cache and reuses the results if their similarity is above a threshold.

For the *LLM* module, SEED generates a prompt that connects an LLM to the data access interfaces associated with the given task and composes an iterative process whereby the LLM can selectively use these interfaces to extract data. This also generates a callable

method, e.g., `entity_resolution_llm()`, which implements a RAG style data curation solution (Sec. 5.3).

Note although not the major focus of this paper, we have spent significant effort in SEED to carefully design templates for dynamic prompt composition that perform well in general on these types of data curation tasks, thus eliminating the need for prompt engineering by users. For the prompts templates please see Appendix C of the extended version [2].

(3) Linking and Execute the Modules. Eventually, SEED links all these internal method calls above to a final method, e.g., `entity_resolution(entity1, entity2)` → `is_same`. The generated method sequentially executes the modules in an order specified by the data curation plan over the data records. Each module determines for each record whether it should directly return the result or pass it to the next module. For example, for CodeGen, generated code snippets are explicitly instructed to refrain from answering when faced with uncertain cases, in which case execution will continue with the next module, while CacheReuse and ModelGen refrain from answering when they produce predictions with low confidence (Sec. 4.2). The LLM module, which typically is the last step in the execution pipeline, always answers, using the available data access tools to retrieve relevant information from data to improve response accuracy.

3 SEED OPTIMIZER

Given a data curation task, the SEED optimizer automatically produces an optimized execution plan $\mathcal{P}$ which selects a set of modules M_i to activate, configures their hyperparameters θ_i, and decides their execution order in the pipeline. The objective is to minimize the overall execution cost C while yielding effectiveness $\mathcal{A} \in [0, 1]$ that is close to that of the most effective plan.

Definition 3.1 (**SEED Optimization**). Given an effectiveness gap G that the users can tolerate, the SEED optimizer targets finding an execution plan $\mathcal{P} = [\mathbb{M}(\theta), O(\mathbb{M})]$, that among all possible plans, has the lowest cost C and has an effectiveness $\mathcal{A}(\mathcal{P}) > \mathcal{A}(\mathcal{P}^*) - G$. Here $\mathcal{P}^*$ denotes the plan with highest effectiveness $\mathcal{A}$; $\mathbb{M}(\theta)$ denotes the set of selected modules M_i and their corresponding hyperparameters θ_i; and $O(\mathbb{M})$ denotes the order of these modules.

For example, if the user sets the effectiveness gap to 10%, the SEED optimizer will produce a plan with minimal execution costs subject to the constraint that it is at most 10% worse than a plan with highest possible effectiveness. Tab. 1 shows the hyperparameters θ_i of each module we currently support.

The SEED optimization problem bears similarity to database query optimization at a high level, in that both choose the physical implementation of operators and decides their ordering. However, SEED optimization is different in several respects. First, database query optimization targets minimizing the execution cost of relational queries, while SEED optimization has to take both execution cost and effectiveness into consideration, thus leading to a larger search space. This is because in relational databases, all valid query plans produce the same results, while in SEED, given a data curation task such as entity resolution, different plans that involve AI models might produce different results with different accuracies. Second, in SEED, the optimal SEED plan evolves over time, because the accuracy of modules like CacheReuse and ModelGen tend to keep improving as more results are produced and cached during execution time. Therefore, SEED requires a dynamic re-optimization approach to adapt the plan.

Table 1: SEED Module Hyperparameters

CacheReuse	Activate Distance Threshold
CodeGen	Activate Num Branches Num Preserved Branches Num Iterations
ModelGen	Activate Confidence Threshold
LLM	Activate Examples Sample Mode

3.1 Generic SEED Optimizer

To generate an efficient and effective plan $\mathcal{P}$, inspired by the classic Selinger optimizer, the SEED optimizer uses dynamic programming to reduce the search space. That is, SEED adds modules one by one, keeping track of the selected subplans in a memoization table to avoid recomputing subplans. However, unlike the Selinger optimizer, which only has to store one lowest-cost subplan for each subquery, SEED has to decide what subplans to store based on both effectiveness and cost. Clearly, SEED cannot keep one single plan with either lowest cost and highest effectiveness. However, it cannot afford to store all plans due to the memory and the search costs. To solve this problem, we propose a skyline-based method which stores a subplan *if and only if* it potentially could be a part of the final full plan. More specifically, SEED will never keep a subplan unless it is a *skyline subplan*, where the concept of *skyline subplan* is defined in Def. 3.2.

Definition 3.2 (**Skyline Plan**). A subplan $\mathcal{P}_i$ dominates the other subplan $\mathcal{P}_j$, if $C(\mathcal{P}_i) < C(\mathcal{P}_j)$ and $\mathcal{A}(P_i) > \mathcal{A}(\mathcal{P}_j)$. A subplan $\mathcal{P}_i$ is a skyline subplan if $\mathcal{P}_i$ is **NOT** dominated by any subplan $\mathcal{P}_j$.

Intuitively, a subplan P_i is a skyline plan if there does not exist any plan that is better than it on both effectiveness and cost. If a subplan is not a skyline plan, it will never be a part of the final plan. This is because replacing it with a skyline plan that dominates it will end up with an equal or better plan.

Next, we describe how the SEED optimizer works in more detail. As shown in Alg. 1, the SEED optimizer iteratively attempts to append a new module to the existing subplans (Alg. 1, L4~5), and then discards the dominated subplans (Alg. 1, L9~10). This process continues until no better plans can be found. Unlike exhaustive search, this approach avoids recomputing previous subplans and prunes invalid and dominated subplans as well as their descendants. Finally, SEED optimizer stores all the skyline plans after considering all the modules, and then by default returns the best plan $\mathcal{P}$, that optimizes $\min_{\mathcal{P}} C(\mathcal{P})$ subject to: $\mathcal{A}(\mathcal{P}^*) - \mathcal{A}(\mathcal{P}) \leq G$ (Alg. 1, L14).

The SEED optimizer does not use a greedy algorithm that optimizes the configuration of each module independently. Instead,

Algorithm 1: Generic SEED Optimizer

Input: Task, Gap G
Output: Configuration Θ

$f[0][0] \leftarrow \mathcal{P}_\emptyset$;
for $i \leftarrow 1$ *to* $+\infty$ **do**
  $f[i] \leftarrow f[i-1]$;
  for $\mathcal{P}^{i-1} \in f[i-1]$ **do**
    for $M \in$ *modules and M is not used in* $\mathcal{P}^{i-1}$ **do**
      for $\theta_M \in$ `Grid(M)` **do**
        $\mathcal{P}^i \leftarrow$ `append`$(\mathcal{P}^{i-1}, \theta_M)$;
        $f[i][\mathcal{A}(\mathcal{P}^i)] \leftarrow \arg\min_{\mathcal{P} \in \{f[i][\mathcal{A}(\mathcal{P}^i)], \mathcal{P}^i\}} C(\mathcal{P})$;
  for $\mathcal{P} \in f[i]$ **do**
    $f[i][\mathcal{A}(\mathcal{P})] \leftarrow \arg\min_{\mathcal{P} \in \{f[i][a] | a \geq \mathcal{A}(\mathcal{P})\}} C(\mathcal{P})$;
  if $f[i] = f[i-1]$ **then**
    break;
$\mathcal{P}^* \leftarrow \arg\max_{\mathcal{P} \in f[i]} \mathcal{A}(\mathcal{P})$;
return $\arg\min_{\mathcal{P} \in f[i]} C(\mathcal{P})$ s.t.: $\mathcal{A}(\mathcal{P}^*) - \mathcal{A}(\mathcal{P}') \leq G$;

when attempting to add a new module M_1 into an existing subplan $\mathcal{P} = [\langle M_2(\theta_2), M_3(\theta_3)\rangle]$, the SEED optimizer will reuse this subplan including the parameter configurations of M_2 and M_3 and their relative order, but decide the configuration of M_1 based on the collective performance $\mathcal{A}(\mathcal{P}')$ of the pipeline composed of the modules in the newly formed plan $\mathcal{P}'$. For example, if $\mathcal{P}' = [\langle M_2(\theta_2), M_3(\theta_3), M_1(\theta_1)\rangle]$, the effectiveness $\mathcal{A}(\mathcal{P}')$ is measured by executing the pipeline that applies M_2, M_3, and then M_1 on a validation dataset.

As shown in Fig. 2, the SEED optimizer may start by choosing the LLM module, deciding its best hyperparameter configuration, and storing it as the current optimal subplan. Then the SEED optimizer attempts to append different modules to the single LLM module subplan. It finds that appending CodeGen or ModelGen in front of the LLM module is able to advance the Pareto frontier. Thus, both $[\langle CodeGen, LLM\rangle]$ and $[\langle ModelGen, LLM\rangle]$ are maintained as valid subplans. Then the SEED optimizer may continue to try plugging more modules. Finally, when the plans stop improving, the SEED optimizer returns the most efficient plan from all the valid plans (i.e., plans where $\mathcal{A}(\mathcal{P}^*) - \mathcal{A}(\mathcal{P}) \leq G$) on the Pareto frontier.

Next, we show that as long as the descendants of a dominated subplan are also dominated during search, pruning the dominated subplans and their descendants does not break the optimal substructure property of dynamic programming and thus preserves the optimality of the SEED optimizer.

Theorem 3.1. As long as the descendants of a dominated subplan are also dominated, the SEED optimizer is guaranteed to find the optimal solution $\mathcal{P}^* = \max_{\mathcal{P}} \mathcal{A}(\mathcal{P})$ as well as the most efficient solution $\min_{\mathcal{P}} C(\mathcal{P})$ subject to: $\mathcal{A}(\mathcal{P}^*) - \mathcal{A}(\mathcal{P}) \leq G$.

Proof. We will prove by contradiction. Assume the SEED optimizer eliminated $\mathcal{P}^*$. Then there must exist a configuration $\mathcal{P}$ which is an ancestor of $\mathcal{P}^*$ (i.e., $\mathcal{P}^* = \mathcal{P} \oplus \mathcal{P}_\Delta$ for some $\mathcal{P}_\Delta$ representing the appended modules during the transition from $\mathcal{P}$ to $\mathcal{P}^*$), and $\mathcal{P}$ was eliminated because it is dominated by some other plan $\mathcal{P}'$. Since the descendants of a dominated subplan are also dominated, $\mathcal{P}' \oplus \mathcal{P}_\Delta$ dominates $\mathcal{P}^*$. This contradicts the optimality of $\mathcal{P}^*$. Therefore, by proof of contradiction, the optimal plan $\mathcal{P}^*$ will not be pruned by the SEED optimizer. Since no optimal plan is removed, the SEED optimizer preserves optimality.

Modules & Order	Skyline
⟨LLM⟩	$\mathcal{P}_1$
⟨CodeGen⟩	$\mathcal{P}_2$
...	...
⟨CodeGen, LLM⟩	$\mathcal{P}_3$
⟨ModelGen, LLM⟩	$\mathcal{P}_4$
...	...
⟨CodeGen, CacheReuse, LLM⟩	$\mathcal{P}_5$
⟨CacheReuse, CodeGen, LLM⟩	$\mathcal{P}_6$
...	...

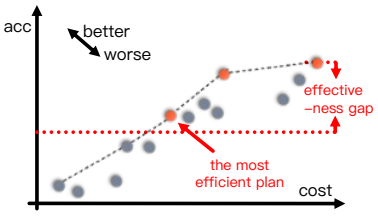

Figure 2: Illustration of the SEED Optimizer.

The same proof applies to $\min_{\mathcal{P}} C(\mathcal{P})$, as the filter $\mathcal{A}(\Theta^*) - \mathcal{A}(\Theta) \leq G$ only takes place at the end of the search. □

Note the assumption that the descendants of a dominated subplan are also dominated typically holds. For example, if the subplan $\mathcal{P} = [\langle M_2(\theta_2), M_3(\theta_3)\rangle]$ composed of modules M_2 and M_3 is dominated by another subplan $\mathcal{P}' = [\langle M_2(\theta_2'), M_3(\theta_3')\rangle]$ which is composed of modules M_2 and M_3 as well, but uses different parameter configurations. Appending a new module M_1 will migrate $\mathcal{P}$ to $[\langle M_2(\theta_2), M_3(\theta_3), M_1(\theta_1)\rangle]$ and $\mathcal{P}'$ to $[\langle M_2(\theta_2'), M_3(\theta_3'), M_1(\theta_1')\rangle]$. It is unlikely that $\mathcal{P}$ will suddenly become better than $\mathcal{P}'$, especially because SEED optimizer will configure module M_1 differently in $\mathcal{P}$ and $\mathcal{P}'$ to maximize their performance boost. Note the Selinger optimizer makes the similar assumption. However, unlike the Selinger optimizer which only keeps one optimal subplan for each subquery, the SEED optimizer potentially could store multiple skyline plans for the same subset of modules. This further reduces the chance that the descendants of a dominated subplan will become the optimal plan at the end.

3.2 Specialized SEED Optimizer

Although it reduces the search space by reusing subplans, the generic SEED optimizer is still very time-consuming. Therefore, we propose to further optimize its efficiency by leveraging the unique properties of the SEED execution pipeline and data curation tasks. In particular, we design a specialized optimizer based on the following optimizations: 1) Prioritizing the ordering of the modules; 2) Incorporating domain-specific constraints to restrict the search space; 3) Accelerating the hyperparameter search for each module; and 4) Cached validation.

Optimization #1: Prioritizing the Module Ordering. When adding a module into an existing subplan and searching for its optimal hyperparameters, the generic SEED optimizer has to enumerate all possible orders, resulting in a large search space. To address this issue, SEED introduces a strategy to prioritize the module ordering before starting to search for the hyperparameters of the new module. It is based on our observation that due to the fallback mechanism of the SEED pipeline, the ordering of the modules often does not matter much to the accuracy. As an example, suppose a pipeline is composed of the ModelGen module and the LLM module, where the LLM module usually has a higher accuracy than the ModelGen module. As long as ModelGen could appropriately fallback to LLM, running ModelGen ahead of LLM would not necessarily lead to an accuracy lower than running LLM first. Moreover, with a limited number of validation examples, in fact it is rather difficult to accurately measure the effectiveness of each possible subplan. Therefore, it seems unpromising to enumerate a larger search space, but only

reap a marginal accuracy gain which could possibly be canceled out by the estimation error.

On the other hand, the ordering indeed matters to the execution cost. Clearly, executing ModelGen first and only fallbacking to LLM on a small number of data records will be much more cost effective than routing all data records to LLM first.

Therefore, in this work, we propose a strategy that efficiently produces an order to minimize the execution cost of the pipeline.

We start with defining a way to quickly estimate the execution cost of a SEED plan.

Definition 3.3 (The Execution Cost of SEED Plan)**.** Each module M_i has a fixed hyperparameter configuration θ_i which determines its execution cost C_i and the fallback probability $p_i \in [0, 1]$ that M_i will route a tuple to the next module in the pipeline. An execution plan $\mathcal{P}$ consists of n modules ordered in the sequence $\langle M_i \rangle_{i=1}^{n}$. The total estimated execution cost of the plan $\mathcal{P}$ can be calculated as:

$$C(\mathcal{P}) = \sum_{i=1}^{n} \left[\left(\prod_{j=1}^{i-1} p_j \right) C_i \right] \tag{1}$$

In Def. 3.3, for each module M_i, its execution cost C_i and fallback ratio p_i are estimated independently. Thus, SEED only has to compute these statistics once and repeatedly uses them in the optimization process. The estimation of C_i and p_i can utilize both labeled validation data and unlabelled data collected during execution, as ground truth is not required for cost and fallback ratio estimation.

$$priority(M_i) = \frac{1 - p_i}{C_i} \tag{2}$$

Once these statistics are obtained, the module ordering algorithm in SEED uses Eq. 2 to calculate the priority for each module and then sorts the modules in descending order of their priorities.

Next, we prove that this simple sorting algorithm indeed minimizes the total execution cost of a plan.

Theorem 3.2. For a set of modules $\mathbb{M}(\theta)$, sorting the modules $M_i \in \mathbb{M}$ in descending order of $priority(M_i)$ will result in a plan $\mathcal{P} = [\mathbb{M}(\theta), O_{priority}(\mathbb{M})]$ that minimizes $C(\mathcal{P})$.

Proof. Given two adjacent modules M_1 and M_2, swapping M_1 and M_2 would not impact the execution cost of the modules before or after them in the pipeline. Thus, consider two adjacent modules M_1: (p_1, C_1) and M_2: (p_2, C_2), denoting the execution cost of the modules before M_1 and M_2 as x, then module M_1 should be in front of module M_2 in the optimal ordering, if and only if:

$$\begin{aligned}
&p_1 \times (p_2 \times x + C_2) + C_1 \le p_2 \times (p_1 \times x + C_1) + C_2 \\
\Leftrightarrow\ &p_1 \times p_2 \times x + p_1 \times C_2 + C_1 \le p_1 \times p_2 \times x + p_2 \times C_1 + C_2 \\
\Leftrightarrow\ &p_1 \times C_2 + C_1 \le p_2 \times C_1 + C_2 \\
\Leftrightarrow\ &\frac{p_1}{C_1} + \frac{1}{C_2} \le \frac{p_2}{C_2} + \frac{1}{C_1} \\
\Leftrightarrow\ &\frac{1 - p_2}{C_2} \le \frac{1 - p_1}{C_1} \\
\Leftrightarrow\ &priority(M_2) \le priority(M_1)
\end{aligned}$$

Then by considering bubble sort, which only swaps the adjacent elements at each step, we can conclude that when all modules are sorted by $(1 - p_i)/C_i$ in descending order, $C(\mathcal{P})$ is minimized. □

Algorithm 2: Specialized SEED Optimizer

Input: Task, Gap G, Init $I \in \{True, False\}$, Beam size B
Output: Configuration $\mathcal{P}$

1 $f[0][0] \leftarrow \mathcal{P}_{\emptyset}$;
2 **for** $i \in [LLM, CodeGen, ModelGen, CacheReuse]$ **do**
3 **if** $f[i]$ *is cached* **then**
4 $f[i] \leftarrow$ `load_cache()`; continue;
5 $f[i] \leftarrow f[i^{prev}]$;
6 **for** $\mathcal{P}^{prev} \in f[i^{prev}]$ **do**
7 **for** $\theta_i \in$ `Default/Grid/Ternary`(M_i) **do**
8 $\mathcal{P}^i \leftarrow$ `sort_by_priority(append(`$\mathcal{P}^{prev}, \theta_M$`))`;
9 $f[i][\mathcal{A}(\mathcal{P}^i)] \leftarrow \arg\min_{\mathcal{P} \in \{f[i][\mathcal{A}(\mathcal{P}^i)], \mathcal{P}^i\}} C(\mathcal{P})$;
10 $\mathcal{P}^* \leftarrow \arg\max_{\mathcal{P} \in f[i]} \mathcal{A}(\mathcal{P})$;
11 **for** $\mathcal{P} \in f[i]$ *s.t.:* $\mathcal{A}(\mathcal{P}^*) - \mathcal{A}(\mathcal{P}') > G$ **do**
12 $f[i]$`.delete(`$\mathcal{P}$`)`;
13 **for** $\mathcal{P} \in f[i]$ **do**
14 $f[i][\mathcal{A}(\mathcal{P})] \leftarrow \arg\min_{\mathcal{P} \in \{f[i][a] | a \ge \mathcal{A}(\mathcal{P})\}} C(\mathcal{P})$;
15 **if** $|f[i]| > B$ **then**
16 $f[i] \leftarrow TopB_{\mathcal{A}}(f[i])$;
17 **if** $i = LLM$ *or* $i = CodeGen$ **then**
18 `save_cache(`$f[i]$`)`;
19 return $\arg\min_{\mathcal{P} \in f[i]} C(\mathcal{P})$;

This insight allows the SEED optimizer to focus on the configuration of the modules rather than their ordering, e.g. determining if it should activate a module or not and selecting its hyperparameters. That is, it adds the modules into the plan one at a time and estimates the cost and effectiveness of this subplan based on the execution order computed with Theorem 3.2, and decides if this subplan should be kept based on the skyline rule described in Sec. 3.1. This is equivalent to directly inserting a new module in the appropriate position based on its priority, without enumerating all possible orders. Note that during optimization the order that considers the modules does not have to comply to their actual execution order determined above. In practice, by optimizing modules in descending order of their estimated effectiveness, SEED optimizer is able to find strong subplans at the early iterations. This yields a tighter Pareto frontier earlier in the process (Alg. 2, L11~12). In this way, the SEED optimizer is able to prune the suboptimal plans more effectively as the optimization proceeds.

Optimization #2: Task-specific Constraints. Secondly, to further reduce the search space, the SEED optimizer implements a set of predefined rules tailored for data curation tasks as well as common sub-types of data curation tasks, like data imputation or data extraction. For example, the hyperaparameters for code generation are typically agnostic to the specific data curation problem, suggesting that SEED could reduce the search space by pre-defining a set of default configurations (Alg. 2, L7) for the CodeGen module and pick one from them during optimization. In practice, SEED restricts CodeGen configuration to three options: 1) not activating CodeGen, 2) using a single code snippet, or 3) using an ensemble with a fixed number of branches. As another example, for data discovery tasks where LLMs show clear advantage over other Modules in effectiveness, SEED disables the CodeGen, CacheReuse and ModelGen modules by default, leaving only the LLM module in play.

Optimization #3: Improved Hyperparameter Search. Thirdly, to configure the hyperparameters within a specific module, various approximated search approaches can be incorporated to speed up the search. For example, for continuous parameters, although a large

step-size grid search is often adequate in most cases, when a more fine-grained search is desirable, SEED can employ ternary search [1] to effectively reduce the search space (see Alg. 2, L7). Another case is when the number of configurations on the Pareto frontier become excessive, a beam search-like approach can be employed to retain only the configurations with the best performance (Alg. 2, L15~16). This allows SEED to trade off the optimality of the optimizer for a further restriction on the number of configurations to search.

Optimization #4: Cached Validation. As mentioned above, given a plan, SEED evaluates its effectiveness $\mathcal{A}$ by executing the plan on a validation dataset. However, evaluating all plans in this way can be extremely costly. This is particularly true for the LLM module, as repeated issuing LLM calls would be prohibitively expensive. To overcome this challenge, we develop a caching mechanism. It caches all input-output pairs for each module $M_i(\theta_i)$. Consequently, if the same $M_i(\theta_i)$ appears in multiple plans, the cached results will be reused. This approach significantly reduces the plan evaluation cost of the SEED optimizer.

3.3 Dynamic Optimization

As mentioned above, instead of generating one single static plan, the SEED optimizer dynamically re-optimizes in response to the changes on the performance of the modules. After initial deployment, SEED will accumulate more data in cache storage. This will benefit the CacheReuse module as well as the ModelGen module through retraining with new data. Specifically, in the beginning when no labelled data is available, the initial plan will only consider the LLM and CodeGen modules. However, after SEED accumulates a certain amount of new data and improves the performance of the CacheReuse and ModelGen modules remarkably, SEED will trigger a re-optimization to regenerate the plan, thus continuously improving the quality of the execution plan.

Rather than re-running the optimization from scratch, the SEED optimizer leverages the results from the previous optimization rounds to reduce search time. By caching and reusing the prior configurations where possible (Alg. 1), it updates the plans more efficiently.

Firstly, because the LLM module and the CodeGen module typically do not benefit much from the accumulated data, SEED only needs to compile these two modules once. Therefore, the SEED optimizer is able to directly reuse their existing configurations during subsequent optimization (Alg. 2, L17~18).

Secondly, during the execution time, assuming the data distribution only changes gradually over short periods, any required modifications to the optimal plan are also likely to be minor. Rather than performing a full grid or ternary search across the entire hyperparameter space of CacheReuse and ModelGen, the optimizer restricts the search space of these continuous hyperparameters to a narrow range centered around the values from the previous optimal plan. This reduced search space exploits the incremental nature of distribution shifts, expediting the optimization process.

Putting It All Together. After incorporating all the optimizations in Sec. 3.2 and Sec. 3.3, the final SEED optimizer (Alg. 2) starts by jointly optimizing the LLM module and CodeGen module and keeps a Pareto frontier. This Pareto frontier is computed once during the initial optimization process and reused for further re-optimizations (Algorithm 2, L17~18). Then during the execution, the CacheReuse and ModelGen modules are periodically re-evaluated, incorporating improvements such as approximate search techniques and heuristics derived from domain knowledge and the previous optimization rounds. This dynamic process allows the SEED optimizer to efficiently generate plans with guaranteed performance.

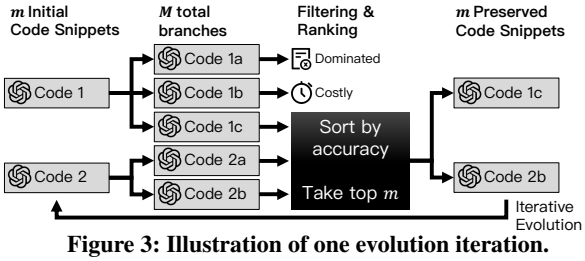

Figure 3: Illustration of one evolution iteration.

4 SEED MODULES

SEED supports three types of modules that use LLMs in various ways: CacheReuse, CodeGen, and ModelGen. Although each module draws inspiration from the existing techniques such as reusing the cached LLM outputs [41], SEED is the first system that leverages these techniques to produce an execution pipeline instance optimized for a data curation task. Moreover, we propose new techniques to make the *CodeGen* and *ModelGen* modules better serve data curation tasks, which are introduced in Sec. 4.1 and Sec. 4.2.

4.1 CodeGen Module

Although using LLMs to generate code has attracted a lot of attention [10, 13, 16, 22, 24], LLMs still struggle to generate large-scale applications or programs requiring complex logic. However, some data curation tasks may require implementing complex logic corresponding to multiple combined rules. For instance, data extraction that extracts multiple types of targeted information from unstructured data (e.g., scraping data records from HTML files) is difficult to accomplish with a single code snippet.

We propose *code ensemble with evolution* to address this challenge. It produces a set of code snippets that compensate for each other. The code ensemble can then effectively handle scenarios that require complex logic in the program. We design an evolutionary method to iteratively optimize the ensemble. Next, we elaborate this technique in more detail.

SEED produces multiple modular code snippets and combines them to form a complete solution. In particular, we design an *evolutionary algorithm* which produces a diverse set of programs, automatically measures their quality (using generated test cases), and aggregates the most effective ones to jointly solve the task.

The Evolutionary Algorithm. As shown in Alg. 3, the evolutionary algorithm contains two parts: *code branching* and *code filtering*. SEED starts by creating m initial code snippets. Then, in each evolutionary iteration, with the aid of LLMs, each code snippet branches into several new versions. All code snippets are then ranked and filtered based on their quality. At the end of each iteration, only the top-M ($M > m$) code snippets are preserved and are ready to branch in the next iteration. This evolutionary process continues until the code ensemble does not get better anymore on the validation set.

SEED uses the following *code branching* strategies:

- **Different Tools**: Here we ask LLMs to use different tools when generating the initial code snippet. For example, in semantic-related

Algorithm 3: Code Ensemble Evolution

Input: Intial Code Snippets $C^0 = \{R\}_{i=1}^{m}$
Output: Code Ensemble C^t

```
1  for t ← 0 to T − 1 do
2      C^{t+1} ← C^t;
3      for each R ∈ C^t do
4          E ← get_errors_in_verificatiion(R);
5          for each e ∈ E do
6              A ← get_advices(R, {e});
7              for each a ∈ A do
8                  C^{t+1}.add(fixing(R, {e}, a));
9          A ← get_advices(R, E);
10         for each a ∈ A do
11             C^{t+1}.add(fixing(R, E, a));
12     for each R_i ∈ C^{t+1} do
13         if R_i timeouts or fallbacks too often then
14             C^{t+1}.del(R_i);
15     for each R_i ≠ R_j ∈ C^{t+1} do
16         if R_j dominates R_i then
17             C^{t+1}.del(R_i);
18     sort C^{t+1} by accuracy from high to low;
19     C^{t+1} ← C^{t+1}_{0...m−1};
20 return C^T;
```

problems, using the `nltk` tool could lead to a syntax tree-based solution, while using `fuzzywuzzy` could lead to a string matching-based solution.

- **Prompt Rephrasing**: Here we use different task descriptions p_{task} when generating the initial code snippet. Rather than leaving the user with the prompt engineering job, using a prompt generator [15] to rephrase the user-provided task prompt could lead to different advice and potentially different solutions.
- **Diversify Advice**: When generating the initial code snippet or fixing a code snippet, we explicitly ask LLMs to produce multiple different pieces of advice to generate different code snippets (Alg. 3, L7, L10).
- **Varied Test Cases**: When fixing a code snippet, if the verification produces multiple error test cases, SEED presents a subset of errors to LLMs for advice: pick each one of the error cases (Alg. 3, L5~6) as well as the full error set (Alg. 3, L9). For a set of n errors, this produces $n + 1$ branches.

If, after applying the branching strategies to each of the initial m code snippets, more than M branches emerge, as in Fig. 3, SEED performs *code filtering* to rank and eliminate ineffective code snippets. A code snippet will be excluded if it meets one of the following criteria:

- **Dominated**: If another code snippet is correct on every test case where the current snippet is correct, the latter is deemed redundant and thus eliminated (Alg. 3, 16).
- **Costly**: Code snippets are encouraged to fall back to LLM queries when confidence is low. However, if a snippet always relies on LLM queries to produce correct answers, it becomes too expensive to utilize and should therefore be discarded. Similarly, snippets that potentially leverage computationally expensive tools are given lower priority or excluded (Alg. 3, L13).
- **Inaccurate**: Code snippets that are incorrect and produce an excessive number of false positives should be discarded. Specifically, after removing dominated and costly code snippets, the surviving code segments are sorted by accuracy on the validation set from high to low and at most M snippets with the highest accuracy are preserved (Alg. 3, L18).

After each iteration, at most M snippets are preserved, where $M > m$ is set as a parameter. This process iterates until convergence. The top-m preserved code snippets from the final iteration are then ensembled into the final solution for inference.

Ensembling. The code snippets are ensembled either in *parallel* or *sequentially*. By default, the code snippets are executed in parallel, and their execution results are aggregated to produce the final answer. This approach is suitable for most scenarios where the return values of the code snippets can be straightforwardly combined. For example, in classification tasks where the code snippets return values from a set of pre-determined classes, a simple majority voting strategy will work. Alternatively, an accuracy-weighted voting strategy can be employed to provide additional flexibility by assigning different weights to the snippets based on their accuracy on the test data, mitigating the influence of the low-accuracy candidates.

While the parallel ensemble is generally suitable, there are specific scenarios where aggregation is challenging. For instance, in data extraction tasks, where the extracted data can be any string, different snippets typically return different results. Running these snippets in parallel and taking the union of all reported answers may result in poor precision (high false positives), while taking their intersection may lead to poor recall (high false negatives). SEED addresses these cases by using a sequential ensemble.

The sequential ensemble first sorts all code snippets by their accuracy on a validation set, from high to low. Then on each data record, SEED executes the code snippet with the highest accuracy first. If this code snippet fails on this data record and returns 'None', the code snippet with the second highest accuracy will be triggered, and so on. If all code snippets fail, the system then falls back to the next module in the pipeline. With this cascading execution strategy, SEED dynamically adapts to the data and runs different code snippets on different data records, thus improving the accuracy.

4.2 ModelGen Module

In SEED, the *ModelGen* module distills a small machine learning model using the LLM responses as pseudo-ground truth. Although using LLMs for annotation is a known idea [23], choosing the model architecture appropriate for a given data curation task presents a challenge due to the vast number of DNN architectures available and the diversity of data curation tasks. It is often hard for the users to decide on the model architecture fitting their tasks.

SEED does not rely on users to select and tune models. Our key insight here is that almost all tasks with expected input-output pairs $\{(x, y)\}$ can be formulated as Seq2Seq generation tasks [12, 51], if we appropriately serialize x and y into a list of tokens. For instance, a data imputation task that infers the age of a person given their birth year and death year can be serialized as x = *"birth_year=?,death_year=?* and y = *"living_age=?"*, where *'?'*s are replaced with the decimal string of the actual year and age data. The Seq2Seq paradigm has been shown to be effective and generalizable in supporting NLP tasks [12, 18, 35], and has led to the development of pre-trained models such as T5 [12] that are widely applicable to a range of tasks.

This suggests it should be possible for us to employ a Seq2Seq-based architecture that can universally support many data curation tasks. For most data curation tasks, except classification and numeric tasks such as data imputation and data extraction, we can directly leverage the Seq2Seq paradigm, while for classification tasks such as entity resolution and data annotation, only the encoder component of the Seq2Seq model is utilized. The encoder is concatenated with an additional Multi-layer Perceptron (MLP) to function as either a classifier or a regressor, replacing the decoder in the Seq2Seq model. In these cases, rather than outputting a list of tokens, the small model produces either a probability distribution over candidate classes or a numeric value. SEED deduces the task type based on the output type, which is specified in the user configuration. A string type suggests a general Seq2Seq model; a float type suggests a regression; while a categorical type suggest a classification task.

When the ModelGen module is uncertain, if falls back to other modules. Specifically, a confidence score can be derived from the output of the deep neural network, and records with low confidence shall fall back to the LLM module. For Seq2Seq tasks, we use the perplexity $PPL(y) = \exp\left(-\frac{1}{n}\sum_{j=1}^{n}\log p(y_j|x, y_{1\ldots j-1})\right)$ of the generated text to compute confidence. Intuitively, perplexity is the inverse probability that the model predicts for each token in the output training sequence, given the tokens that came before it. A lower perplexity value corresponds to a higher level of confidence in the prediction. Thus, $\frac{1}{PPL(y)} \in (0, 1]$ could be used as the confidence indicator. As for classifiers, the probability associated with the highest predicted class can serve as an indicator of confidence. The closer this probability is to 1, the higher the confidence in the prediction. More specifically, given a K-way classification prediction, which is a probability distribution $\{p_y^i\}_{i=1}^K$, where p_y^i is the presumed probability that the instance belongs to class i, then $\frac{K\cdot\max_i(p_y^i)-1}{K-1} \in [0, 1]$ could be used as the confidence. Such a confidence threshold is a hyperparameter searched by the optimizer.

5 INFRASTRUCTURE

SEED infrastructure features *cache storage*, *query batching*, and *tools integration*. Universally supporting various data curation tasks, these components boost the performance of the automatically generated, domain specific data curation solution.

5.1 Cache Storage

The cache storage preserves all LLM queries and their corresponding responses throughout the compilation and execution phases. ModelGen utilizes the cached LLM responses as training data, while CacheReuse constructs a vector index and directly employs the indexed responses to respond to new LLM queries. Additionally, SEED logs the LLM calls invoked by the CodeGen module during code generation, thereby reducing the expense of repetitive code validation. Apart from storing LLM queries, SEED also caches the input-output pairs of each module, enabling cached validation when executing the specialized SEED optimizer (Sec. 3.2).

5.2 Query Batching

Query batching merges multiple queries into a single batched query. For the CacheReuse and ModelGen modules, it allows for batched vector computation. For the CodeGen module, it allows potential parallel execution. But more importantly, for the LLM modules, it eliminates the redundancy among the task descriptions and other auxiliary prompts, thus reducing the overall number of tokens sent to LLMs. Furthermore, in the context of few-shot or zero-shot learning, batching queries supplies LLMs with additional examples. These examples carry more information with respect to the data distribution, making LLMs more robust and reliable in answering queries, thereby potentially improving the effectiveness (Tab. 7).

For example, in a data annotation task that classifies the text into different topics, instead of asking separately *"Here are some topics: A. Sports B. Business C. Education D. Gaming ... What is the topic of the following paragraph? <TEXT>"* for each instance `<TEXT>`, we could ask *"Here are some topics: A. Sports B. Business C. Education D. Gaming ... What is the topic of each of the following paragraphs? 1.<TEXT1> 2.<TEXT2> 3.<TEXT3> ..."*.An LLM is then instructed to reply in the format of *"1. A. 2. F. 3. B. ..."*.

SEED investigates 5 methods to form a batch. These methods use the embedding of each individual data record produced by an Sentence-BERT encoder. Given n queries and a fixed batch size B:

- `RND`: random batching. It randomly composes n/B batches, making each batch adhere to the distribution of the query workload.
- `DIV`: diverse batching. It runs B-way balanced clustering on the embeddings, and then assigns queries in different clusters to the same batch.
- `PRX`: proximal batching. It runs n/B-way balanced clustering on the embeddings, and the queries in the same cluster form a batch.
- `FAR` & `CLS`: distance-based batching. It randomly samples a starting point and then iteratively adds the data point farthest/nearest from existing queries into the batch.

Intuitively, `SIM` and `CLS` present similar queries jointly to LLMs. The intuition underneath is to allow LLMs to see similar cases and thus easier to make decisions. On the other hand, `DIV` and `FAR` present diverse queries together to LLMs. This allows LLMs to see diverse cases and make more robust decisions.

5.3 Tools Integration

SEED seamless integrates the tools supplied by the users into the solution such that it could efficiently access local data and extract the required information on demand, without having to upload the entire data to LLMs. As an example, in a table retrieval task, allowing LLMs to use some tools to collect information from the local databases will help them better align the user queries with discovered tables and thus significantly enhance the correctness.

Once the users specify a set of tools, SEED has to decide which tools to use for a given LLM query. This is challenging because different queries, even if they belong to the same task, might need different tools under different situations. Moreover, it might need multiple tools to collaboratively solve a complex problem, where the order of calling these tools matters.

SEED leverages the reasoning ability of LLMs to address the above challenges. Rather than count on one single LLM query to solve the problem at once, SEED interacts with LLMs, automatically decomposes the original complex problem into multiple steps, and solves the problem step by step. In this iterative task-solving process,

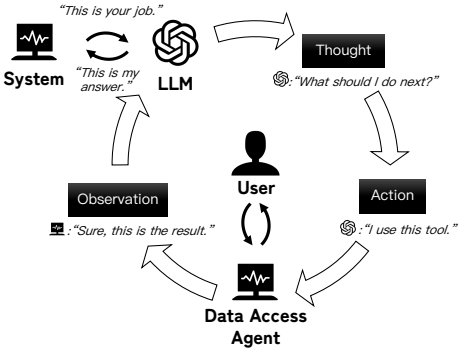

Figure 4: Illustration of iterative tool invocation.

it automatically and dynamically decides which tool to use based on the feedback acquired in the last step.

As shown in Fig. 4, there are three key parts in this process: action, thought, and observation [25]. Each action corresponds to a tool to be used. A thought corresponds to an idea that LLM suggests taking to solve the problem, while an observation represents the execution results after running a tool.

More specifically, given an input x and a list of actions $A = (T \bigcup \{\texttt{SUBMIT}\}) \times S$, where T represents the set of tools and S denotes their parameters. The `SUBMIT` action is a special tool that terminates the multi-step iterative process. In each step, the LLM first outputs its thought p_i in natural language, which triggers one of the actions $a_i = (t_i, s_i) \in A$. Consequently, SEED would invoke the tool t_i with parameter s_i corresponding to action a_i, resulting in an observation $o_i = t_i(s_i)$. Subsequently, based on the newly received observation o_i, the LLM produces a new thought p_{i+1} which invokes a new action $a_{i+1} \in A$. This process continues until an observation occurs that triggers a special termination tool $a = (\texttt{SUBMIT}, y)$, where the parameter y represents the LLM's final answer.

We use a table retrieval task as an illustrative example to further elucidate this process. With a set of pre-defined tools, given a query, the LLM starts by generating a thought that it should first get candidates through `SEARCH_KEYWORDS` or `BM25`. After the LLM chooses an action, SEED will invoke the corresponding tool and returns a list of candidate tables. Based on the list of candidate tables, the LLM may get a new thought that a few tables are worth further investigation. Consequently, it selects the `GET_SCHEMA` action to get the schema so that it has more information of these tables. After SEED returns the observation, the LLM either proceeds to get more candidates or invokes the `SUBMIT` action to submit some of these tables as the final answer.

6 EXPERIMENTS

Our experiments focus on answering the following questions:

- How do the solutions that SEED produces perform in comparison to its generic counterpart and other task specific solutions?
- How do the solutions that SEED produces perform in comparison to using LLMs to process each data record?
- how does the SEED optimizer perform?
- How do each module in SEED and the respective optimizations we propose perform?

As shown in Tab. 2, we selected 9 datasets spanning 5 tasks to show the performance of SEED. Together these tasks cover all types of modules in SEED and the typical execution pipelines. We present the experiments *task by task*, including the datasets the task runs on, the evaluation metric, the baseline methods we compare against, and the results. We evaluate the internals of the SEED optimizer on the entity resolution task with 2 datasets. Due to the limited space, we describe the user configuration (input text) in Appendix A and the data discovery experiment in Appendix B.1 of the extended version [2].

Parameter Settings. For all the experiments, we employ a GPT-4 [45] with temperature $T = 0$ as the LLM backend. Since SEED primarily operates through code and small models, or calls the LLM service, there is no special hardware required to run the experiments. In the case of ModelGen, we initialize it using a `t5-small` model and then fine-tune them on 1024 unlabeled validation data, employing LLM-generated pseudo labels to simulate a deployed system. The CacheReuse model uses a pre-trained frozen `t5-small` model checkpoint. CodeGen selects its branching number from either $m = 1$ or $m = 4$, where a branching number of $m = 4$ means that four code snippets are generated. As the cost of calling LLMs dominates the cost of the execution pipeline, we use the number of queries processed by the LLM to represent the cost C of a plan, denoted as "LLM R".

Table 2: Task, Datasets, and utilized components.

Task	Dataset	CodeGen	CacheReuse	ModelGen	LLM
Entity Resolution	Amazon-Google Abt-Buy	✓	✓	✓	✓
Data Imputation	Buy Restaurant	✓	✓	✓	✓
Data Extraction	Wiki NBA	✓			✓
Data Annotation	OS		✓		✓
Data Discovery	Spider OTT-QA KGDBQA				✓

6.1 Entity Resolution

Entity resolution determines whether two data records refer to the same entity.

Datasets. We use the *Amazon-Google* and *Abt-Buy* datasets [38] for evaluation. Both tasks aim to determine whether two online products are the same. Amazon-Google has three attributes: title, manufacturer, and price. Abt-Buy has three attributes as well: name, description, and price. We adopt the test split used in MatchGPT [46].

Evaluation Metric. We report the *F1* score, as in previous works.

Baselines. We compare against two approaches: (1) DITTO [31], a supervised learning method trained on thousands of labeled examples. (2) MatchGPT [46], the current SOTA in using LLMs to solve the entity resolution problem. MatchGPT has two versions. MatchGPT Generic manually selects few-shot examples, while MatchGPT Specialized utilizes the Jaccard similarity between two entities to retrieve few-shot examples.

In Table 3, "LLM Only" represents a solution that uses only the LLM module in SEED. "SEED Max" represents the best F1 score among all the plans enumerated by the SEED optimizer. "SEED Opt" denotes the plan selected by the SEED optimizer, when the effectiveness gap parameter G (Def. 3.1) is set to 5%. Table 3 shows that the SEED optimizer produces a plan that reduces the number of LLM calls by at least 90%, while guaranteeing that the F1 is at

Table 3: Entity Resolution: Quantitative Results

Method	Amazon-Google		Abt-Buy	
	F1	LLM R	F1	LLM R
DITTO	80.7	N / A	91.3	N / A
MatchGPT Generic	76.4	100.0%	95.8	100.0%
MatchGPT Specialized	85.2	100.0%	94.4	100.0%
SEED LLM Only	77.4	100.0%	92.1	100.0%
SEED Max	79.2	22.4%	92.8	27.1%
SEED Opt	76.2	10.4%	88.0	2.9%

most 5% lower than SEED Max. This confirms the effectiveness of the SEED optimizer. The accuracy of SEED is close to DITTO trained on thousands of labels and outperform MatchGPT Generic. MatchGPT Specialized has a higher F1 than SEED Max, because it uses prompts carefully tuned w.r.t. the dataset. SEED instead uses a generic prompt template without prompt engineering.

Table 4: Entity Resolution: SEED Optimizer

	Amazon-Google					Abt-Buy				
	Max		Opt		#Plan	Max		Opt		#Plan
	F1	LLM R	F1	LLM R		F1	LLM R	F1	LLM R	
BF	79.2	22.4%	76.8	3.7%	50146	92.8	27.1%	87.9	1.8%	50146
GEN	79.2	22.4%	76.8	3.7%	19541	92.8	27.1%	87.9	1.8%	20300
SPE	79.2	22.4%	76.2	10.4%	650	92.8	27.1%	88.0	2.9%	782

Next, we evaluate the effectiveness of the SEED optimizer. Table 4 presents the results of three different optimization approaches: 'BF' represents an exhaustive search over all possible plans; GEN represents the generic SEED optimizer (Section 3.1); while SPE represents the specialized SEED optimizer (Section 3.2). Note that the generic SEED optimizer consistently identifies the same "Max" and "Opt" plans to the exhaustive search, indicating that the assumption in Sec. 3 that the descendants or a dominated plan would not become a part of the final optimal plan holds in practice. The specialized SEED optimizer SPE produces the same "Max" plan but a slightly less efficient "Opt" plan. However, it reduces the number of plans that need to be searched by 99.5%. In the "Opt" plan produced by SPE, the LLM only handles 10.4% of the queries, while CacheReuse and ModelGen answer 41.7% and 47.9% of the queries respectively.

6.2 Data Imputation

As a typical data cleaning task, data imputation [49] replaces missing or corrupted data with substituted values.

Datasets. We use the *Buy* dataset [43] and *Restaurant* dataset [43] for evaluation. The Buy dataset has three attributes: product name, product description, and manufacturer. In this dataset, the manufacturer is masked and considered as the missing attribute to be imputed. The Restaurant dataset has five attributes: restaurant name, address, city, phone number, and food type. In this dataset, the city is masked and considered as the missing attribute to be imputed. For both datasets, when using CodeGen, we randomly select 3 examples that make at least one initially generated code snippets fail in validation.

Evaluation Metric. We report the *imputation accuracy*, where an imputation is considered correct if it exactly matches the gold value, non-trivially contains the gold value, or is non-trivially contained within the gold value.

Baselines. The only available baseline for few-shot imputation on the Buy dataset is FMs [43], a pure end-to-end LLM solution with 10-shot examples. In addition, we compare against the statistical data cleaning method HoloClean [27] and the supervised method IMP [32] which uses a large number of training data.

Table 5: Data Imputation

Method	Buy		Restaurant	
	Acc	LLM R	Acc	LLM R
HoloClean [27]	16.2	N / A	33.1	N / A
IMP [32]	96.5	N / A	77.2	N / A
FMs [43]	98.5	100.0%	88.4	100.0%
SEED LLM only (Max)	96.9	100.0%	89.8	100.0%
SEED Opt	91.9	39.3%	84.7	0.0%

As indicated in Table 5, SEED achieves an impressive imputation accuracy of 96.1% on the Buy dataset, using only 3 examples. This performance significantly surpasses the generic data cleaning method HoloClean [27]. Furthermore, it demonstrates comparable results to the state-of-the-art supervised solution IMP [32], which relies on training with thousands of labeled examples. When compared to FMs [43], which employ the LLM on each row in the table, SEED Opt represents the plan produced by the specialized SEED optimizer, with the accuracy gap G set to 5%. LLM only is the 'SEED Max' plan that the SEED optimizer finds. SEED Opt reduces the number of LLM calls by 60.7% on the Buy dataset, with only a 6.6% accuracy gap compared to the supervised IMP. In this plan, only 39.3% of queries are answered by the LLM, while 55.3% are answered by CodeGen, and 5.5% are answered by ModelGen. On the Restaurant dataset, SEED Opt uses only the CodeGen module, without making any LLM calls at all; and its accuracy is only 5% lower than the LLM only solutions.

For the code generated by SEED, please refer to Appendix D of the extended version [2].

6.3 Data Extraction

This task extracts structured information from unstructured data [7].

Datasets. We use the *Wiki NBA* dataset [21], which contains Wikipedia pages of NBA players. Each page is in a complex HTML format. This task extracts 19 user-defined attributes for each player, including name, height, college attended, and other relevant facts. When using CodeGen, for each of the 19 attributes to be extracted, we provided one example to help code generation: since there are empty attributes, we randomly select one example that has a non-empty value on this attribute.

Evaluation Metric. As in EVAPORATE [21], we report *token F1*. This is the geometric mean of the precision and recall, where precision measures the ratio of matched tokens among all extracted tokens and recall measures the ratio of matched tokens among all correct tokens.

Baselines. We compare against pre-trained machine learning models SimpleDOM, RoBERTa-Base, RoBERTa-Structural, and DOM-LM [30]; EVAPORATE-DIRECT [21], which is a pure LLM method; and EVAPORATE-CODE+ [21] which is an LLM-based code generation approach. Note the EVAPORATE approaches are specifically designed for data extraction.

As shown in Table 6, the plan produced by the SEED optimizer achieves the highest F1 score among all methods, with only 22.5%

Table 6: Data Extraction: Quantitative Results

Method	Wiki NBA F1
SimpleDOM [34]	28.8
RoBERTa-Base	34.2
RoBERTa-Structural	48.6
DOM-LM [30]	64.8
EVAPORATE-DIRECT [21]	84.6
EVAPORATE-CODE+ [21]	84.7
SEED Opt	88.6

cases processed by the LLM. The Performance of the trained machine learning models in general is poor, confirming that small models are lack of the semantics understanding ability to accurately extract data. However, EVAPORATE, which uses LLM to either directly extract data or produce code, shows impressive performance, indicating that using LLMs to synthesize code is effective on this task. For the code generated by SEED please refer to Appendix D of the extended version [2].

6.4 Data Annotation

Data annotation classifies data records into different categories.

Datasets. We use the *OS* dataset [8] for evaluation. This is a binary classification tasks where the goal is to annotate a tweet as either offensive or benign.

Evaluation Metric. We measure the *classification accuracy* as used in EFL [52].

Baselines. We compare against EFL [52] which achieves state-of-the-art few-shot results on the OS dataset amongst non-LLM approaches. We also compare against FT, which is a few-shot fine-tuning of a language model as the basic approach. Both EFL and FT use 8-shot examples. We also compare with an LLM-only approach.

Table 7: Data Annotation: Quantitative Results

Method	Accuracy	LLM R	#LLM
FT [52]	70.0	N / A	N / A
EFL [52]	79.8	N / A	N / A
SEED LLM Only	84.2	100.0%	100.0%
SEED Opt	88.3	31.1%	1.0%

As shown in Tab. 7, SEED achieves the best accuracy, while only calling the LLM on 31.1% of the data records. Moreover, due to the query batching optimization, SEED reduces the number of LLM calls and consumed tokens by 99% and 95% respectively, compared to the approaches that perform one individual LLM call on each data record. Note that the accuracy is even better than always querying the LLM. This is because the pseudo-annotated examples accumulated in the model module provide more information about the query workload, overcoming the few-shot limitations of querying the LLM, despite the relatively limited capacity of a small model.

7 RELATED WORK

LLMs in Data Curation. Most recently, the data curation researchers have begun to exploit LLMs' abilities to solve data curation problems. Specifically, most studies attempt to directly employ LLMs to solve a task by describing the task in a natural language. Zero-shot or few-shot LLMs are then utilized to generate textual responses, which are converted to the expected task output. For instance, in entity resolution, an LLM is asked if two records in a sentence belong to the same entity [39]. Such empirical research [9, 17, 26, 43] has demonstrated the feasibility of applying LLMs to data curation tasks. However, these studies mainly focus on designing appropriate prompts to make LLM effective for a specific task, while we aim to offer a generic tool to automatically synthesize domain-specific data curation solutions.

Directly using LLMs as a universal solution from data curation tasks, despite its convenience, has very high performance overhead as we have shown. Although some early studies have targeted some of these problems caused by the limited tunability and high computational cost of LLMs, these works, again, only focus on individual data curation tasks. For instance, EVAPORATE [21] focuses on information extraction; Chameleon [20] targets question answering; PromptNER [3] focuses on named entity recognition, etc.

Applied, Open Source LLM Projects. Apart from academic research, large open-source projects like langchain [4] have emerged to integrate recent advances in large models into a single platform. Nevertheless, similar to other infrastructures like huggingface [28], langchain mainly serves as an implementation platform where the users can leverage the existing techniques to develop LLM-based applications. It does not provide any built-in support to better solve data curation problems. Moreover, the efficiency and scalability issues are largely overlooked.

Individual applied projects are emerging as well, which implement and integrate practical optimizations into LLMs. For example, GPTCache [41] caches LLM QA responses for reuse; LlamaIndex [42] develops retrieval-augmented LLMs to overcome the hallucination problem; AgentGPT [47] focuses on customizing LLMs as dialogue agents; Auto-GPT [50] includes textual memory and internet access to search and gather information; ChatDB [11] extends LLMs with external symbolic memories, etc. However, each project only focuses on one specific optimization with straightforward implementation, rather than systematically studying and addressing the challenges raised in effectively and efficiently using LLMs to construct data curation solutions.

LLMs with Tools. Since ChatGPT plug-ins [44], a substantial number of studies have investigated the combination of LLMs and tools. For example, HuggingGPT [33] uses LLM to choose huggingface checkpoints; Chameleon [20] combines LLMs with a set of tools including web search and program execution; StructGPT [17] uses LLMs to generate parameters for function calling. Inspired by ReAct [25], the tools invocation in SEED incorporates a cognitive process to chose actions and provide feedback after each action. Moreover, SEED offers a set of predefined tools specifically designed for data curation tasks and allows users to use the synthesized code and small model modules as new tools.

8 CONCLUSION

In this paper, we introduce SEED, a system that leverages LLMs to synthesize domain-specific data curation solutions, which by fully leveraging LLMs' synthesis, reasoning, semantics understanding abilities as well as the encoded common knowledge, ensure the effectiveness and efficiency of the automatically produced solutions. The results confirm that SEED achieves SOTA performance on various tasks and significantly reduces the number of LLM calls.

REFERENCES

[1] 2023. Ternary Search - Algorithms for Competitive Programming. https://cp-algorithms.com/num_methods/ternary_search.html.

[2] Anonymous. 2023. SEED: Domain-Specific Data Curation With Large Language Models. https://anonymous.4open.science/r/SEED/paper.pdf

[3] Dhananjay Ashok and Zachary C. Lipton. 2023. PromptNER: Prompting For Named Entity Recognition. *CoRR* abs/2305.15444 (2023). https://doi.org/10.48550/ARXIV.2305.15444

[4] Harrison Chase. 2022. LangChain. https://github.com/hwchase17/langchain.

[5] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to Answer Open-Domain Questions. In *Association for Computational Linguistics (ACL)*.

[6] Wenhu Chen, Ming wei Chang, Eva Schlinger, William Wang, and William Cohen. 2021. Open Question Answering over Tables and Text. *Proceedings of ICLR 2021* (2021).

[7] James R. Cowie and Wendy G. Lehnert. 1996. Information Extraction. *Commun. ACM* 39, 1 (1996), 80–91.

[8] Thomas Davidson, Dana Warmsley, Michael W. Macy, and Ingmar Weber. 2017. Automated Hate Speech Detection and the Problem of Offensive Language. In *ICWSM 2017*. AAAI Press, 512–515.

[9] Aakanksha Chowdhery et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[10] Bei Chen et al. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=ktrw68Cmu9c

[11] Chenxu Hu et al. 2023. ChatDB: Augmenting LLMs with Databases as Their Symbolic Memory. *CoRR* abs/2306.03901 (2023).

[12] Colin Raffel et al. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[13] Dong Huang et al. 2024. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010 [cs.CL]

[14] El Kindi Rezig et al. 2019. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. *Proc. VLDB Endow.* 12, 12 (2019), 1954–1957.

[15] Fatih Kadir Akın et al. 2023. Awesome ChatGPT Prompts. https://github.com/f/awesome-chatgpt-prompts.

[16] Hung Le et al. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html

[17] Jinhao Jiang et al. 2023. StructGPT: A General Framework for Large Language Model to Reason over Structured Data. *CoRR* abs/2305.09645 (2023).

[18] Mike Lewis et al. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7871–7880.

[19] Michael Stonebraker et al. 2013. Data Curation at Scale: The Data Tamer System. In *CIDR*.

[20] Pan Lu et al. 2023. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. *CoRR* abs/2304.09842 (2023).

[21] Simran Arora et al. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *Proc. VLDB Endow.* 17, 2 (2023), 92–105. https://www.vldb.org/pvldb/vol17/p92-arora.pdf

[22] Sirui Hong et al. 2023. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352 [cs.AI]

[23] Shuohang Wang et al. 2021. Want To Reduce Labeling Cost? GPT-3 Can Help. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 4195–4205. https://doi.org/10.18653/V1/2021.FINDINGS-EMNLP.354

[24] Shiqi Wang et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 13818–13843. https://doi.org/10.18653/V1/2023.ACL-LONG.773

[25] Shunyu Yao et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models.

[26] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners.

[27] Theodoros Rekatsinas et al. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *VLDB* (2017).

[28] Thomas Wolf et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online.

[29] Tao Yu et al. 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, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff et al. (Ed.). Association for Computational Linguistics, 3911–3921. https://doi.org/10.18653/V1/D18-1425

[30] Xiang Deng et al. 2022. DOM-LM: Learning Generalizable Representations for HTML Documents. *CoRR* abs/2201.10608 (2022). https://arxiv.org/abs/2201.10608

[31] Yuliang Li et al. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (2020), 50–60. https://doi.org/10.14778/3421424.3421431

[32] Yinan Mei et al. 2021. Capturing Semantics for Imputation with Pre-trained Language Models. In *ICDE*.

[33] Yongliang Shen et al. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in HuggingFace. *CoRR* abs/2303.17580 (2023). https://doi.org/10.48550/ARXIV.2303.17580

[34] Yichao Zhou et al. 2021. Simplified DOM Trees for Transferable Attribute Extraction from the Web. *CoRR* abs/2101.02415 (2021). https://arxiv.org/abs/2101.02415

[35] Zhengxiao Du et al. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 320–335.

[36] Zihui Gu et al. 2023. Interleaving Pre-Trained Language Models and Large Language Models for Zero-Shot NL2SQL Generation. *CoRR* abs/2306.08891 (2023). https://doi.org/10.48550/ARXIV.2306.08891

[37] André Freitas and Edward Curry. 2016. Big data curation. (2016).

[38] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.* 3, 1 (2010), 484–493. https://doi.org/10.14778/1920841.1920904

[39] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.* 3, 1 (2010), 484–493.

[40] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers. 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)*. Association for Computational Linguistics, Online, 2261–2273. https://aclanthology.org/2021.acl-long.176

[41] Frank Liu. 2023. GPTCache : A Library for Creating Semantic Cache for LLM Queries. https://github.com/zilliztech/GPTCache.

[42] Jerry Liu. 2022. *LlamaIndex*. https://doi.org/10.5281/zenodo.1234

[43] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *Proc. VLDB Endow.* 16, 4 (2022), 738–746. https://doi.org/10.14778/3574245.3574258

[44] OpenAI. 2023. ChatGPT — Release Notes. https://help.openai.com/en/articles/6825453-chatgpt-release-notes.

[45] OpenAI. 2023. GPT-4 Technical Report.

[46] Ralph Peeters and Christian Bizer. 2024. Entity Matching using Large Language Models. arXiv:2310.11244 [cs.CL]

[47] reworkd. 2023. AgentGPT. https://github.com/reworkd/AgentGPT.

[48] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389. https://doi.org/10.1561/1500000019

[49] D. B. Rubin. 1987. *Multiple Imputation for Nonresponse in Surveys*. Wiley. 258 pages.

[50] Significant-Gravitas. 2023. Auto-GPT: An Autonomous GPT-4 Experiment. https://github.com/Significant-Gravitas/Auto-GPT.

[51] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani et al. (Ed.). 3104–3112.

[52] Sinong Wang, Han Fang, Madian Khabsa, Hanzi Mao, and Hao Ma. 2021. Entailment as Few-Shot Learner. *CoRR* abs/2104.14690 (2021).

A TASK CONFIGURATION

Data Imputation. For the Buy dataset, the API is defined as *"data_imputation(name,desc) -> manufacturer"*, with description *"name: str. The name of the product."* and *"desc: str. The description of the product. Might be empty (string 'nan')."*. The task is described as *"Given a product's information online, please deduce its manufacturer."*. For the Restaurant dataset, the API is defined as *"data_imputation(name,desc) -> manufacturer"*, with description *"name: str. The name of the restaurant."*, *"addr: str. The address of the restaurant."*, *"phone: str. The phone number of the restaurant."*, *"type: str. The food type of the restaurant."*, and *"city: str. The city the restaurant is in."*. The task is described as *"Given a restaurant's information, deduce the city it is located in."*.

Data Extraction. For this data extraction task on *Wiki NBA* data set [21], as 19 different attributes are required to be extracted, which subject to different rules, we generate 19 different programs using the attribute placeholder *"<ATTR>"*. The API is defined as *"information_extraction_<ATTR>(html) -> <ATTR>"*, with description *"html: str. The HTML string."* and *"str. The <ATTR> information extracted."*. The task is described as *"Given the HTML string of the wikipedia page of an NBA basketball player, extract the player's <ATTR> information."*.

Data Annotation. For this task, the API is defined as *"topic_classification(text) -> topic"*, with description *"text: str. The tweet message."* and *"topic: int. Output 0 for hatespeech, 1 for benign."*. The task is described as *"Given a tweet message, determine whether it is hatespeech or benign."*.

Entity Resolution. For the entity resolution task, both data sets (*Amazon-Google* and *Abt-Buy* [38]) share the same API definition *"entity_resoluion(entity1, entity2) -> is_same"* and task description *"Given two products, determine whether they are the same product."*, however with different descriptions. The entity description for Amazon-Google is *"entity1: dict. It contains three attributes: 'title', 'manufacturer', 'price'. 'title' and 'manufacturer' are strings, 'price' is float."*, while the entity description for Abt-Buy is *"entity1: dict. It contains three attributes: 'name', 'description', 'price'. 'name' and 'description' are strings, 'price' is float."*.

Data Discovery. For this task, on all data sets (*Spider* [29] and *OTT-QA* [6], and *KaggleDBQA*) [40] datasets. All datasets share the same task of table retrieval. The difference is that multiple tables in the Spider dataset are related to the query, while OTT-QA has only one ground truth table for each query. For the Spider dataset, the API is defined as *"data_discovery(query) -> tables_list"*, with description *"query: str. The natural language query."* and *"tables_list: List[str]. A list of table names that are found related to the given query."*. The task is described as *"Given a natural language query, find tables that can help answer the query."*. For the OTT-QA dataset, instead of *"tables_list"*, only a single *"table"* is asked for. In the current infrastructure, we provided five tools: `GET_SCHEMA(table_name)` which gets the schema of a specified table, `SEARCH_KEYWORDS(keywords)` which finds the top-20 tables related to a list of keywords using exact string match, `SEARCH_VALUE(value)` which finds the top-20 tables related to a value using a fixed 0.2 Levenshtein distance filter, `JOINT_SEARCH(keywords, value)` which retrieves both, and `BM25(query)` which finds the top-20 tables sorted by BM25.

B EXPERIMENTS

B.1 Data Discovery

In this experiment, we use a *table retrieval* task to evaluate SEED's performance using LLM module with tools integration. As a specific data discovery task, table retrieval finds data that is relevant to a given natural language query.

Datasets. We use the *Spider* dataset [29], the *OTT-QA* dataset [6], and the *KaggleDBQA* dataset [40]. Because Spider and KaggleDBQA are NL2SQL benchmarks, we treat one *"question"* as the input query and the tables involved in the corresponding ground-truth SQL query as the related tables. For the OTT-QA dataset, we use its specified table retrieval task. We use only one manual example query to demonstrate to the LLM how to use the tools. Apart from `SUBMIT`, the available tools include `GET_SCHEMA`, `BM25`, `SEARCH_KEYWORDS`, `SEARCH_VALUE`, and `JOINT_SEARCH`.

Evaluation Metric. We measure the *F1 score* which reflects both precision and recall, where precision represents the ratio of ground truth tables within discovered tables, and recall measures the ratio of discovered tables within ground truth tables.

Baselines. We compare SEED with BM25 [48] (a classic information retrieval method) as well as DrQA [5], an embedding-based method. We consider the top-3 results returned by the BM25 and DrQA method as predictions on the Spider and KaggleDBQA dataset. For the OTT-QA dataset, since there is only one ground truth table for each query, we only consider the top-1 result returned by the BM25 and DrQA method. Additionally, we compare our solution with a baseline approach that is LLM only, but not augmented with our tool integration. In this case, we present to the LLM the top-20 candidates returned by BM25 by concatenating their titles and schema. This is because each time LLM can only ingest at most 20 candidates due to the context length constraint, while iteratively feeding all tables to the LLM is not feasible due to cost issues.

Table 8: Data Discovery: Quantitative Results

Method	Spider	OTT-QA	KGDBQA
BM25	34.8	62.0	24.1
DrQA	35.5	50.0	34.7
SEED LLM only	50.7	71.0	56.1
SEED Opt	85.7	67.8	63.5

As show in Table 8, SEED LLM only represents using the LLM module without tools integration, while Opt adds tools. With the assistance of the tools, SEED LLM module surpasses BM25 and DrQA on all datasets and significantly outperforms the solution that only uses the LLM on Spider and KGDBQA. However, SEED is less effective on the OTT-QA dataset. This is because the titles of the tables in this dataset are typically meaningless, thus often misleading SEED when selecting the tools.

B.2 Ablation Study

We mainly conducted the ablation study on data annotation on the OS dataset, to investigate the impact of different query batching sizes and strategies, as well as different distance thresholds for the CacheReuse module.

Query Batching. We tested how the *batching* optimization for the LLM module performs. Here SEED LLM Only refers to directly querying GPT-4 with the few-shot prompt produced by SEED's prompt template.

Table 9: Data Annotation: Varying Batch Sizes

Method	OS	#LLM	#Tokens
SEED LLM Only	84.2	512	$\sim 443K$
SEED (Batch$^{B=4}$)	83.4	128	$\sim 170K$
SEED (Batch$^{B=8}$)	85.4	64	$\sim 113K$
SEED (Batch$^{B=16}$)	89.3	32	$\sim 85K$
SEED (Batch$^{B=32}$)	90.6	16	$\sim 70K$

Varying Batch Sizes. Herein, B denotes the number of queries within a single batch. We increase the batch size until we reach the maximal context length of an LLM. As shown in Table 9, batching significantly reduces both the number of required LLM queries and the total number of tokens submitted to the LLM. The larger the batch size, the more the saving. Moreover, larger batch size tends to lead to better accuracy. This is because a larger batch size provides more contextual information for the LLM to make better decisions. Therefore, our batching optimization is a win-win.

Table 10: Ablation Study on Batching Strategy.

Method	OS
SEED (Batch$^{B=32}$)	90.6
SEED (Batch$^{B=32,\texttt{DIV}}$)	92.4
SEED (Batch$^{B=32,\texttt{PRX}}$)	91.6
SEED (Batch$^{B=32,\texttt{CLS}}$)	89.6
SEED (Batch$^{B=32,\texttt{FAR}}$)	90.8

Different Batching Strategies. In SEED, the default batching strategy adheres to the distribution of the dataset by randomly sampling queries to form a batch. Other batching strategies including `DIV`, `PRX`, `CLS`, `FAR` are discussed in Sec. 5.2. As shown in Tab. 10, different batching strategies lead to slightly different accuracies. Because the performance of the default random sampling strategy is already satisfactory, we recommend using the random sampling to save users' effort in selecting the best strategy.

Frozen Model with Different Distance Thresholds. The OS dataset is a relatively easy dataset, which enables a large distance threshold for the CacheReuse. In The extreme case, when using a relatively large d threshold, SEED reduces the ratio of LLM calls on OS by 95.5%, with an accuracy comparable to or even better than always querying GPT-4.

Table 11: Data Annotation: Varying Threshold d

Method	OS	k (#LLM)	LLM R
SEED (LLM only)	84.2	512	100.0%
SEED (CacheReuse$^{d=0.4}$)	86.1	368	71.9%
SEED (CacheReuse$^{d=0.6}$)	88.3	159	31.1%
SEED (CacheReuse$^{d=0.8}$)	87.9	64	12.5%
SEED (CacheReuse$^{d=1.0}$)	86.7	23	4.5%

C PROMPTS

As the prompts in SEED are dynamically composed, we use <> to represent placeholders for simplicity.

C.1 Prompts Related to Direct LLM Querying

Task Profile. The task profile contains the description of the task, inputs, outputs, and examples. This is the basic building block and will be repeatedly used in other prompts as a component. We will refer to it as `<task_profile>` in the future. We demonstrate the task profile with an entity resolution example.

```
Given two products, determine whether they are the same
    product.
The input contains the following attributes:
- entity1: dict. It contains three attributes: `title`, `
    manufacturer`, `price`. `title` and `manufacturer`
    are strings, `price` is float.
- entity2: dict. Same as entity1.
You are expected to output:
- int. Output 0 if the two product are not identical, 1
    of the two products are identical.
Examples:
Example #0:
Inputs:
- entity1: {'title': "mia 's math adventure : just in
    time", 'manufacturer': 'kutoka', 'price': 19.99}
- entity2: {'title': "kutoka interactive 61208 mia 's
    math adventure : just in time !", 'manufacturer': '
    kutoka interactive', 'price': 24.99}
Output:
- 1
Example #1:
Inputs:
- entity1: {'title': 'adobe creative suite cs3 design
    standard upgrade [ mac ]', 'manufacturer': 'adobe',
    'price': 399.0}
- entity2: {'title': '29300183dm adobe creative suite 3
    design standard media tlp tlp nonprofit download -',
     'manufacturer': nan, 'price': 20.97}
Output:
- 0
Example #2:
Inputs:
- entity1: {'title': 'instant immersion 33 languages', '
    manufacturer': 'topics entertainment', 'price':
    49.99}
- entity2: {'title': 'instant immersion 33 languages', '
    manufacturer': nan, 'price': 47.36}
Output:
- 1
```

LLM Query Prompt. The LLM query prompt simply reuses the task profile and adds an instance. We demonstrate the task profile with a specific entity resolution instance.

```
<task_profile>
Now consider the following instance:
- entity1: {'title': 'high school advantage 2008', '
    manufacturer': 'encore', 'price': 39.99}
- entity2: {'title': 'elementary advantage 2008 encore',
    'manufacturer': 'nan', 'price': 39.95}
Please respond with the answer only. Please do not output
     any other responses or any explanations.
```

C.2 Prompts Related to Code Generation

Tools Profile. This describes the tools which can either be used in code generation (e.g., using a third-party python package) or in the LLM module as interactive tools. Each tool is in the format of `- <tools_api>: <tools_desc>`. For example, in data discovery:

```
- SUBMIT(table): Input a string. Submit the table.
- GET_SCHEMA(table_name): Input a string. Return the schema (the name of all the columns) of the given table.
- SEARCH_KEYWORDS(keywords): Input a list of keywords. Return a list containing at most 20 tables whose title or schema strictly contains at least one given keyword. The tables that contain more keywords will be ranked higher.
- SEARCH_VALUE(value): Input a value in any type. Return a list containing at most 20 tables that contains this value (the value is fuzzy matched as a string). The tables that contain more of this value will be ranked higher.
- BM25(query): Input a string. Return a list containing at most 20 tables that are related to the query, found by running bm25 algorithm over table title and schema.
- JOINT_SEARCH(keywords, value): Input a list of keywords and a value. Return a list containing at most 20 tables that contains both the value and at least one of the keywords.
```

When the tools profile is used in code generation a default profile allows the code to use any python package:

```
- python packages: You can use any python packages you want. You do not need to install but only import them before using. You can not use supervised-learning method as there is no training data. Though, you can use frozen models if you want.
```

We will refer to the tools profile as `<tools_profile>` in the future.

Advice Generation.

```
Please help me with the following Python programming task:
<task_profile>
<tools_profile>
Notice that the evaluation will severely punish incorrect outputs. Thus, when the function is uncertain, please return `None` to abstain instead of returning an incorrect guess.
The generated function should be robust, instead of only passing the provided examples. You are allowed use global variables to keep track of new incoming data.
Please provide a brief advice on how I should complete this programming task. Provide 2-3 concise sentences summarizing the key coding strategy.
```

We will refer to the advice generated as `<advice>` in the future.

Code Generation.

```
Please write a Python function `<task_api>` that completes the following goal:
<task_profile>
<tools_profile>
Hint: <advice>
Notice that the evaluation will severely punish incorrect outputs. Thus, when the function is uncertain, please return `None` to abstain instead of returning an incorrect guess.
The generated function should be robust, instead of only passing the provided examples. You are allowed use global variables to keep track of new incoming data.
Please respond with the Python implementation of `<task_api>` only. Please do not output any other responses or any explanations.
Your response should be in the following format (the markdown format string should be included):
```python
def <task_api>:
    '''Your Implementation Here.'''
```
```

Logical Correctness Evaluation.

```
Consider the following task:
<task_profile>
Consider the following Python function `<task_api>` that is expected to complete the above task:
```python
<code>
```
Please determine whether the function `<task_api>` is correct.
Please output your judgement in the following format: first output "Thought:" and then thoughts on whether this function is correct or not, then output "Answer:" followed by a single answer "yes" or "no".
```

Example (Test Case) Generation. Each test case is viewed as an assertion statement in Python. We demonstrate the example generation using data imputation on the Buy dataset:

```
Consider the following task:
<task_profile>
Consider the following Python function `data_imputation(name, desc)` that is expected to complete the above task:
```python
<code>
```
Please provide a 3 ~ 5 test cases that you think are effective and comprehensive for determining whether the program is correct.
You should focus on the logical correctness of the program.
Do not design corner cases such as small floating point errors, values beyond natural ranges, behaviors around boundary, etc.
The provided test cases should be multiple lines of compilable code, each line should be in the same format as below:
```python
assert data_imputation(name, desc) == ?
```
Example test cases:
```python
assert data_imputation("Linksys EtherFast EZXS55W Ethernet Switch", "5 x 10/100Base-TX LAN") == "Linksys"
assert data_imputation("PlayStation 2 Memory Card 8MB", "nan") == "Sony"
assert data_imputation("Directed Electronics SCH1 SiriusConnect Home Tuner", "SIRIUS SCH1 SIRIUSConnect Home Tuner") == "Sirius"
```
Now, please design some new test cases. Please respond with the test cases only.
```

Some generated test cases:

```
assert data_imputation("Apple iPhone 13 Pro Max", "A15 Bionic chip with Neural Engine") == "Apple"
assert data_imputation("Samsung Galaxy Watch 4", "nan") == "Samsung"
assert data_imputation("Canon EOS R5 Mirrorless Camera", "45 Megapixel Full-Frame CMOS Sensor") == "Canon"
assert data_imputation("Bose QuietComfort 35 II Wireless Headphones", "nan") == "Bose"
assert data_imputation("LG 55-inch 4K OLED Smart TV", "Dolby Vision and Dolby Atmos") == "LG"
```

Code Fixing Decision.

```
Consider the following task:
<task_profile>
Consider the following Python function `data_imputation(name, desc)` that is expected to complete the above task:
```python
<code>
```
The function seems to be incorrect:
<error_info>
Please determine whether: A. the function is actually incorrect and can be fixed. B. the function is actually incorrect and can not be easily fixed. C. the function is correct while the case evaluation is incorrect.
Please output your judgement in the following format: first output "Thought:" and then thoughts on whether this function is correct or not, then output "Answer:" followed by a single answer "A", "B" or "C".
```

Code Fixing Advice Generation.

```
Consider the following task:
<task_profile>
Consider the following Python function `data_imputation(name, desc)` that is expected to complete the above task:
```python
<code>
```
The function seems to be incorrect:
<error_info>
Please provide a brief advice on how I should fix the function. Provide 2-3 concise sentences summarizing the key coding strategy.
The fix should be robust and general, instead of only passing the provided error cases.
```

Code Fixing.

```
Consider the following task:
<task_profile>
Consider the following Python function `data_imputation(name, desc)` that is expected to complete the above task:
```python
<code>
```
The function seems to be incorrect:
<error_info>
Hint: <advice>
Please fix the code. The fix should be robust and general, instead of only passing the provided error cases.
Please respond with the fixed Python implementation of `<task_api>` only. Please do not output any other responses or any explanations.
Your response should be in the following format (the markdown format string should be included):
```python
def <task_api>:
    '''Your Implementation Here.'''
```
```

C.3 Prompts Related to LLM Query

Batched Query Prompt. The batched query prompt modifies the LLM query prompt by batching multiple instances. For simplicity, we demonstrate the batched query prompt with the entity resolution instance from the LLM query prompt and omit the other instances.

```
<task_profile>
Now consider the following instances:
Instance #1:
- entity1: {'title': 'high school advantage 2008', 'manufacturer': 'encore', 'price': 39.99}
- entity2: {'title': 'elementary advantage 2008 encore', 'manufacturer': 'nan', 'price': 39.95}
Instance #2:
...
Please respond with the answer only, one line for each instance. Please do not output any other responses or any explanations.
Each response should start with "Output #<index>: ". For example:
Output #1: ...
Output #2: ...
```

LLM with Tools Integration. As the invocation process is iterative, we first show the initial prompt:

```
<task_profile>
Please do not directly answer the problem. This should be an interactive process. You are allowed to take one of the following actions:
<tools_profile>
Interaction Examples:
<interaction_examples>
Now consider the following instance:
Your respond should strictly follow this format: first output `Thought:` followed by your thought process, then output `Action:` followed by one of the actions mentioned above.
```

Then we demonstrate the interaction process with a data discovery instance on Spider:

```
...
Thought: I need to find tables related to the query, try with keywords 'employees', 'job', and 'past'.
Action: SEARCH_KEYWORDS(['employees', 'job', 'past'])
Observation: The top results: hr_1-employees, store_1-employees, phone_market-market, network_2-Person, hr_1-jobs, hr_1-job_history, department_store-Staff_Department_Assignments, department_management-department, customer_deliveries-Employees, cre_Doc_Tracking_DB-Employees, cre_Doc_Control_Systems-Employees, college_1-EMPLOYEE, yelp_user, yelp_tip, yelp_review, yelp_neighbourhood, yelp_checkin, yelp_category, yelp_business, wta_1-rankings

Thought: It seems `hr_1-employees`, `hr_1-jobs`, and `hr_1-job_history` are related to the query. I should confirm them by checking their schema.
Action: GET_SCHEMA('hr_1-employees')
```

```
Observation: The table has columns: `EMPLOYEE_ID`, `FIRST_NAME`, `LAST_NAME`, `EMAIL`, `PHONE_NUMBER`, `HIRE_DATE`, `JOB_ID`, `SALARY`, `COMMISSION_PCT`, `MANAGER_ID`, `DEPARTMENT_ID`

Thought: This table seems to be related to the query as it contains information about employees and their jobs. Now I need to check the schema of the other tables.
Action: GET_SCHEMA('hr_1-jobs')
Observation: The table has columns: `JOB_ID`, `JOB_TITLE`, `MIN_SALARY`, `MAX_SALARY`

Thought: This table seems to be related to the query as it contains information about jobs. Now I need to check the schema of the last table.
Action: GET_SCHEMA('hr_1-job_history')
Observation: The table has columns: `EMPLOYEE_ID`, `START_DATE`, `END_DATE`, `JOB_ID`, `DEPARTMENT_ID`

Thought: This table seems to be related to the query as it contains information about the job history of employees. It is time to submit the tables related to the query.
Action: SUBMIT(['hr_1-employees', 'hr_1-jobs', 'hr_1-job_history'])
```

D GENERATED CODE

D.1 Examples for Data Imputation

Example #1: a code snippet generated for the Buy dataset with LLM-generated advice.

```python
# Advice: Use a Named Entity Recognition (NER) model. NER is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc. In this case, you can train or use a pre-trained NER model to recognize manufacturer names in the product name and description. This method requires a lot of computational resources and may not be feasible if you don't have a large amount of training data.
import spacy
nlp = spacy.load("en_core_web_sm")
def data_imputation(name, desc):
    doc = nlp(name)
    for ent in doc.ents:
        if ent.label_ == "ORG":
            return ent.text
    if desc != 'nan':
        doc = nlp(desc)
        for ent in doc.ents:
            if ent.label_ == "ORG":
                return ent.text
    return None
```

Example #2: a code snippet generated for the Buy dataset with an alternate advice also generated by LLM.

```python
# Advice: Use a pre-defined dictionary of known manufacturers and their products. This method requires a lot of manual work to create and maintain the dictionary, but it can be very accurate. For example, you can define a dictionary where the keys are product names or keywords (like "PlayStation") and the values are the corresponding manufacturers (like "Sony"). Then, you can search for these keywords in the product name and description to deduce the manufacturer.
def data_imputation(name, desc):
    manufacturers = {
        "Linksys": ["EtherFast", "EZXS55W", "Ethernet Switch"],
        "Sony": ["PlayStation"],
        "Sirius": ["SiriusConnect", "Home Tuner"]
    }
    for manufacturer, keywords in manufacturers.items():
        for keyword in keywords:
            if keyword in name or (desc != 'nan' and keyword in desc):
                return manufacturer
    return None
```

Example #3: a code snippet branched from failure cases. More manufacturers are included in the list.

```python
def data_imputation(name, desc):
    import re
    # List of common manufacturers
    manufacturers = ['Linksys', 'Sony', 'Sirius', 'Apple', 'Samsung', 'Microsoft', 'Dell', 'HP', 'Lenovo', 'Asus', 'Acer', 'LG', 'Panasonic', 'Philips', 'Canon', 'Nikon', 'Bose', 'JBL', 'Logitech', 'Netgear', 'Seagate', 'Western Digital', 'Kingston', 'SanDisk', 'Corsair', 'Intel', 'AMD', 'Nvidia', 'Qualcomm', 'Broadcom']
    # Check if manufacturer is in the name
    for manufacturer in manufacturers:
        if manufacturer.lower() in name.lower():
            return manufacturer
    # Check if manufacturer is in the description
    if desc != 'nan':
        for manufacturer in manufacturers:
            if manufacturer.lower() in desc.lower():
                return manufacturer
    # If manufacturer is not found, return None
    return None
```

Example #4: a code snippet generated for the Restaurant dataset with LLM-generated advice and improved upon failure cases.

```python
# Use Geocoding APIs: You can use a geocoding API, such as Google's Geocoding API, to convert the restaurant's address into geographic coordinates, and then extract the city name from the returned data. This approach is more complex and requires internet access, but it is likely to be the most accurate, especially if the restaurant's name and phone number do not contain the city name.
import re
import phonenumbers
import usaddress
from geopy.geocoders import Nominatim
def data_imputation(name, addr, phone, type):
    # Extract city from restaurant name
    city_in_name = re.search(r'\((.*?)\)', name)
    if city_in_name:
        return city_in_name.group(1).lower()
    # Extract city from phone number
    try:
```

```
        parsed_phone = phonenumbers.parse(phone, 'US')
        if parsed_phone.country_code == 1 and parsed_phone.national_number:
            city_in_phone = phonenumbers.geocoder.description_for_number(parsed_phone, 'en')
            if city_in_phone:
                return city_in_phone.lower()
    except:
        pass
    # Extract city from address using Geocoding API
    geolocator = Nominatim(user_agent="geoapiExercises")
    location = geolocator.geocode(addr)
    if location:
        address_components = usaddress.parse(location.address)
        for component in address_components:
            if 'PlaceName' in component:
                return component[0].lower()
    return None
```

D.2 Examples for Information Extraction

Example #5: a code snippet generated for the Wiki NBA dataset *without* LLM-generated advice.

```
from bs4 import BeautifulSoup
import re
def information_extraction_college(html):
    soup = BeautifulSoup(html, 'html.parser')
    for table in soup.find_all('table', {'class': 'infobox'}):
        for row in table.find_all('tr'):
            th = row.find('th')
            if th and 'college' in th.text.lower():
                td = row.find('td')
                if td:
                    college = re.sub(r'\[[^]]*\]', '', td.text) # Remove any reference tags
                    return college.strip()
    return None
```

Example #6: a code snippet generated for the Wiki NBA dataset with LLM-generated advice. The main difference is that the class containing the relevant information is not "infobox" but "infobox vcard", which is addressed in the advice.

```
# You can use BeautifulSoup, a Python library for parsing HTML and XML documents. It creates parse trees that are helpful to extract the data easily. You can search for the player's college information by finding the table with class 'infobox vcard' which usually contains the player's information on Wikipedia. Then, iterate over the rows of the table until you find a row with the header 'College'. The next cell in that row should contain the college information. If no such row is found, return None.
from bs4 import BeautifulSoup
def information_extraction_college(html):
    soup = BeautifulSoup(html, 'html.parser')
    table = soup.find('table', {'class': 'infobox vcard'})
    if table:
        for row in table.find_all('tr'):
            if row.find('th') and 'College' in row.find('th').text:
                return row.find('td').text.split('(')[0].strip()
    return None
```