2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Automated, Unsupervised, and Auto-parameterized Inference of Data Patterns and Anomaly Detection

Qiaolin Qin, Heng Li*, Ettore Merlo, Maxime Lamothe
Dept. of Computer and Software Engineering
Polytechnique Montreal, Montreal, Canada
{qiaolin.qin, heng.li, ettore.merlo, maxime.lamothe}@polymtl.ca

***Abstract*— With the advent of data-centric and machine learning (ML) systems, data quality is playing an increasingly critical role for ensuring the overall quality of software systems. Data preparation, an essential step towards high data quality, is known to be a highly effort-intensive process. Although prior studies have dealt with one of the most impacting issues, data pattern violations, these studies usually require data-specific configurations (i.e., parameterized) or use carefully curated data as learning examples (i.e., supervised), relying on domain knowledge and deep understanding of the data, or demanding significant manual effort. In this paper, we introduce RIOLU: Regex Inferencer autO-parameterized Learning with Uncleaned data. RIOLU is fully automated, automatically parameterized, and does not need labeled samples. RIOLU can generate precise patterns from datasets in various domains, with a high F1 score of 97.2%, exceeding the state-of-the-art baseline. In addition, according to our experiment on five datasets with anomalies, RIOLU can automatically estimate a data column's error rate, draw normal patterns, and predict anomalies from unlabeled data with higher performance (up to 800.4% improvement in terms of F1) than the state-of-the-art baseline, even outperforming ChatGPT in terms of both accuracy (12.3% higher F1) and efficiency (10% less inference time). A variant of RIOLU, with user guidance, can further boost its precision, with up to 37.4% improvement in terms of F1. Our evaluation in an industrial setting further demonstrates the practical benefits of RIOLU.**

***Index Terms*—Pattern anomaly detection, Pattern-based data profiling, Unsupervised learning, Supervised learning**

I. INTRODUCTION

Data are an essential part of modern software [1]. In particular, data-centric [2] or AI-enabled software systems [3] incorporate massive data to provide querying or intelligent services to users. Many studies have discussed the importance of data quality in software engineering [4]–[9]: poor data quality can harm software engineering efforts. In particular, prior studies have stressed the importance of data quality assurance for the overall software quality [10]–[12]. For example, a recent study discusses the data inconsistency issue in mobile apps which can confuse users and cause app quality degradation [10]. According to an industrial investigation by Stonebraker and Rezig, data preparation could take over 80% of the development time when executed manually [13]. To reduce the labor cost of data preparation, companies and researchers aim to find automatic ways to understand the quality and detect anomalies in their data.

*Corresponding author.

Data pattern-related anomalies (or pattern anomalies) are a major type of flat structural data anomaly [14], [15] and are widely studied by researchers [14]–[17], given their hard-to-detect and hard-to-debug nature [17]. Unlike log anomalies [18] [19] and other time-series anomalies [20] [21] [22] which describe abnormal system events or status and typically require feature sets for anomaly detection, data pattern anomalies describe anomalies in the data structures themselves which may arise in features. They occur when a record violates predefined or implicit patterns [14], [15]. Prior work [23] defines data patterns as *regular expressions (regexes) that succinctly describe the syntactic variations in the strings*. For example, a data pattern for a dataset of dates could be $\backslash d\{4\}$-$\backslash d\{2\}$-$\backslash d\{2\}$. While pattern-based **data profiling** focuses on data description with patterns, pattern-based **anomaly detection** leverages the patterns to detect anomalies (i.e., data instances that violate the data patterns) [24]–[26]. However, two critical *challenges* face automated data profiling and pattern anomaly detection: (1) How can we automatically infer data patterns without prior data format knowledge, given its infinite possibilities? (2) How can we ensure the health of the inferred patterns (i.e., avoid anti-patterns that cover anomalies themselves)? This work addresses these challenges and provides an automated, unsupervised, and automatically-parameterized solution for pattern inference and anomaly detection.

Prior work has proposed different approaches for pattern-based data profiling [23], [24], [27] (i.e., pattern inference) and anomaly detection [17], [25], [26], [28]. These approaches use either *declarative* (i.e., manually designed) or *automatically inferred* regular expressions to capture the patterns in the data [17]. Although declarative tools such as Google's TFDV [24] and Amazon's Deequ [25] are straightforward to use, manually designing the regular expressions for each data column is still very effort- and time-consuming [17]. To reduce human effort, researchers designed tools that can automatically draw patterns from the data and identify anomalies. For example, XSystem [27] split the structure into three layers of representation (i.e., branch, token, symbol) to incrementally extract regular expressions from a provided table. FlashProfile [23] clusters the records by their syntactic similarity and assigns patterns for each cluster; it then filters anomalies by dropping low-frequency (e.g., less than 1%) patterns, after synthesizing column patterns. Auto-Validate [17] treats pattern quality as a global notion and ensures the generalizability of a pattern

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00078

by learning it from one column and testing its generalization ability on other similar columns. However, these tools still suffer from two shortcomings: (1) parameter configuration that requires domain expertise: for example, XSystem requires a predefined number of branches for pattern generation, which necessitates prior knowledge of the number of valid patterns; (2) insufficient precision: for example, the similarity-based approach used by FlashProfile cannot distinguish records that are seemingly close to each other, leading to inaccurate patterns.

To overcome the difficulty of parameter configuration and improve the precision, we propose RIOLU (Regex Inferencer autO-parameterized Learning with Uncleaned data), a pattern inference and anomaly detection approach that can be adaptively parameterized. Inspired by Auto-Validate [17], RIOLU regards high-quality patterns as patterns that can cover adequate healthy samples; inspired by Potter's wheel [26], RIOLU selects the most expressive and concise patterns. Our work uses statistical heuristics and K-Means clustering to automatically infer parameters, thus avoiding manual parameter setting. Moreover, the approach is built using a rule-based progressive structure which enhances the pattern quality. To assess the usefulness of RIOLU, we address the following two research questions (RQs) in this study:

RQ1: How well can RIOLU profile data in different domains? Deriving precise patterns is a prerequisite for pattern anomaly detection. This RQ aims to assess RIOLU's ability in deriving patterns for data description from all records (*a.k.a.*, data profiling). Our results show that RIOLU outperforms FlashProfile, a state-of-the-art open-sourced data profiler, and GPT-3.5 Turbo, on data in various domains.

RQ2: How precisely can RIOLU detect pattern anomalies? Pattern anomalies are common in uncleaned data. This RQ assesses RIOLU's ability to learn patterns and detect anomalies in such data. We observe that RIOLU can effectively detect pattern anomalies in public datasets and an industry setting, outperforming state-of-the-art baselines.

Our work makes the following main contributions:

- We proposed a fully automated approach, RIOLU, for data pattern inference and anomaly detection without supervision or parameter configuration.
- We demonstrated that a human-guided variant of RIOLU (Guided-RIOLU) can further improves the performance of the fully automated version (i.e., Auto-RIOLU[1]).
- We evaluated Auto-RIOLU on real-world commercial data, with practitioners confirming its ability to generate precise patterns for data quality verification and anomaly detection.

Practitioners and researchers can leverage RIOLU to understand their data and improve data quality, reducing their time and effort in data preparation or data quality assurance. Our data and code are published in a replication package [29].

II. Background

RIOLU targets pattern-based data profiling and anomaly detection. This section first defines *patterns* in our context, followed by the definitions of the two pattern-based tasks.

A. Patterns

Following Song and He [17], we define patterns as structured sequences containing basic components. In English-based patterns, basic elements include upper letters, lower letters, digits, and symbols [17], [30], [31]. According to Ilyas *et al.* [27], symbols serve as a natural delimiter in splitting the patterns. For example, *2024-Mar-01* is a string containing all these pattern elements and can be represented as "digit{4}symbol{1}upper{1}lower{2}symbol{1}digit{2}". Such patterns can be used to match or describe a data domain.

B. Pattern-Based Data Profiling

The goal of pattern-based data profiling is to distinguish patterns in the records by describing the data using derived regular expressions. Compared to previous pattern-based data profiling tools such as Microsoft's SQL Server Data Tools (SSDT) [32] and Ataccama One [33], which heavily relies on a small predetermined set of atoms, FlashProfile [23] uses a bottom-top technique to cluster records through syntactic similarity and then extract patterns from each cluster; it requires no user-defined domains or base patterns (e.g., dates, phone numbers) to reach a high precision. However, we noticed that the tool may not be robust when the differences between records are subtle. For example, a column of state/province abbreviations should only contain two upper letters (i.e., [A-Z]{2}). Due to poor data quality, the column may also contain noisy records with lower letters (e.g., Qc, on) or incorrect string lengths (e.g., M, ABB). Given the small syntactic distances between different spelling types, they may be identified as one cluster, and the profiling result would be "[a-zA-Z]+", which fails to precisely describe the underlying pattern.

C. Pattern Anomaly Detection

Pattern anomalies indicate records that violate pre-defined or inferred regex-like patterns [14] [15]. Following Huang and He [30], we define pattern anomaly detection as the process of filtering inconsistent data (i.e., anomalies) using regular expressions. As introduced in Sec I, existing tools provide automatic inference approaches for pattern anomaly detection. Although these tools have largely reduced human labor, they always have the problem of requiring domain-dependent parameter settings. For example, XSystem [27] requires the user's prior knowledge of the number of branches (i.e., number of acceptable pattern types), which could be hard to fix at the first use and may not be constant across domains (e.g., a column for "id" can only accept one branch as $\backslash d\{5\}$, while a column for "date" may accept both patterns of $\backslash d\{4\}$-$\backslash d\{2\}$-$\backslash d\{2\}$ and $\backslash d\{4\}/\backslash d\{2\}/\backslash d\{2\}$, written as 2024-01-01 and 2024/01/01, respectively, as valid). FlashProfile [23] require users to define the threshold for low-frequency patterns (i.e., a pattern with a frequency lower than the threshold is considered as abnomal). However, a threshold may not hold for different datasets, and it is difficult for users to determine a solid threshold without fully comprehending the dataset.

[1]We use the name Auto-RIOLU to refer to the fully automated version, in order to distinguish it from the Guided-RIOLU version.

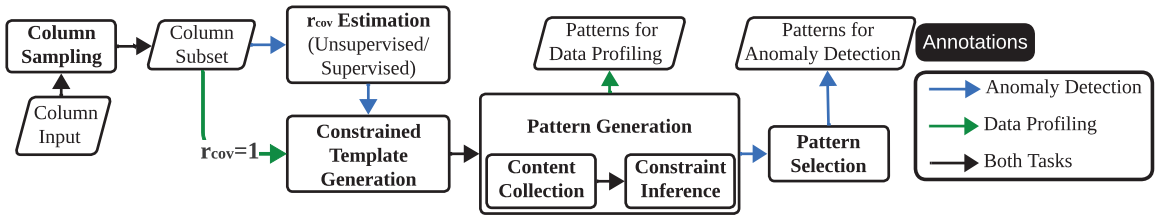

Fig. 1. An overview of RIOLU's structure.

III. Approach

A. Overview

The input of RIOLU is a two-dimensional table with random numbers of columns and rows. The structure of each column is undefined: for example, they can be names, IDs, or URLs. The goal of RIOLU is to automatically derive the patterns (e.g., YYYY-MM-DD) of each column without prior knowledge; the patterns can then be used to detect data anomalies (e.g., invalid URLs). As shown in Fig 1, the approach of RIOLU for determining patterns and detecting anomalies in a column can be decomposed into the following steps:

- **Column Sampling**: Sample a subset of data from the column to generate the patterns.
- **Coverage Rate (r_{cov}) Estimation**: Estimate the percentage of healthy values (r_{cov}) in each column.
- **Constrained Template Generation**: Generate raw templates for each record with a granularity constraint.
- **Pattern Generation**: Generate pattern constraints for each template according to the coverage rate.
- **Pattern Selection**: Select patterns based on some heuristics (e.g., their generalizability).

As shown in Fig 1, all five steps are needed for the anomaly detection task. On the other hand, the data profiling task only involves the steps of column sampling, constrained template generation, and pattern generation, as all data are assumed to be healthy for this task (i.e., r_{cov} is constantly 1). Estimating the portion of healthy data (i.e., r_{cov}) is an essential process in generating patterns for anomaly detection, as healthy patterns should only be learned from healthy data. Since automated coverage rate estimation depends on template generation, pattern generation, and pattern selection, we introduce this step after explaining the other three steps.

B. Column Sampling

Typically, a data column (e.g., a column about dates) only covers a subset of the entire data domain (all possible values of dates) that may come through the data pipeline. Ideally, high-quality patterns should not only cover all the samples in the available data that can be seen during the pattern generation process, but also be able to generalize to unseen future data [17]. Therefore, we use a subset from the column as the pattern generation subset and use the whole data column for generalizability evaluation and pattern selection. To ensure the representativeness of the sample and the pattern generation efficiency, we use a sample size N_{tr} which is calculated using a z-score with a confidence level of 95% and an error margin of 5%; N_{tr} samples are randomly drawn from a column whose size is N without replacement.

C. Constrained Template Generation

Template generation without constraints may result in an over-abundance of templates. Indeed, the basic structure of patterns can be captured by raw templates that contain **TOKEN**s and **delimiters**. We could finely split all records using all of their symbol strings (i.e., symbols containing no tokens within, such as "++" in Fig. 2) as delimiters. However, overly fine-grained splitting of the records may result in under-generalization, a problem in anomaly detection [31]. To prevent under-generalization, we establish an **exact matching rate** r_{EM} to control the granularity of raw templates. r_{EM} controls the portion of records that need to be fully split, thus determining the required maximum number of delimiters and the granularity of raw templates. As illustrated in the right part of Fig 2, fully splitting all records (i.e., r_{EM} = 1) results in the last two templates matching only one record. The scattered raw template distributions may cause the generated patterns to have low frequency, leading to a biased result in the pattern selection step (see Section III-E).

Constrained template generation aims to convert records to raw templates under the constraint of r_{EM}. For the data profiling task, r_{EM} is fixed to 1, as we aim to capture all template structures and do not require pattern selection. For data anomaly detection, we should only obtain templates for the exact matching of the healthy data. Hence, we set the exact matching rate with the same value as the estimated coverage rate (i.e., the estimated percentage of healthy data, see III-F): $r_{EM} = r_{cov}$, with the intuition that all the healthy data would have exact template matching. Fig 2 is an example of a datetime column with five records from a sampled set ($N_{tr} = 5$). First, we calculate the number of records to be fully split using r_{EM}. The number shall be calculated with $N_{tr} * r_{EM}$ and rounded to an integer. In our example, when setting r_{EM} to 1, we should fully split all five samples; when $r_{EM} = 0.8$, four samples are to be fully split. When $r_{EM} < 1$, we determine the samples to be fully split using the number of delimiters they contain. According to the minimum description length principle [34], we capture the minimum number of needed delimiters to fulfill the r_{EM} constraint. The numbers of delimiters in the example are $\{4, 6, 4, 4, 5\}$. Hence, when $r_{EM} = 0.8$, we accept five as the maximum number of acceptable delimiters, as it is the minimum number of delimiters required to fully split four samples.

2011-10-01T00:00:00
2019-10-02T00:00:00 GMT++5
2016-10-01T00:00:00
2011-12-06T00:00:00
2019-12-20T00:00:00 AM

r_{EM}	Max-D	Templates
1	6	T-T-T:T:T T-T-T:T:T T T-T-T:T:T T++T
0.8	5	T-T-T:T:T T-T-T:T:T T

Fig. 2. An example of 5 date time records and the templates generated under different exact matching rates (r_{EM}). T in templates represents tokens. Max-D stands for the maximum number of delimiters accepted under certain r_{EM}.

After determining the maximum number of acceptable delimiters, we generate raw templates by iterating through all the records. For records containing less or equal to the maximum number of delimiters, we fully split them and take all their symbol strings as delimiters. Otherwise, we stop splitting when the maximum value is reached. Under the setting of $r_{EM} = 0.8$, the second record contains six delimiters and will be split as "T-T-T:T:T T" (i.e., matching the template of the last record). The portion "T++T" in the raw template is merged as a "T" after reaching the maximum delimiter number.

D. Pattern Generation

Each record is matched to a raw template after the constrained template generation stage. We further elaborate on the content constraints and form patterns based on the templates. Raman *et al.* [26] use **token range** and **token length** to constrain the tokens. Instead of using the tokens, Ilyas *et al.* [27] split the tokens into characters and discuss whether the character slot contains specific **static characters** or covers one or more **character type**. Inspired by these approaches, we consider these four types of constraints for pattern generation: token range, token length, static character, and character type.

We use a waterfall structure to infer the constraints at the four layers, with an order from stricter to looser: 1) token range determination, 2) token length determination, 3) static character determination, and 4) static type determination. The intuition is simple: if a token has a stricter constraint (e.g., with a token range constraint: being either "AM" or "PM"), then we do not need to infer the looser constraints (e.g., then token length or character-level constraints). We iterate records in the sampled set to collect all the contents on the four layers and select content constraints using r_{cov} as a threshold. We use Fig 3 as a running example and assume $r_{cov} = r_{EM} = 0.8$.

Content Collection. Fig 3 illustrates five date time records that match with a template "T-T-T:T:T". Each token in the template is matched to a string in the records. For example, the first "T" in the first record corresponds to "2011". Following this method, we collect the contents of each token from the five examples, as shown in the "Tokens" frame. These contents are used for token range and token length constraint inference.

The tokens can be further decomposed into character slots for static character and character type inference. In Fig 3, we decompose the third token as an example. The character collection is similar to token content collection: we iterate through the tokens and fit their contents into the corresponding slot position, as shown in the "Character Slots for T3" frame.

Constraint Inference. When selecting content constraints, Auto-Validate [17] aims to find patterns that capture most of the values under a small tolerance value θ (e.g., 1%, 5%) for anomalies in the training set. However, we argue that using a constant predefined θ is hardly precise. Instead, the value should be adjusted based on the specific error rate of each dataset. Therefore, we estimate the percentage of healthy values r_{cov} to guide the constraint selection stage for the anomaly detection task. As mentioned in Sec III-A, for data profiling, the coverage rate is 1, given that the goal is to describe all data fully.

*1) **Token Range Determination**:* Constant token contents (e.g., "AM" or "PM") are highly frequent [23]. We use a two-class K-Means clustering technique to cluster the token values based on their frequency. We choose K-Means due to its efficiency and wide use for data clustering. The frequency range that can be reached by a token is: $[\frac{1}{N_{tr}}, 1]$. To ensure that two clusters (i.e., the high and low-frequency cluster) can always be created, we manually insert 1 and $\frac{1}{N_{tr}}$ into the frequency list. The frequencies in the high-frequency cluster are then summed to compare with the coverage rate r_{cov}: if the sum is larger than r_{cov}, then the high-frequency values are making an adequate match, and thus a token range is found. Otherwise, we shall further determine whether the token has a length constraint. In our date time example, the template contains five tokens. As shown in Fig 3, all the values for T_4 and T_5 share the value of "00": the token value "00" has a frequency of 1. Hence, a token range is assigned to these two tokens. Conversely, T_1, T_2, and T_3 do not have any token range since each token value has a frequency of 0.2, and is clustered with $\frac{1}{6} \approx 0.167$ (i.e., the low-frequency cluster).

*2) **Token Length Determination**:* While a specific content range may not be applicable for a token, the token may have a limitation to the length. A static length len for a token is determined if and only if most token contents have a fixed length; if the system fails to detect a static length for the current token, the minimum token length is captured as len_{Min}. In our running example in Fig 3, we should detect token length constraints for T_1, T_2, and T_3. All the contents for T_1 and T_2 are in a fixed length, while T_3 contains four (80%) 5-character contents and one (20%) 4-character content. Under the setting of $r_{cov} = 0.8$, we determine that T_3 also has a token length constraint since 80% of the contents satisfy the constraint of having 5 character slots.

*3) **Static Character Determination**:* We use K-Means clustering based on frequencies to determine static characters in the pattern, similar to the strategy for token range determination. The frequencies are calculated by counting the existence frequency for a character on a certain slot (e.g., for the first character slot in T_3, the frequency of "0" is 0.6 while the frequency for "1" is 0.4). We also insert 1 and $\frac{1}{N_{tr}}$ into the frequency list to ensure the split. For static length tokens (i.e., with len), we detect static character ranges for all len characters; for tokens without static length constraints, we detect static character ranges for the first len_{Min} characters. We decompose T_3 in Fig 3 to demonstrate the approach. The

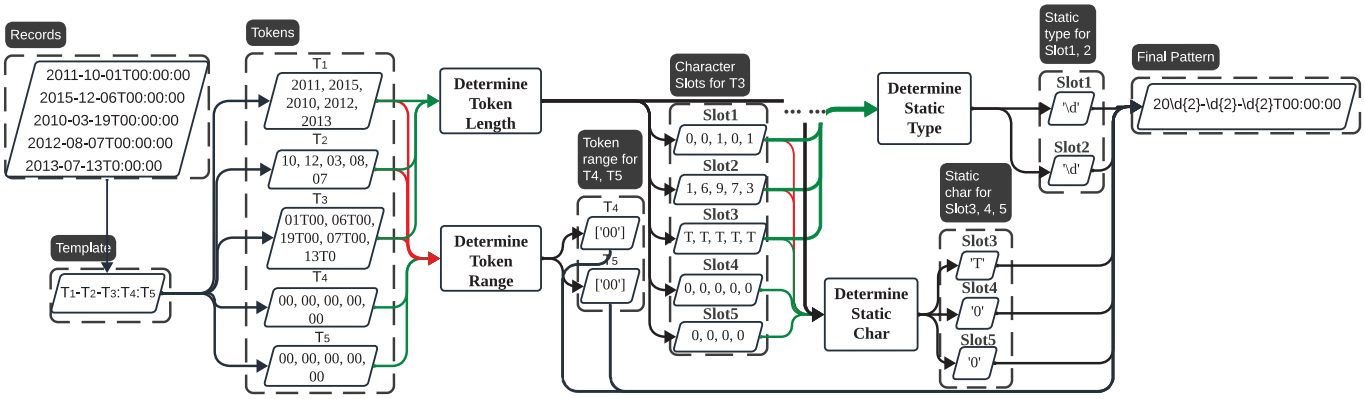

Fig. 3. A date time example of using RIOLU generating patterns ($r_{cov} = 0.8$). "T" stands for tokens. The green paths indicate a constraint has been found for the input (i.e., token or character slot) on the corresponding layer, while the red paths suggest that the input does not have a constraint on this layer.

constant length constraint is 5 for this token. According to the collected character in the "Character Slots for T_3" frame, the last three slots have static characters, but no static character is detected for the first two slots.

*4) **Static Type Determination**:* Characters are usually categorized into upper/lower letters, digits, and symbols. Static type is to be detected if and only if when a range of static characters (i.e., a stricter constraint) fails to be determined. For each character slot, the character types are ranked according to their frequencies from high to low, and the types are selected until their cumulative frequency exceeds r_{cov} (i.e., the types cover the majority of the space). If all types appear in the majority of current character slot, the slot would have no constraint. Following the intuition in static character determination, for static length tokens, the type constraint applies to all len characters; for tokens without static length constraints, we detect character types for the first len_{Min} characters. For the example token T_3 in Fig 3, since the first two slots in T_3 do not have static characters, we check their static character type. Both slots have a static type of digits, written as "\d".

*5) **Pattern Composition**:* After constraints on the four layers are detected, we compose them into a pattern. For tokens with token range constraints, we directly replace the token using the range (e.g., the last two tokens in Fig 3 are replaced by "00"). Otherwise, we write the regular expression for the token using detected constraints. Take T_3 as an example: using the "\d" constraints for the first two character slots and the "T00" constraints for the last three slots, the regular expression for T_3 is "\d{2}T00". Following this procedure, a pattern is generated as illustrated in the "Final Pattern" frame in Fig 3.

E. Pattern Selection

A pool of candidate patterns is constructed after the pattern-generation step. However, not all patterns are considered as healthy patterns. As shown in Fig 1, we further perform pattern selection for the data anomaly detection task. Similar to Padhi *et al.* [23], we consider patterns with low matching rates on the dataset as anomalies. For each pattern p_i, its matching rate (i.e., frequency) on the whole dataset is calculated through the number of records matched using regex. We then apply K-Means clustering to split the patterns into two clusters based on their matching rate automatically [35]: the **high-frequency** and **low-frequency** clusters. The clustering technique could avoid the domain-dependent threshold determination problem raised in Sec II-C. We assume that at least one pattern shall be accepted for the corresponding column. Thus, we insert a low matching rate noise (i.e., $\frac{1}{N}$) to ensure a low-frequency cluster is created. Patterns labeled as **high-frequency** are selected as healthy patterns, whereas those labeled as **low-frequency** are not further used. Finally, the healthy patterns are used to detect anomalous records in the column: records that do not match any healthy pattern are identified as anomalies.

F. Coverage Rate (r_{cov}) Estimation

Coverage rate r_{cov} is essential in template generation and pattern generation. Similar to column sampling (Sec III-B), for the sake of generalizability and efficiency, we use a statistically representative sample (of size N_{tr}) of a data column to estimate the corresponding r_{cov}, either in a supervised or an unsupervised way.

Guided-RIOLU: Supervised Estimation. With human involved, it is feasible to notice pattern inconsistency in the sampled data and label them with domain knowledge. Users are required only to label whether a record in the sampled set (of size N_{tr}) is a pattern anomaly in the column. The labeled data can be used to calculate the portion of healthy data, which is the portion of data to be covered (r_{cov}).

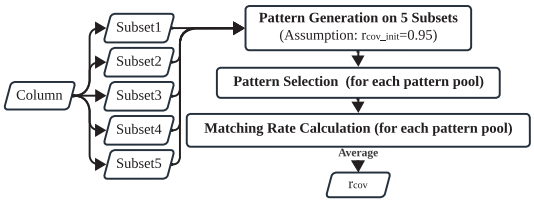

Fig. 4. The coverage rate estimation process in unsupervised approach.

Auto-RIOLU: Unsupervised Estimation. Our intuition for pattern anomalies is that these anomalies cannot form large pattern clusters, similar to Padhi *et al.* [23]. Conversely,

healthy patterns are highly frequent and robust enough to form a large cluster. As illustrated in Fig 4, we apply a three-step approach to estimate the coverage rate. We first randomly draw five subsets (i.e., $N_{subset} = 5$) from the considered data column to ensure measurement stability, each with size N_{tr}. Then we generate five initial pattern pools from these five subsets with the assumption of $r_{cov\_init} = 0.95$, following the steps described in III-C and III-D. We choose $r_{cov\_init} = 0.95$ as the initial overage rate since prior work shows that the anomaly data rates are typically less than 5% [17]. We further evaluate the sensitivity of our approach for the settings of the r_{cov_init} value and the N_{subset} value in IV-C. In the second step, we select patterns with high matching rates from the five initial pattern pools through the pattern selection process (as described in III-E). In the last step, each selected pattern pool is used to calculate the portion of matches they can create (i.e., matching rate) on the whole dataset. The matching rates provide us with an estimation of the portions of large clusters (i.e., potentially healthy clusters); noise patterns provide few matches and will be dropped in the pattern selection stage. Using five subsets can statistically suppress the randomness of the evaluation process. Hence, we take the average matching rate as the estimated r_{cov}.

IV. Experiments

This section describes our experiment design and results for evaluating RIOLU through our two research questions. RQ1 evaluates RIOLU's capacity to produce high-quality patterns, in a pattern-based data profiling setting. RQ2 evaluates RIOLU's capacity to detect pattern anomalies from uncleaned data in an unsupervised or semi-supervised manner.

A. RQ1: How well can RIOLU profile data in different domains?

Patterns extracted for data profiling should adequately and precisely describe the corresponding data domain. Further, it does not assume that the dataset is dirty in nature. Hence, the coverage rate to be reached in data profiling tasks shall be 1, and all the generated patterns should be accepted to describe the domain. Through the data profiling quality analysis, we could gain insight into RIOLU's pattern generation capability.

*1) **Dataset**:* We choose a set of public datasets [36] used to evaluate data profiling in prior work (FlashProfile [23]). To ensure generalizability, we consider the entire *DOMAINS* data with 63 datasets with 45 to around 20k records, from a variety of domains (e.g., amino structure, software configuration, IP address); the files, along with detailed domain information, are listed in our replication package. Using these datasets allows us to have a fair comparison with prior work [23].

*2) **Methods Compared**:* We compare RIOLU with the state-of-the-art open-source tool for data profiling (FlashProfile [23]), as well as ChatGPT which has recently shown promising results for regex generation [37].

- **ChatGPT** [37]. LLMs have shown their power in various tasks including regex inference [38]. We specified the pattern-based data profiling task to the GPT-3.5 Turbo API and obtained the responses, as it was one of the most powerful and affordable at the experiment time. The regexes are manually extracted from the responses. We provide the prompt template and responses in the replication package.
- **FlashProfile [23]**. FlashProfile is the state-of-the-art open-sourced tool for pattern-based data profiling. FlashProfile clusters record using syntactic similarity and draw a pattern for each cluster. It significantly outperforms previous state-of-the-art tools, including MicrosoftSSDT [32] and Atacama One [33].
- **RIOLU**. Data profiling does not require estimating the coverage rate, given that the patterns should cover all data. To use RIOLU for data profiling, we set the coverage rate r_{cov} to 1 and skip the pattern selection step.

TABLE I
THE AVERAGE PERFORMANCE OF CHATGPT, FLASHPROFILE AND RIOLU FOR DATA PROFILING (TP: AVERAGE TRUE POSITIVE RATE; FP: AVERAGE FALSE POSITIVE RATE; P: AVERAGE PRECISION; R: AVERAGE RECALL; F1: AVERAGE F1 SCORE).

	TP	FP	P	R	F1
ChatGPT	82.6%	0.57%	90.6%	82.6%	86.4%
FlashProfile	93.2%	2.3%	97.8%	93.4%	96.2%
RIOLU	95.1%	0.16%	99.4%	95.1%	97.2%

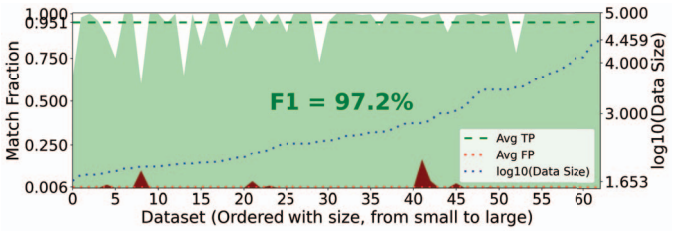

Fig. 5. RIOLU's profiling quality on FlashProfile-DOMAINS dataset. The white peaks in the upper part denote the false negative rate, and the red peaks in the lower part indicate the false positive rate.

*3) **Evaluation and Metrics**:* The goal of data profiling is to extract patterns that can fully and solely describe a domain. We followed the evaluation settings used in prior work [23]. 20% of the data from each dataset is randomly extracted as the pattern generation dataset. While prompting ChatGPT, we observed that, for 7 datasets containing many (i.e., ∼10k) records, or long content (e.g., protein structures), a random sample of 15% of the data exceeded the token limit of ChatGPT. Thus, we only use 10% of these datasets for training data. The *true positive rate* under this scenario is the portion of matches produced by the pattern pool in the remaining 80% (or 90% for 7 datasets with ChatGPT) data from the same domain. An equal number of records are randomly drawn from other domains, and the *false positive rate* is the matching rate on records belonging to the other domains. The intuition is that patterns drawn from a dataset shall describe all the records in this domain while not over-generalizing to other domains. We also calculate the *precision*, *recall*, and *F1 score* based on these metrics.

*4) **Evaluation Result**:* Table I shows the average performance of RIOLU and the baselines for data profiling. Fig 5 illustrates RIOLU's profiling quality in the same manner as

FlashProfile [23] does, showing the detailed performance for each individual dataset. As reported in [23], FlashProfile has an average true positive rate of 93.2%, an average false positive rate of 2.3%, with an average F1 score reaching a high value of 96.2%. According to Table I, RIOLU pushed the accuracy higher with an average F1 score of 97.2%. The average true positive rate obtained by RIOLU is around 95.1%, while the false positive rate is 0.16% (less than 10% of that of FlashProfile). On the other hand, the overall performance of ChatGPT is worse than both FlashProfile and RIOLU: the average F1 score is only 86.4%. In particular, ChatGPT's false positive rate is about 3.6 times that of RIOLU. Besides, ChatGPT generated one pattern that encountered issues during regex compiling (i.e., bad character range issue in a piece of regex "[9-100]"), and 5 patterns that failed to match any record in the domain (e.g., "[A-Fa-f0-9]{24}" for code records with only 22 characters). The results indicate that the patterns created using our raw templates and water flow constraint selection approaches can accurately capture pattern features for each domain and distinguish them from other domains. Moreover, we noticed that data-driven pattern profiling methods tend to be more accurate than ChatGPT.

We also observe a general trend of improved performance of RIOLU when the data size increases in Fig 5. This is because a small data size may fail to sample adequate supportive records as all the patterns may have a low frequency. Therefore, these patterns cannot be accurately captured. When the data size increases, the trend of highly frequent patterns becomes significant, and thus, RIOLU can learn more robust patterns.

Summary. RIOLU performs well on the data profiling task on 63 datasets from various domains. In particular, the false positive rate of RIOLU is less than 10% of that of the state-of-the-art baseline (FlashProfile) and about 28% of ChatGPT's, indicating that Auto-RIOLU can describe the data more precisely and avoid over-generalization, which is essential for data anomaly detection.

B. RQ2: How precisely can RIOLU detect pattern anomalies?

To answer this question, we evaluate RIOLU's capacity for detecting data pattern anomalies using three sets of data: open-sourced tabular data, Java project method names, and commercial data. We first compare Auto-RIOLU and Guided-RIOLU with the baselines on five open-sourced tabular datasets. We also explore the method naming patterns in popular Java projects and examine the inconsistencies using Auto-RIOLU. On the commercial dataset, we predict patterns using Auto-RIOLU in different domains and ask the data analysis team in our industry partner CompanyX to validate the patterns.

Evaluation on Open-source Datasets.

*1) **Dataset**:* To estimate the anomaly detection performance of RIOLU, we collected five publicly available datasets in different domains and with various sizes. These datasets contain multiple types of errors, including pattern violations and other types of data quality issues. We obtained the original data and their cleaned versions from previous data quality studies [15] [16] [39] [40], and selected the columns containing pattern anomalies. Table II shows the data size, error rate, and description of each dataset. The data columns' domains include date, time, postal code, phone number, etc.

TABLE II
THE INFORMATION ABOUT THE FIVE PUBLIC DATASETS. THE SIZE OF EACH DATASET IS WRITTEN AS ROWS*COLUMNS.

Dataset	Size	Error Rate	Description
Hosp-1k	999*3	23.9%	Hospital data from US Department of Health& Human Services.
Hosp-10k	10000*3	23.2%	
Hosp-100k	100000*3	23.7%	
Flights	74066*6	52.5%	Flight data from 38 airline-related websites.
Movies	7390*4	0.2%	IMDb film data.

*2) **Methods Compared**:* We compare Auto-RIOLU and Guided-RIOLU with ChatGPT and two versions of FlashProfile, the state-of-the-art open-sourced approach.

- **ChatGPT**. Constrained by the token limit of ChatGPT, we randomly sample 3000 records for each column and present them to GPT-3.5 Turbo API for regex inference. For certain columns with a larger number of tokens, we fed 2500 records or 2000 records to meet the token limit. The inferred regexes are then used for anomaly detection on the entire dataset. The sampled dataset and zero-shot prompt template can be found in our replication package.
- **FlashProfile** [23]. As mentioned in IV-A, FlashProfile clusters data with different patterns and infers regular expressions for each cluster. Pattern anomalies can be found using a frequency threshold: patterns that are rare tend to be anomalies. We follow the suggestion provided in the instructions [41] and set the threshold to 0.01.
- **D-FlashProfile (Dynamic-FlashProfile)**. D-FlashProfile is a combination of FlashProfile with our automated pattern selection step (see Sec III-E). Based on our pattern selection step, we avoid a fixed threshold but dynamically select the patterns based on their frequencies. The cluster with a higher frequency is accepted as the healthy pattern pool.
- **Auto-RIOLU**. Auto-RIOLU learns and selects patterns using automated coverage (r_{cov}) estimation on unlabeled data (Sec III-F).
- **Guided-RIOLU**. Guided-RIOLU estimates r_{cov} according to the error rate of a labelled subset of the data (Sec III-F).

We also considered XSystem [27] for evaluation. However, given that the system heavily relies on determining the number of branches (i.e., the number of different patterns) which requires domain expertise, we failed to obtain reliable patterns. Thus, we excluded the tool from the comparison.

*3) **Evaluation and Metrics**:* As anomalies have the nature of being rare in the wild, the healthy and anomaly records are often imbalanced: an approach that constructs a broad pattern and predicts only the majority class (e.g., predicting every record as "normal" using ".*") can achieve high accuracy without actually detecting any anomalies, which can give a biased impression of the performance. Thus, we do not evaluate the accuracy. Anomaly detection aims to identify all the anomalies while not misidentifying any healthy data. Hence, we leverage *precision*, *recall*, and *F1 score* for the evaluation. Precision indicates the percentage of predicted

TABLE III
THE EXPERIMENT RESULTS FOR QUANTITIVE ANALYSIS ON FIVE DATASETS. (P: PRECISION; R: RECALL; F1: F1-SCORE)

	Hosp-1k			Hosp-10k			Hosp-100k			Flights			Movies		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ChatGPT	0.617	0.435	0.510	0.667	0.280	0.385	1.0	0.379	0.550	0.958	0.594	0.733	1.0	0.637	0.778
FlashProfile	0.333	0.036	0.065	0.667	0.070	0.126	0.366	0.044	0.079	0.850	0.641	0.731	1.0	1.0	**1.0**
Dynamic-FlashProfile	0.333	0.123	0.181	0.667	0.170	0.271	0.667	0.157	0.254	0.706	0.658	0.681	1.0	1.0	**1.0**
Auto-RIOLU	0.712	0.536	0.611	0.8	0.35	0.487	0.8	0.348	0.485	0.790	0.692	0.738	1.0	1.0	**1.0**
Guided-RIOLU	0.66	0.758	**0.706**	1.0	0.502	**0.669**	1.000	0.489	**0.657**	0.824	0.798	**0.81**	1.0	1.0	**1.0**

TABLE IV
THE INFERENCE TIME STATISTICS ON THE DATASETS (BY SECOND) OF FOUR COMPARED TOOLS.

Time(s)	ChatGPT	FlashProfile	Auto-RIOLU	Guided-RIOLU
[Min, Max]	[10.95, 27.44]	[17.08, 406.26]	[0.34, 46.43]	[0.27, 7.91]
Average	13.72	111.16	12.35	2.63

anomalies that are actually anomalies; recall indicates the percentage of anomalies that are correctly predicted; the F1 score is the harmonic average of precision and recall.

*4) **Quantitative Evaluation Result**:* We ran the experiment on a Mac desktop with an 8-core CPU (M2 chip) and 16G memory. The time used for pattern inference is reported in Table IV. The response time of ChatGPT was captured as the inference time. We did not include the execution time of using the generated patterns with regex matching to detect anomalies, since all tools use the same piece of regex matching code for the detection and the time consumption is neglectable. We did not include D-FlashProfile's inference time because this approach combines FlashProfile and our pattern selection component; the time consumption is larger than FlashProfile. Since the ground truths have been obtained from previous studies, the labeling time is excluded for Guided-RIOLU. Both versions of RIOLU require less inference time on average (Auto-RIOLU requires 10.0% less time than ChatGPT, and Guided-RIOLU requires 80.8% less).

The performance results generated by the five tools compared are shown in Table III. To suppress the bias caused by random sampling, the results for both versions of RIOLU are the average on five detection runs. According to the table, both versions of RIOLU outperform all the baselines on four out of five datasets. ChatGPT obtained high precisions and showed great potential in anomaly detection; on the *Hosp-100k* dataset, ChatGPT detected more true positives in the state code domain and obtained a higher F1 score than Auto-RIOLU.

We noticed that Guided-RIOLU can boost performance on four datasets using a set of labeled data samples, achieving the best performance on all datasets. On four datasets, Guided-RIOLU obtained up to 37.4% improvement of F1 over Auto-RIOLU. The sample size to be labeled is less than 0.4% of the whole dataset. With the labeled small subset, RIOLU can estimate the coverage rate more accurately.

It is also shown in Table III that combining our pattern selection step with FlashProfile (i.e., Dynamic-FlashProfile) enhanced its F1 score on three out of the five datasets: the automated selection of patterns can increase the quality of pattern pools on the three hospital datasets, proving the efficiency of the selection step. However, we noticed a performance drop in the *Flights* dataset. The ground truth of the *Flights* dataset contains scattered patterns (i.e., records with different patterns are accepted as ground truths). Since all the patterns do not have high coverage (i.e., with coverage around 10%), they cannot be well-clustered based on their frequency.

TABLE V
PATTERNS GENERATED BY CHATGPT, FLASHPROFILE, DYNAMIC-FLASHPROFILE(D-FLASHPROFILE), AUTO-RIOLU, AND GUIDED-RIOLU FOR FOUR EXAMPLE DATA DOMAINS.

	State Code	Zip	Phone	Duration
ChatGPT	[A-Z](?![A-Z]) [A-Z]{2}	\d{5} \d{5}(?:-\d{4})?	\d{10}	(\d+)\s*hr\.\s* (\d+)\s*min
FlashProfile	[a-zA-Z]+	[0-9]{5} [0-9]{4} [0-9]{3}	[0-9]+ [0-9]+[a-z][0-9]+ [0-9]{10}*** [a-z][0-9]{10}	[0-9]+\ min \ [0-9]{4}\ [A-Z][a-z]+"
D-FlashProfile	[a-zA-Z]+	[0-9]{5}	[0-9]+	[0-9]+\ min
Auto-RIOLU	[A-Z]{2}[A-Z]*	\d{5}	\d{10}	\d{2}\d*\min
Guided-RIOLU	[A-Z]{2}	\d{5}	\d{10}	\d{2}\d*\min

*5) **Qualitative Evaluation Result**:* We extracted the patterns generated by ChatGPT, FlashProfile, D-FlashProfile, Auto-RIOLU, and Guided-RIOLU in four domain fields to analyze the results qualitatively. The first three domains are state code, zip, and phone number from the *Hosp-100k* dataset and the duration domain from the *Movies* dataset. We examine the quality of the patterns using domain knowledge gained through observing the records. According to Table V, Guided-RIOLU can generate and select correct patterns corresponding to prior knowledge for all four domains. Given that the coverage rate is automatically estimated, Auto-RIOLU provides a false positive pattern (i.e., strings containing more than two upper letters are accepted as state code).

ChatGPT has external domain knowledge from pre-training. Hence, it inferred and selected the patterns based on both the provided column and its training data. For state code, it included a false positive regex matching any single uppercase letter that is not immediately followed by another uppercase letter. For zip code, ChatGPT accepted another format (i.e., \d{5}(?:-\d{4})) apart from five digits. The duration in the *Movies* dataset should use minutes as units instead of hours. However, ChatGPT accepted the records with hours as a unit based on its external knowledge of time calculation, leading to more false negatives in anomaly detection.

It can be observed that FlashProfile accepts several suboptimal patterns with the default frequency threshold (e.g., phone numbers shall only contain digits; however, it provides us with three patterns, including either letters or symbols). With our pattern selection module attached, the patterns learned by Dynamic-FlashProfile are more precise. Although the length constraints cannot be determined accurately due to the pre-clustering technique of FlashProfile,

Dynamic-FlashProfile eliminated patterns with significantly wrong types; for example, the three phone number patterns containing letters or symbols are dropped after clustering.

The qualitative result suggests that although Auto-RIOLU can learn constraints on character types and length constraints, the learning quality is still to be improved with a more accurate estimation of the coverage rate. The estimation can be more accurate with humans involved to label a small subset (e.g., with around 350 samples). We also found that RIOLU's accuracy can be limited when patterns are scattered (i.e., when a data column has many low-frequency healthy patterns). In such cases, RIOLU may fail to distinguish anomalies from low-frequency healthy patterns (Also see Sec VI).

Detection of Naming Inconsistency in Java Projects.

Identifier (e.g., variable or method) naming has been widely recognized as an essential software engineering task [42] [43] [44]. Indeed, inappropriate naming can cause problems in developer comprehension [45] and even code quality degradation [46]. Oracle [47] suggests that Java method names should start with an upper letter and with the first letter of each internal word capitalized.

We want to evaluate the ability of Auto-RIOLU to infer project-wise naming conventions and detect anomalies (i.e., inconsistencies) automatically, thus we do not specify any naming convention with Auto-RIOLU. To this end, we collected method names in the 11 top Java projects from the CodeAttention training set [48]. We ran Auto-RIOLU with default settings, obtained the inferred patterns, and detected anomalies. For all projects, our inferred healthy patterns match the Oracle Java method naming convention. In other words, all the detected normal names match the standard naming convention (i.e., no false negatives). Thus, we only manually analyze the detected anomalies, to understand the categories of anomalies and potential false positives. Given that the *liferay-portal* project contains more than 20k detected anomalies, we randomly sampled 379 anomaly examples (i.e., 95% confidence level with 5% error rate) for labeling. Table VI shows the detection and labeling results. The labels can be found in our replication package.

As shown in Table VI, 9 out of the 11 projects have a consistent naming rate higher than 95%. There are two categories of naming anomalies (true positives): 1) method names containing symbols such as "_" and "$", and 2) method names starting with an uppercase letter or a digit. For the project with the lowest consistent naming rate (i.e., liferay-portal), 70.6% of the detected anomalies contain symbols. The detailed statistics can be found in our replication package.

For 9 out of the 11 projects, Auto-RIOLU's false positive rates are relatively low, ranging from 0 to 29.2%. For one project (liferay-portal), the false positive rate in the detected anomalies is 50%, which may be caused by its low anomaly rate: there are only 4 detected anomalies. Among these projects, the false positives are mainly caused by short names (e.g., "go", "f"): Auto-RIOLU tends to treat such short names as anomalies since typical Java method names tend to have more than three characters.

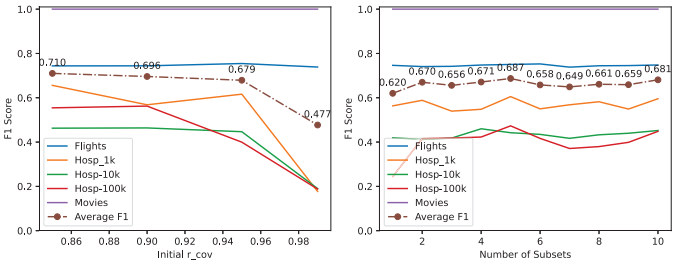

Fig. 6. The impact of choosing different r_{cov_init} values (0.85, 0.9, 0.95, and 0.99) and N_{subset} (range from 1 to 10) for Auto-RIOLU.

Evaluation on CompanyX database.

The data pattern inconsistency would block the automatic software pipeline and lead to software quality problems. Our industrial partner *CompanyX* uses a large data warehouse to manage data from various sources and domains; the data are produced by some software (sub)systems (e.g., web applications) and consumed by other (sub)systems (e.g., data analytics pipelines). *CompanyX* faces the problem of data patterns not being aligned. The data pattern inconsistency would block the automatic software pipeline and lead to software quality problems. However, given the large volume of data, it is a great challenge to design all the regular expressions for each data domain manually. We deployed RIOLU on the data warehouse management platform (Databricks) and evaluated its ability in generating patterns and detecting potential anomalies. We performed our evaluation using three representative large tables (i.e., tables containing over 20 columns and 500k records). Before executing the tool, we cooperated with the data analysis team to determine columns in target tables that require pattern validation. According to our discussion, the columns to be checked cover various domains, including date, ID, emails, province code, etc. The error rate of each column remains unknown before our detection. We then used Auto-RIOLU with the default setting to automatically generate the data patterns and detect potential anomalies for these columns. The team manually verifies the validity of each generated pattern by inspecting records in the corresponding columns. Auto-RIOLU can generate robust patterns for most columns and detect inconsistent data records (i.e., anomalies). For instance, based on Auto-RIOLU's results, we noticed that 0.15% of liability codes are "Unknown", which may need the production team's attention; account IDs should contain a fixed number of digits, but 1.5% of the records have abnormal patterns and require further investigation. We observed that only email pattern generation had been complex, according to the heterogeneity of email addresses (e.g., john.doe@school.edu and johndoe@school.com can both be valid addresses, but they belong to different templates due to the symbol "." existing in the first email address). Future work is needed to improve RIOLU for data with such such heterogeneous patterns.

C. Ablation Study & Sensitivity Analysis

Several design choices and parameters may affect the performance of RIOLU. Thus, we carry out an ablation study and a sensitivity analysis to discuss their impacts.

TABLE VI
THE METHOD NAMES CONSISTENT RATE (I.E., THE PORTION OF NAMING MATCHED WITH THE INFERRED PATTERN), THE NUMBER OF ANOMALIES DETECTED, AND THE FALSE POSITIVE RATE.

Project Name	presto	wildfly	libgdx	intellij-community	gradle	liferay-portal	spring-framework	cassandra	hibernate-orm	hadoop-common	elasticsearch
Consistent Rate	0.998	0.980	0.963	0.992	0.999	0.813	0.982	0.949	0.996	0.989	0.980
# Detected Anomalies	4	361	619	1488	17	379	600	1029	96	390	435
FP Rate	0.5	0.100	0.018	0	0.294	0.292	0.36	0.052	0.042	0.167	0.290

Ablation study. As described in Sec III, RIOLU takes five steps to automatically infer patterns and detect pattern anomalies: column sampling, coverage rate (r_{cov}) estimation, constrained template generation, pattern generation, and pattern selection. To understand the impacts of these steps and the design decisions therewithin, we carried out an ablation study on Auto-RIOLU for pattern anomaly detection using the 5 public datasets in RQ2. The ablation study considers five variants of Auto-RIOLU: 1) In the column sampling step, sampling 20% of a column's records (similar to FlashProfile [23]) instead of using a statistically representative sample; 2) removing the r_{cov} estimation step and using a static default r_{cov} of 0.95; 3) in the constrained template generation step, removing the constraint, generating templates that give exact match for every record (i.e., r_{EM}=1); 4) using a static pattern selection threshold (0.01 suggested by the online document of FlashProfile) instead of K-Means clustering; and 5) selecting all generated patterns (i.e., no pattern selection).

Table VII presents the results of our ablation study. Overall, the original Auto-RIOLU has the highest and most stable performance. We observed that removing or modifying the pattern selection step (the "No P.S." and "Static P.S. Threshold" rows in Table VII) leads to the most significant performance drop, especially when the error rate is too high (e.g., the *Flights* dataset, with F1 from 0.738 to 0.135, an 81.7% drop): without proper pattern selection, anti-patterns may be selected, thus r_{cov} cannot be accurately estimated. Besides, subsampling 20% of the records might lead to unstable results, as a fixed-rate sampling may lead to under- or over-generalization of the original population. In addition, fixing r_{cov} to an initial default value such as 0.95 (i.e., without automated r_{cov} estimation) will allow RIOLU to over-optimistically assume that a fixed percentage (e.g., 95%) of the records are healthy, leading to sub-optimal results; constantly setting r_{EM} to 1 can over-generate trivial patterns and reduce their generalizability.

Sensitivity analysis. As introduced in Sec III-F, the r_{cov} estimation relies on the r_{cov_init} and the number of subsets sampled from the column (N_{subset}). To validate the impact of our default choice (i.e., r_{cov_init}=0.95, N_{subset}=5), we carried out a sensitivity analysis. We vary r_{cov} from 0.85 to 0.99 in increments of 0.05 (except the last increment to 0.99); and N_{subset} from 1 to 10 in increments of 1.

We observe that the performance of Auto-RIOLU is relatively stable across different parameter settings in Fig. 6. The average F1 score across the datasets only changes 3.1% when we vary the r_{cov_init} from 0.85 to 0.95; further increasing the r_{cov_init} (i.e., an initial too-high assumption of the health rate) would notably decrease the performance. The optimal choice of r_{cov_init} varies across datasets, given that their error rates are drastically different. Nevertheless, we recommend the use of the default parameter setting ($r_{cov\_init} = 0.95$) as it provides generalizable and near-optimal performance for most of the datasets, including those not used when building our tool. While changing the number of subset samples (N_{subset}), Auto-RIOLU also shows a relatively stable performance. We observed a peak of average F1 score when setting N_{subset} to 5: a smaller number may cause the samples being less representative, whereas a larger number would decrease the efficiency and may cause too many overlaps among the samples. Hence, we selected 5 as the default N_{subset} value.

TABLE VII
THE ABLATION STUDY OF AUTO-RIOLU. THE BOLD VALUES INDICATE THE HIGHEST, AND THE ITALIC VALUES SHOW THE LOWEST F1 SCORES FOR EACH DATASET, RESPECTIVELY. "PATTERN SELECTION" IS ABBREVIATED AS "P.S.".

	Hosp-1k	Hosp-10k	Hosp-100k	Flights	Movies
Original	**0.611**	**0.487**	**0.485**	0.738	**1.0**
20% Column Sampling	*0.436*	*0.362*	0.462	**0.772**	**1.0**
Static r_{cov}=0.95	0.509	0.440	0.408	0.569	**1.0**
Static r_{EM}=1	0.537	0.445	0.323	0.767	**1.0**
Static P.S. Threshold	0.521	0.428	*0.296*	0.147	0.468
No P.S.	0.541	0.435	0.415	*0.135*	*0.360*

Summary. The fully automated version of RIOLU (Auto-RIOLU) outperforms the baselines on four out of five public datasets and achieves better efficiency. Guided-RIOLU further improves the performance (with up to 37.4% improvement of F1 over Auto-RIOLU) with the guidance of a small set of labeled data. Our extended evaluation on Java naming inconsistency detection and in an industry setting further demonstrates RIOLU's generalizability in detecting data pattern anomalies, hence contributing to the overall quality of software that relies on data.

V. RELATED WORKS

Data quality has incited tremendous interest in the software engineering (e.g., [9], [12], [49], [50]) and data engineering (e.g., [17], [24], [27], [51]) communities. In particular, pattern violation is a major type of data quality issue as it directly impacts the quality of downstream software [12], [14]–[16]. Below, we discuss prior work related to pattern-based data profiling, pattern anomaly detection, and work on language mining which shares some similarity with pattern inference.

Pattern-Based Data Profiling. The goal of pattern-based data profiling is to describe data with a set of patterns, to help understand the data [23]. Such comprehension of data is important for improving and maintaining data for software development and operations [12]. Pattern-based data profiling tools use syntactic structures to describe data. Microsoft's

SQL Server Data Tools [32] pioneered learning descriptive regular expressions, but they lack extensibility and comprehensiveness. Ataccama One [33] is a commercial offering that profiles the data using a set of base patterns. Potter's Wheel [26] describes the data domain using the most frequent pattern. FlashProfile [23] clusters records according to syntactic similarity and assigns a pattern to each cluster. However, we observe that these approaches (e.g., FlashProfile) are not sensitive to subtle data noises: they could lead to false positives caused by over-generalization. Hence, we propose a novel approach to generate precise and general patterns, even when noises exist in the data (RIOLU can automatically distinguish healthy patterns from noises).

Pattern Anomaly Detection. Pattern-based anomaly detection leverages manually designed or automatically inferred patterns to detect data anomalies [17]. TFDV [24] and Deequ [51] require manual design of regular expressions as patterns. These approaches are not effective when users lack knowledge of their data, and would be time-consuming for a large volume of data. Data-Scanner-4C [28] considers a human-in-the-loop method and lets users select healthy examples for pattern inference and inconsistency identification. However, example selection can still be labor-consuming since users must inspect the dataset for multiple loops. To save manual effort, some studies focus on automated pattern inference. Potter's wheel [26] leverages minimum description length to extract patterns for each column; XSystem [27] infers pattern for every column with a branch-merge technique; and FlashProfile [23] first clusters records by domain using syntactic similarity, then assign patterns to each cluster. Auto-Detect [30] and Auto-Validate [17] both consider pattern inference with multiple columns by leveraging knowledge from similar columns, aiming to infer precise and generalizable patterns and suppress the false positive rate for anomaly detection. However, we noticed that these previous approaches require domain-specific thresholds to identify pattern anomalies, posing difficulty for users to use them for different data domains. Therefore, in this work, we propose a fully automated, unsupervised, and auto-parameterized approach for pattern-bassed anomaly detection.

Language Mining. The task of pattern inference is related to the field of language mining and general grammar induction. For example, the L* algorithm [52] is an impacting method for inferring language using labeled samples. Recent works such as Glade [53], Mimid [54], and Arvada [55] further enhanced the generalizability and speed of context-free grammar inference with positive examples; there is also specific research on grammar inference for Ad-Hoc parsers [56]. However, these works do not explicitly support the detection of anomalies and do not offer an unsupervised variant. Users have to determine whether an input is a positive sample before feeding it into the algorithm. Moreover, although context-free grammars are expressive, the rules are more difficult for non-expert users to understand, validate, and modify. Conversely, Regexes are easier to write and expressive to validate tabular data with patterns (e.g., DateTime, URL), thus are used by IT companies such as Microsoft [23], [32] and Amazon [25].

VI. Threats to Validity

Construction Validity. Given the potential noise in the data and randomness in our approach (e.g., from column sampling), the pattern inference results and the performance of RIOLU may fluctuate. Thus, we ran the tests on RIOLU five times to suppress the impact and collect the average results. Programming language and network latency may impact the efficiency comparison among the tools: RIOLU was coded in Python, while FlashProfile was written in .NET, which is a more time-efficient language; ChatGPT inferred the patterns with powerful server hardware and ran remotely, while other tools were run locally. The time efficiency of RIOLU can be further improved with more efficient language or powerful hardware.

Internal Validity. We assume that the health data patterns should cover more records than anomaly patterns. Thus, we use K-Means to find a natural threshold for the patterns' coverage rate. K-Means is used given its low time complexity and wide use for data clustering. We acknowledge that more advanced clustering approaches (e.g., deep learning-based ones [57]) exist; their combination with RIOLU can be verified in the future. We recognize that the generated patterns' quality may be limited by the ignorance of data types (e.g., generating "non-existing" patterns for free text), or some records with rare legitimate patterns may be flagged due to their statistical minority; we suggest that these problems could be feasibly fixed through human validation on the corresponding domain.

External Validity. We tested RIOLU's performance on data in various domains and sizes (e.g., the anomaly detection datasets have 999 to 100k records). However, our findings may be limited to the datasets that we considered. Our work would benefit from future work that evaluates RIOLU on more diverse datasets, especially in practical settings.

VII. Conclusions

This work proposes a fully automated pattern inference and anomaly detection approach, RIOLU, which does not require human labeling or parameter configuration. RIOLU leverages statistical heuristics and automated clustering to mitigate the problem of parameter configuration, and uses a rule-based progressive structure to ensure the precision of pattern inference. Our experiment results show that RIOLU can precisely generate patterns to describe data from various domains and detect pattern anomalies, in a fully automated manner, outperforming the baselines (including ChatGPT) in terms of both accuracy and efficiency. Moreover, we found that Guided-RIOLU, a variant of RIOLU with the guidance of a small labeled dataset, can further improve the anomaly detection performance. Our evaluation in an industrial setting proves the usefulness of our approach in a practical setting. RIOLU has the potential to benefit software and data engineering communities by improving the quality of the ever-growing data and the software built upon them. For example, our lightweight approach may be incorporated into the continuous integration pipeline of data-centric systems to support unit tests for data.

REFERENCES

[1] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2015.

[2] M. Carey, S. Ceri, P. Bernstein, U. Dayal, C. Faloutsos, J. Freytag, G. Gardarin, W. Jonker, V. Krishnamurthy, M. Neimat *et al.*, "Data-centric systems and applications," 2008.

[3] I. Ozkaya, "What is really different in engineering ai-enabled systems?" *IEEE software*, vol. 37, no. 4, pp. 3–6, 2020.

[4] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104773, 2022.

[5] R. Croft, Y. Xie, and M. A. Babar, "Data preparation for software vulnerability prediction: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1044–1063, 2022.

[6] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 107–119.

[7] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.

[8] S. Xu, Y. Yao, F. Xu, T. Gu, J. Xu, and X. Ma, "Data quality matters: A case study of obsolete comment detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 781–793.

[9] A. Davoudian and M. Liu, "Big data systems: A software engineering perspective," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–39, 2020.

[10] Y. Hu, H. Jin, X. Wang, J. Gu, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Autoconsis: Automatic gui-driven data inconsistency detection of mobile apps," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, pp. 137–146.

[11] W. Tange, C. Wang, P. Yao, R. Wu, X. Fu, G. Fan, and C. Zhang, "Dclink: Bridging data constraint changes and implementations in fintech systems," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 914–925.

[12] L. E. Lwakatare, E. Rånge, I. Crnkovic, and J. Bosch, "On the experiences of adopting automated data validation in an industrial machine learning project," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 248–257.

[13] M. Stonebraker and E. K. Rezig, "Machine learning and big data: What is important?" *IEEE Data Eng. Bull.*, vol. 42, no. 4, pp. 3–7, 2019.

[14] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang, "Detecting data errors: Where are we and what needs to be done?" *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 993–1004, 2016.

[15] M. Mahdavi, Z. Abedjan, R. Castro Fernandez, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, "Raha: A configuration-free error detection system," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 865–882.

[16] L. Visengeriyeva and Z. Abedjan, "Metadata-driven error detection," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, 2018, pp. 1–12.

[17] J. Song and Y. He, "Auto-validate: Unsupervised data validation using data-domain patterns inferred from data lakes," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1678–1691.

[18] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1356–1367.

[19] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[20] Y. Li, D. Zha, P. Venugopal, N. Zou, and X. Hu, "Pyodds: An end-to-end outlier detection system with automated machine learning," in *Companion Proceedings of the Web Conference 2020*, 2020, pp. 153–157.

[21] K.-H. Lai, D. Zha, G. Wang, J. Xu, Y. Zhao, D. Kumar, Y. Chen, P. Zumkhawaka, M. Wan, D. Martinez *et al.*, "Tods: An automated time series outlier detection system," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 18, 2021, pp. 16 060–16 062.

[22] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 187–196.

[23] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. Millstein, "Flashprofile: a framework for synthesizing data profiles," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[24] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich, "Data validation for machine learning." in *MLSys*, 2019.

[25] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1781–1794, 2018.

[26] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *VLDB*, vol. 1, 2001, pp. 381–390.

[27] A. Ilyas, J. M. da Trindade, R. C. Fernandez, and S. Madden, "Extracting syntactical patterns from databases," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 41–52.

[28] S. Yu, L. Han, M. Indulska, S. Sadiq, and G. Demartini, "Human-in-the-loop regular expression extraction for single column format inconsistency," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 3859–3867.

[29] "The replication package of riolu," https://github.com/mooselab/Discover-Data-Quality-With-RIOLU, 2024.

[30] Z. Huang and Y. He, "Auto-detect: Data-driven error detection in tables," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1377–1392.

[31] Y. He, J. Song, Y. Wang, S. Chaudhuri, V. Anil, B. Lassiter, Y. Goland, and G. Malhotra, "Auto-tag: Tagging-data-by-example in data lakes," *arXiv preprint arXiv:2112.06049*, 2021.

[32] Microsoft, "SQL Server Data Tools Documentation," https://docs.microsoft.com/en-gb/sql/ssdt, [Online; accessed 18-March-2024].

[33] Ataccama Corporation, "Ataccama Website," https://www.ataccama.com/, [Online; accessed 18-March-2024].

[34] P. D. Grünwald, *The minimum description length principle*. MIT press, 2007.

[35] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[36] S. Padhi, "Data profiling dataset," https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests, 2024, [Online].

[37] OpenAI, "Chatgpt," https://openai.com/chatgpt/, 2024, [Online].

[38] M. L. Siddiq, J. Zhang, and J. C. D. S. Santos, "Understanding regular expression denial of service (redos): Insights from llm-generated regexes and developer forums," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 190–201.

[39] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré, "Holoclean: Holistic data repairs with probabilistic inference," *arXiv preprint arXiv:1702.00820*, 2017.

[40] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava, "Truth finding on the deep web: Is the problem solved?" *arXiv preprint arXiv:1503.00303*, 2015.

[41] Microsoft, "Prose pattern inspector document," https://www.microsoft.com/en-us/research/project/prose-pattern-inspector/, 2024, [Online; accessed Jul 16, 2024].

[42] S. McConnell, *Code Complete: Steve McConnell*. Microsoft Press, 1993.

[43] K. Beck, *Implementation patterns*. Pearson Education, 2007.

[44] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[45] A. A. Takang, P. A. Grubb, R. D. Macredie *et al.*, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.

[46] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 31–35.

[47] Oracle, "Java naming conventions," https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html, 2024, [Online; accessed Jul 16, 2024].
[48] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning (ICML)*, 2016.
[49] H. Foidl, M. Felderer, and R. Ramler, "Data smells: categories, causes and consequences, and detection of suspicious data in ai-based systems," in *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, 2022, pp. 229–239.
[50] A. Shome, L. Cruz, and A. Van Deursen, "Data smells in public datasets," in *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, 2022, pp. 205–216.
[51] S. Schelter, P. Schmidt, T. Rukat, M. Kiessling, A. Taptunov, F. Biessmann, and D. Lange, "Deequ - data quality validation for machine learning pipelines," in *NeurIPS 2018*, 2018. [Online]. Available: https://www.amazon.science/publications/deequ-data-quality-validation-for-machine-learning-pipelines
[52] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.
[53] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 95–110, 2017.
[54] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 172–183.
[55] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 456–467.
[56] M. Schröder and J. Cito, "Grammars for free: Toward grammar inference for ad hoc parsers," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2022, pp. 41–45.
[57] Y. Ren, J. Pu, Z. Yang, J. Xu, G. Li, X. Pu, S. Y. Philip, and L. He, "Deep clustering: A comprehensive survey," *IEEE Transactions on Neural Networks and Learning Systems*, 2024.