Towards Automated Task-Aware Data Validation

Hao Chen
BIFOLD & TU Berlin
Berlin, Germany
hao.chen@tu-berlin.de

Sebastian Schelter
BIFOLD & TU Berlin
Berlin, Germany
schelter@tu-berlin.de

ABSTRACT

Data is a central resource for modern enterprises and institutions, and data validation is essential for ensuring the reliability of downstream applications. However, a major limitation of existing automated data unit testing frameworks is that they ignore the specific requirements of the tasks that consume the data. This paper introduces a task-aware approach to data validation that leverages large language models to generate customized data unit tests based on the semantics of downstream code. We present `tadv`, a prototype system that analyzes task code and dataset profiles to identify data access patterns, infer implicit data assumptions, and produce executable code for data unit tests. We evaluate our prototype with a novel benchmark comprising over 100 downstream tasks across two datasets, including annotations of their column access patterns and support for assessing the impact of synthetically injected data errors. We demonstrate that `tadv` outperforms task-agnostic baselines in detecting the data columns accessed by downstream tasks and generating data unit tests that account for the end-to-end impact of data errors. We make our benchmark and prototype code publicly available.

ACM Reference Format:
Hao Chen and Sebastian Schelter. 2025. Towards Automated Task-Aware Data Validation. In *Workshop on Data Management for End-to-End Machine Learning (DEEM '25), June 22–27, 2025, Berlin,Germany.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3735654.3735939

1 INTRODUCTION

Data is a central resource for modern enterprises and institutions, and data issues, such as missing or incorrect information, can seriously impact their operations [1, 24]. As a consequence, data validation frameworks such as TensorFlow Data Validation [14], Amazon's Deequ [13, 16, 19], and Great Expectations [5] have become widely used in industry in the last years. These frameworks generate data unit tests by profiling a data sample and subsequently inferring constraints based on heuristics, against which to validate unseen data.

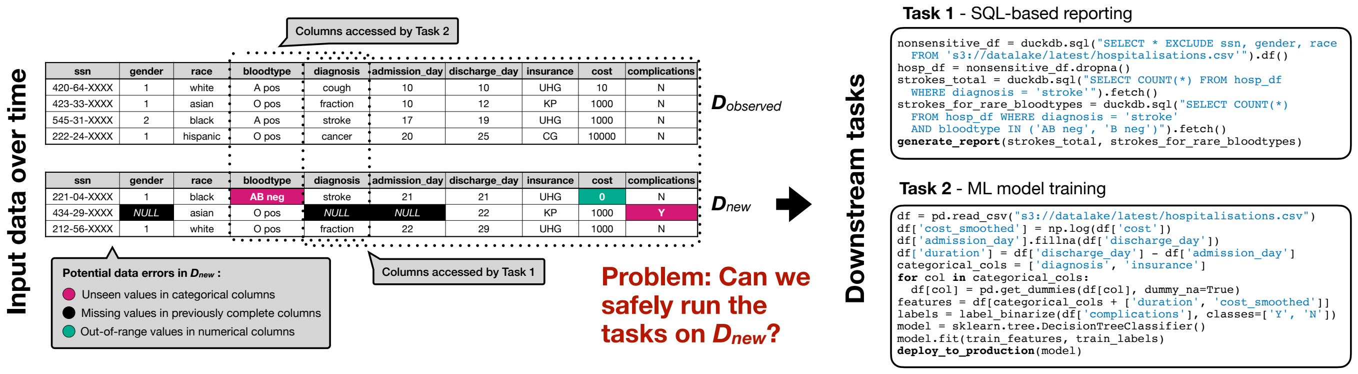
Shortcomings of data unit test frameworks in industry. However, existing frameworks suffer from various shortcomings: (i) the heuristically suggested constraints are often either too strict or too general and must typically be adjusted and extended by an end user with domain knowledge; (ii) data unit tests are often developed in a tedious, reactive way: once data problems become apparent in production systems, these problems are manually fixed and the tests are extended to prevent their reoccurrence in the future; (iii) it is difficult to tradeoff the precision/recall of alerts from data unit tests, leading to a tension between false alarms and missed issues. Researchers proposed several extensions to address these shortcomings in recent years, which either leverage statistics from historical executions [15, 20, 23] and large data corpora [4, 8, 21] or require a human in the loop [7, 9, 11]. However, none of these approaches have gained wide real-world adoption so far.

Towards automated task-aware data validation. We argue that a major limitation of current approaches is that they focus on observed data only and ignore the characteristics of downstream tasks that consume the unseen data. This leads to several missed opportunities to improve data unit tests and address some of the outlined shortcomings. First of all, certain downstream tasks might only access parts of the data, especially for large denormalized datasets common in enterprise data lakes, which means that only issues in these parts of the data must be considered for the task. Secondly, in many cases, the code of downstream tasks will be written by experienced data engineers, will have implicit knowledge about the data "baked in", and might therefore already be robust against some data issues. Some downstream tasks like ML training tasks might even be naturally robust against certain types of noise in data. In summary, we argue that there is a lot of potential to improve the automated generation of data unit tests by *specializing them to the downstream tasks* for which they are deployed.

However, this specialization is inherently difficult as it requires an "understanding" of downstream task code. Even seemingly simple problems like identifying which columns a piece of code accesses are challenging and typically handled via static code analysis with hand-curated knowledge bases [12]. Approaches like fuzzing-based testing [14] are also difficult to apply in practice, as they assume that one can generate synthetic input data and repeatedly execute the downstream tasks in a "test mode". This is impossible in many industry settings, where task execution has side effects.

Overview and contributions. We propose to take downstream tasks into account for automated data unit test generation. In particular, we proactively [25] generate specialized data unit tests for each task, which take into account the specific data access pattern and data assumptions of the task. We motivate this direction with a running example (Section 2), present our LLM-based prototype system `tadv`, which breaks down the task-aware data unit test generation into multiple steps (data profiling, column access detection, constraint generation, and code generation). We discuss how to implement each of these steps and leverage the code understanding [6] and rule generation capabilities of LLMs [9, 10] in Section 3. Finally, we design a novel benchmark for task-aware data unit test generation with more than 100 downstream tasks of various types (ML pipelines, SQL queries, website generation code) for two datasets. This benchmark includes hand-labeled ground truth of

DEEM '25, June 22–27, 2025, Berlin, Germany
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1924-0/2025/06.
https://doi.org/10.1145/3735654.3735939


Input data over time
Columns accessed by Task 2
Columns accessed by Task 1
D observed
D new
Potential data errors in D new :
● Unseen values in categorical columns
● Missing values in previously complete columns
● Out-of-range values in numerical columns
Problem: Can we safely run the tasks on D new?
Downstream tasks

ssn	gender	race	bloodtype	diagnosis	admission_day	discharge_day	insurance	cost	complications
420-64-XXXX	1	white	A pos	cough	10	10	UHG	10	N
423-33-XXXX	1	asian	O pos	fraction	10	12	KP	1000	N
545-31-XXXX	2	black	A pos	stroke	17	19	UHG	1000	N
222-24-XXXX	1	hispanic	O pos	cancer	20	25	CG	10000	N

ssn	gender	race	bloodtype	diagnosis	admission_day	discharge_day	insurance	cost	complications
221-04-XXXX	1	black	AB neg	stroke	21	21	UHG	0	N
434-29-XXXX	*NULL*	asian	O pos	*NULL*	*NULL*	22	KP	1000	Y
212-56-XXXX	1	white	O pos	fraction	22	29	UHG	1000	N

Task 1 - SQL-based reporting

```
nonsensitive_df = duckdb.sql("SELECT * EXCLUDE ssn, gender, race
  FROM 's3://datalake/latest/hospitalisations.csv'").df()
hosp_df = nonsensitive_df.dropna()
strokes_total = duckdb.sql("SELECT COUNT(*) FROM hosp_df
  WHERE diagnosis = 'stroke'").fetch()
strokes_for_rare_bloodtypes = duckdb.sql("SELECT COUNT(*)
  FROM hosp_df WHERE diagnosis = 'stroke'
  AND bloodtype IN ('AB neg', 'B neg')").fetch()
generate_report(strokes_total, strokes_for_rare_bloodtypes)
```

Task 2 - ML model training

```
df = pd.read_csv("s3://datalake/latest/hospitalisations.csv")
df['cost_smoothed'] = np.log(df['cost'])
df['admission_day'].fillna(df['discharge_day'])
df['duration'] = df['discharge_day'] - df['admission_day']
categorical_cols = ['diagnosis', 'insurance']
for col in categorical_cols:
  df[col] = pd.get_dummies(df[col], dummy_na=True)
features = df[categorical_cols + ['duration', 'cost_smoothed']]
labels = label_binarize(df['complications'], classes=['Y', 'N'])
model = sklearn.tree.DecisionTreeClassifier()
model.fit(train_features, train_labels)
deploy_to_production(model)
```

Figure 1: Running example of a healthcare dataset evolving over time, where newly received data potentially contains errors. Two downstream tasks, SQL-based reporting and ML model training, depend on this data, and a data unit test is required to check whether these tasks can safely execute on the new data without failure or performance degradation.

column access patterns and functionality to evaluate the impact of randomly injected data errors onto the performance of downstream tasks (Section 4). In summary, we provide the following contributions.

- We *introduce the problem* of automated task-aware data unit test generation with a running example (Section 2).
- We discuss tadv, our LLM-based *prototype system*, which breaks down task-aware data unit test generation into multiple steps such as data access pattern detection and constraint generation from a task's source code (Section 3).
- We design a *novel benchmark* for task-aware data unit test generation with *more than 100 downstream tasks* of various types for two datasets, and conduct an *extensive experimental evaluation* of our prototype (Section 4).
- We make our code and benchmark available under an open license at https://github.com/guangchen811/tadv/blob/deem/.

2 PROBLEM STATEMENT

We introduce the problem of this paper with a running example.

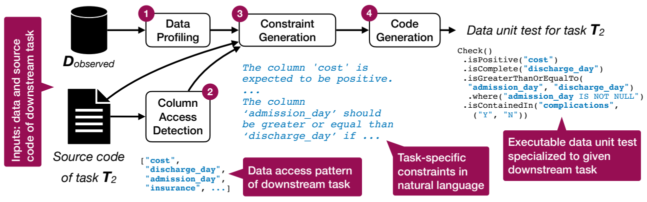
Running example. Our running example evolves around a fictitious healthcare provider, which processes externally supplied data about the hospitalizations of patients, as shown on the left side of Figure 1. This data includes personal information (ssn, gender, race, bloodtype), information about diagnosis and stay (diagnosis, admission_day, discharge_day), financial information (insurance, cost) and an indication of whether there were any complications. Updates to the data arrive in bulk over time, e.g., a new data partition is received and has to be processed every night. In our example, $\mathcal{D}_{\text{observed}}$ denotes the data seen and processed so far, while $\mathcal{D}_{\text{new}}$ denotes the latest data partition received, which has not been processed yet. Several downstream tasks have to be run regularly to process newly arrived data. Our scenario includes the following two toy examples for tasks:

- *Task 1 - SQL-based reporting* – The first task (code shown on the top right side of Figure 1) generates a report from the data by processing it via dataframe and SQL queries. In particular, it computes the fraction of patients with rare blood types who are diagnosed with a particular disease.
- *Task 2 - ML model training* – The second task (code shown on the bottom right side of Figure 1) trains and deploys a machine learning model to identify patients with potential complications who might need prioritized care.

The data engineering team of the healthcare provider has repeatedly had to fix data quality incidents where downstream tasks failed due to issues in the data, and the engineers had to spend their weekends fixing the data and rerunning the affected downstream tasks. To avoid such problems in the future, they now want to implement a *data unit test* [16] for the newly received data $\mathcal{D}_{\text{new}}$, which is supposed to tell them whether it is safe to ingest the new data. For that, they try out the automated generation of data unit tests from TFDV (via "schema inference" [22]) and from Deequ (via "constraint suggestion" [3]). These libraries profile the existing data $\mathcal{D}_{\text{observed}}$ and apply several heuristics to produce a set of constraints to evaluate for the new data $\mathcal{D}_{\text{new}}$. We provide a notebook[1] with the corresponding code.

Shortcomings of existing data validation approaches. The data unit tests from both libraries reject $\mathcal{D}_{\text{new}}$ due to the following issues: the columns bloodtype and complications contain previously unseen values (AB neg and Y) and the columns gender, diagnosis and admission_day have missing values even though they were complete in $\mathcal{D}_{\text{observed}}$. This rejection triggers a detailed investigation of the data from the engineering team and they also study the code of the downstream tasks. During this investigation, they gain the following insights: Task 1 can be safely run on $\mathcal{D}_{\text{new}}$, as it only accesses the diagnosis and bloodtype columns of the data and is robust to tuples with missing values, which it explicitly removes via a dropna operation. Furthermore, the code shows that AB neg is actually a valid value for a (very rare) bloodtype, as it is hardcoded in a SQL query. Task 2 is also robust against the issues flagged by TDFV and Deequ: it does not access the gender column, it handles missing values in admission_day via a fillna operation and safely encodes missing values in diagnosis via the get_dummies operation and also expects the N value in complications for the label_binarize operation. However, Task 1 actually

[1] https://github.com/guangchen811/tadv/blob/deem/workflow/s2_experiments/t2_constraint_inference/running_example/constraint_inference_demo.ipynb


Inputs: data and source code of downstream task
$D_{observed}$
Source code of task T_2
1 Data Profiling
2 Column Access Detection
3 Constraint Generation
4 Code Generation
Data unit test for task T_2
Data access pattern of downstream task
Task-specific constraints in natural language
Executable data unit test specialized to given downstream task

```
["cost",
 "discharge_day",
 "admission_day",
 "insurance", ...]
```

```
The column 'cost' is
expected to be positive.
...
The column
'admission_day' should
be greater or equal than
'discharge_day' if ...
```

```
Check()
 .isPositive("cost")
 .isComplete("discharge_day")
 .isGreaterThanOrEqualTo(
  "admission_day", "discharge_day")
  .where("admission_day IS NOT NULL")
 .isContainedIn("complications",
   ("Y", "N"))
```

Figure 2: Task-aware data unit test generation in `tadv` (for Task 2 from our running example). We propose to generate a specialized data unit test per downstream task, based on the data and task code as inputs.

crashes when run on $\mathcal{D}_{\text{new}}$, but both Deequ and TFDV failed to detect the corresponding data issue: the `cost` column must not contain non-positive values due to a `np.log` operation used to encode this feature, as the logarithm is only defined for positive values. After the investigation, the data engineers realized the following shortcomings of current approaches for generating data unit tests:

- Data unit tests generated via data profiling and heuristic rules are often unreliable (e.g., with respect to rare categorical values) and have to be manually adjusted by someone with appropriate domain knowledge.
- In many cases, it is insufficient to have a single set of data unit tests for a dataset, instead, a custom test for each downstream task is needed, as each task accesses different parts of the data and contains varying assumptions about the data.

Formal problem statement. We formally define the problem of generating task-aware data unit tests. We are given a relational dataset $\mathcal{D} = \mathcal{D}_{\text{observed}} \cup \mathcal{D}_{\text{new}}$ and a set of downstream tasks $\mathcal{T} = \{T_1, \ldots, T_n\}$ with their corresponding source code $\{S_1, \ldots, S_n\}$, which operate on this data. We assume that we initially have access to $\mathcal{D}_{\text{observed}}$, which contains well-formed data that can be successfully processed by the downstream tasks, and that we must validate an unseen data partition $\mathcal{D}_{\text{new}}$, which arrives in bulk. A constraint $c : \mathcal{D} \rightarrow \{0, 1\}$ is a boolean function, which typically computes and evaluates an aggregate statistic on the data (e.g., whether the number of missing values in a column is larger than a given threshold). The concrete research problem is now to generate a data unit test $U_i = \{c_{i0}, \ldots, c_{it}\}$ for each task T_i based on the observed data $\mathcal{D}_{\text{observed}}$ and the source code S_i of the task. The unit test U_i should operate in a way that the satisfaction of all constraints $c \in U_i$ implies successful execution of the downstream task T_i with $\mathcal{D}_{\text{new}}$ as input:

$$\bigwedge_{c \in U_i} c(\mathcal{D}_{\text{new}}) \Leftrightarrow T_i \textit{ produces correct results for } \mathcal{D}_{\text{new}}$$

3 APPROACH

We propose to break down the task-aware data unit test generation problem into a set of steps, as illustrated in Figure 2, where steps ②-④ are repeated for each task to generate custom data unit tests. Our approach proceeds as follows: First, we run ① *data profiling* on the observed data $\mathcal{D}_{\text{observed}}$ to infer data types, obtain descriptive statistics, etc. We reuse existing techniques for this step [2, 9, 16]. The next step is ② *column access detection*, where we inspect the source code S_i of a T_i to determine which columns of $\mathcal{D}_{\text{observed}}$ are accessed by the task. Next, we run ③ *constraint generation* to generate a set of constraints for the task T_i in natural language, based on its accessed columns of $\mathcal{D}$, the results from the data profiling step and implicit assumptions mined from its source code S_i. Finally, we execute the ④ *code generation* step to turn the natural language constraints into an executable data unit tests for task T_i.

Implementation. We implement our approach in the prototype system `tadv`. We use Deequ for data profiling ①. We implement column access detection ② and constraint generation ③ via LLM calls with custom-designed prompts and access to the source code of the downstream tasks. Inspired by Toolformer [18], we also use an LLM call to convert the constraints into executable Deequ code ④. We implement two variants of the constraint generation step. In `tadv [+deequ]`, we provide the LLM with the constraints suggested by Deequ, allowing it to refine, add, or remove constraints as needed. The variant `tadv [+exp]` instead supplies the LLM with a natural language description of the heuristic rules used by Deequ, offering insight into its underlying logic without prescribing specific constraints. Throughout the process, we ask the LLM to produce the output in JSON format. Since LLMs do not always generate valid JSON consistently, we implement a retry mechanism and re-run each LLM call up to three times if the output is invalid.

4 PRELIMINARY EXPERIMENTS

Datasets. We experiment with two datasets from Kaggle: a loan approval[2] dataset with 58,645 rows and 13 different columns, and a dataset from the healthcare domain[3] with synthetic patient records with 55,500 rows and 15 different columns. These real-world datasets span diverse domains, various data types and contain semantically rich personal information (e.g., age, income, blood type).

Downstream tasks. We experiment with three types of downstream tasks for each of the two datasets: ML scripts in Python (15 per dataset), SQL queries (30 per dataset), and Python scripts for static website generation (10 per dataset). We generate the code for the resulting 110 tasks via OpenAI's GPT-4o model, guided by data statistics, task descriptions, and input/output format requirements. We ensure that each task is executable and manually refine the code to make it reflect real-world characteristics (e.g., by adding comments, introducing dead code, etc.).

4.1 Column Access Detection

Next, we evaluate how well `tadv` identifies accessed columns, which is essential for focusing constraint suggestions on relevant data.

Experimental setup. We generate a benchmark for column access detection by manually inspecting the code of each of our 110 downstream tasks and hand-labeling the exact columns it accesses. We expose the task code to both `tadv` and a string-matching baseline. The baseline infers accessed columns by checking whether each column name, or a slight variation of it, appears as a substring in the code. We predict the accessed columns with both approaches and use the macro-averaged F1 score to evaluate predictive performance, which balances precision (correctly predicted columns) and recall (all accessed columns captured) across tasks.

[2] https://www.kaggle.com/competitions/playground-series-s4e10
[3] https://www.kaggle.com/datasets/prasad22/healthcare-dataset

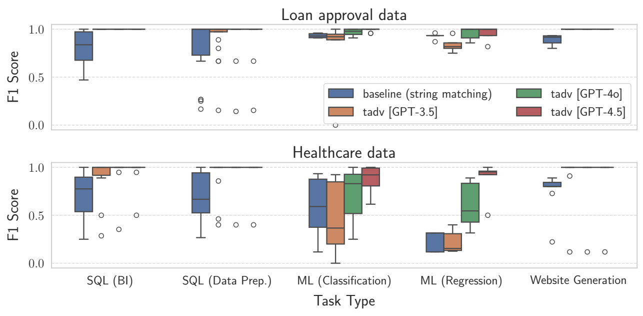


Figure 3: F1 scores for column access detection on various downstream tasks. Our system tadv outperforms the baseline across all task types, often by a wide margin.

Results and discussion. We boxplot the distribution of the resulting F1 scores in Figure 3. Our system `tadv` outperforms the string-matching baseline, often by a wide margin. On average across all settings, `tadv` with GPT-3.5 achieves an F1 score 12.1% higher than the string-matching baseline, while `tadv` with GPT-4o outperforms the baseline by 22.3%. The best F1 scores come from `tadv` with GPT-4.5 (26.3% higher than the baseline), although this LLM is currently about 30 times more expensive than GPT-4o.

4.2 End-To-End Error Impact

This experiment evaluates `tadv`'s ability to distinguish between harmful and non-harmful data errors in an end-to-end setting. It assesses whether data unit tests can effectively detect issues that adversely affect downstream task behavior.

Experimental setup. We extend Jenga [17] to inject diverse errors into different columns of our datasets, simulating real-world data issues such as missing or new categorical values, missing or extra columns, numerical scaling, and noise. Thereby, we create a variety of corrupted datasets on which we execute the tasks to assess their performance, covering more than 500 cases. Next, we evaluate generated data unit tests on these cases. Such a test is considered effective if it passes on data that does not harm the downstream task and fails on data that does. For ML tasks, we check whether the script crashes or whether the model accuracy drops by more than 5%. For other tasks, we evaluate harm based solely on whether the script executes successfully, as we lack an objective performance metric, such as the F1 score used in ML tasks.

Methods. We compare fully automated data unit test generation approaches, and use the "constraint suggestion" feature of Deequ [16] (`deequ`) and the "schema inference" feature from TensorFlow Data Validation [14] (`tensorflow-dv`) as baselines. Both baselines generate constraints solely from the observed data without leveraging downstream task information. We cannot include Great Expectations as a baseline, as it currently does not provide an automated way to generate data unit tests. We evaluate `tadv` as an automated and task-aware approach. We additionally include its two variants, `tadv [+deequ]` and `tadv [+exp]` (see Section 3), to analyze the benefits of integrating Deequ's constraints and heuristics into `tadv`.

Results and discussion. The results in Table 1 show that in terms of impactful error detection, `tadv` drastically outperforms both `deequ` (with a difference of more than 0.5 in F1 score) and `tensorflow-dv` (with a difference of more than 0.3 in F1 score). This is expected since the constraint generation mechanisms in `deequ` and `tensorflow-dv` ignore the characteristics of the downstream tasks and are designed to produce strict, conservative constraints, which lead to a high number of false alarms (> 300 for both) in this setup. Our system variant `tadv [+exp]` produces the lowest number of false alarms while maintaining strong performance in detecting task failures, highlighting the effectiveness of leveraging task semantics and heuristic priors. In summary, these findings confirm the potential of specializing data unit tests to their respective downstream tasks.

	No impact		Task failure		
Method	Pass	False alarm	Detection	Miss	**F1 Score**
`deequ`	95	370	41	4	0.337
`tensorflow-dv`	161	304	45	0	0.514
`tadv`	373	92	22	23	0.866
`tadv [+deequ]`	344	121	33	12	0.838
`tadv [+exp]`	374	91	28	17	**0.874**

Table 1: Detection performance with respect to the impact of data errors on downstream tasks. Task-aware methods outperform deequ and tensorflow-dv by raising fewer false alarms, with tadv [+exp] achieving the highest F1 score.

4.3 Uncovering Implicit Data Assumptions

We present a selection of examples, which showcase that `tadv` can generate task-aware constraints based on data assumptions in the code. We find cases that show that `tadv` can trace operations on columns through code, e.g., in one task, the column `"Date of Admission"` is renamed via `df.rename({"Date of Admission": "admission_dt"})` and later converted to a date value via `pd.to_datetime(df.admission_dt)` using the new name. Still, `tadv` correctly generates a constraint stating that "Date of Admission should be convertible to a valid datetime format". We also observe a case where the task code explicitly checks the value range of the Age column via `assert X["Age"].min() > 0` and `assert X["Age"].max() <= 100`. We find that `tadv` recognizes this and generates corresponding constraints: "Age values must be greater than 0" and "Age values must be less than or equal to 100".

5 NEXT STEPS & OPEN QUESTIONS

To improve the robustness and accuracy of constraint generation, we plan to incorporate additional task types and explore structured code representations that better capture data operations. Future extensions include support for more complex scenarios, such as constraints spanning multiple columns and tables, as well as handling multi-file code bases. Moreover, we plan to expand our benchmark with a dataset of detailed, task-specific constraints across all three supported task types to enable more systematic evaluation. An open question is how to define and measure the impact of data errors in non-ML tasks, such as SQL queries and website generation, where execution failures are currently the only available signal and may be too coarse. In terms of failure handling, a key direction is to determine whether validation failures arise from the data or the task code, and to develop automated repair strategies for both.

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data profiling: A tutorial. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1747–1751.

[3] Amazon. 2025. Automatic Suggestion of Constraints. https://github.com/awslabs/deequ/blob/master/src/main/scala/com/amazon/deequ/examples/constraint_suggestion_example.md. [Online; accessed March-2025].

[4] Sijie Dong, Soror Sahri, Themis Palpanas, and Qitong Wang. 2025. Automated Data Quality Validation in an End-to-End GNN Framework. (2025).

[5] Great Expectations. 2024. Great Expectations. https://greatexpectations.io/. [Online; accessed January-2025].

[6] Stefan Grafberger, Hao Chen, Olga Ovcharenko, and Sebastian Schelter. 2025. Towards Regaining Control over Messy Machine Learning Pipelines. In *Workshop on Data-AI Systems (DAIS) at ICDE*.

[7] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[8] Zhipeng Huang and Yeye He. 2018. Auto-detect: Data-driven error detection in tables. In *Proceedings of the 2018 International Conference on Management of Data*. 1377–1392.

[9] Zezhou Huang and Eugene Wu. 2024. Cocoon: Semantic table profiling using large language models. In *Proceedings of the 2024 Workshop on Human-In-the-Loop Data Analytics*. 1–7.

[10] Hoa Thi Le, Angela Bonifati, and Andrea Mauri. 2025. Graph Consistency Rule Mining with LLMs: an Exploratory Study. (2025).

[11] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A Configuration-Free Error Detection System. *SIGMOD* (2019).

[12] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated provenance tracking in data science scripts. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 1542–1551.

[13] David Nigenda, Zohar Karnin, Muhammad Bilal Zafar, Raghu Ramesha, Alan Tan, Michele Donini, and Krishnaram Kenthapadi. 2022. Amazon sagemaker model monitor: A system for real-time insights into deployed machine learning models. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3671–3681.

[14] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. 2019. Data validation for machine learning. *MLSys* 1 (2019), 334–347.

[15] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion.. In *EDBT*. 61–72.

[16] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.

[17] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. 2021. JENGA - A Framework to Study the Impact of Data Errors on the Predictions of Machine Learning Models. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:232283615

[18] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[19] Amazon Web Services. 2024. AWS Glue Data Quality. https://docs.aws.amazon.com/glue/latest/dg/glue-data-quality.html. [Online; accessed January-2025].

[20] Shreya Shankar, Labib Fawaz, Karl Gyllstrom, and Aditya Parameswaran. 2023. Automatic and precise data validation for machine learning. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 2198–2207.

[21] Jie Song and Yeye He. 2021. Auto-validate: Unsupervised data validation using data-domain patterns inferred from data lakes. In *Proceedings of the 2021 International Conference on Management of Data*. 1678–1691.

[22] Tensorflow. 2025. TensorFlow Data Validation - An Example of a Key Component of TensorFlow Extended. https://colab.research.google.com/github/tensorflow/tfx/blob/master/docs/tutorials/data_validation/tfdv_basic.ipynb. [Online; accessed March-2025].

[23] Dezhan Tu, Yeye He, Weiwei Cui, Song Ge, Haidong Zhang, Shi Han, Dongmei Zhang, and Surajit Chaudhuri. 2023. Auto-Validate by-History: Auto-Program Data Quality Constraints to Validate Recurring Data Pipelines. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4991–5003.

[24] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. Scoded: Statistical constraint oriented data error detection. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 845–860.

[25] Sepanta Zeighami, Yiming Lin, Shreya Shankar, and Aditya Parameswaran. 2025. LLM-Powered Proactive Data Systems. *arXiv preprint arXiv:2502.13016* (2025).