Towards Regaining Control over Messy Machine Learning Pipelines

Stefan Grafberger, Hao Chen, Olga Ovcharenko, Sebastian Schelter
BIFOLD & TU Berlin
{grafberger, hao.chen, ovcharenko, schelter}@tu-berlin.de

***Abstract*—Software systems that learn from data with machine learning (ML) are increasingly used to automate impactful decisions. However, the resulting ML pipelines suffer from many unsolved data management challenges with respect to personal and security-critical data and compliance with legal regulations. We argue that this is due to shortcomings in existing ML pipeline abstractions and "messy" imperative code produced by data scientists. We propose a new approach for ML pipelines that leverages the code generation capabilities of large language models to extract declarative logical query plans from messy data science code. We envision this as a foundation to manage deployed ML pipelines and their data artifacts in upcoming Data-AI systems. We discuss a challenging example scenario and present initial experiments with a prototype to validate our vision.**

I. Introduction

Software systems that learn from data with machine learning (ML) are increasingly used to automate impactful decisions, raising critical data management questions [1]–[5].

Unsolved data management challenges in ML applications. Modern data-driven organizations run hundreds of ML pipelines [6], but often struggle with managing sensitive personal and security-critical data [7], [8]. Google's text completion system, for example, contained credit card numbers from personal emails [9]. Furthermore, complex ML applications often reproduce and amplify existing discrimination, for example with respect to age [10], race [11] or gender [12]. Consequently, new regulatory requirements for ML applications are emerging globally, such as the right to delete personal data [13]–[15] or comprehensive data governance obligations for high-risk ML applications [16]. We argue that the current data pipeline design in ML applications lacks the foundations to adequately address these data management challenges.

The need to retroactively raise the level of abstraction in data science code. Current ML pipeline libraries lack fundamental data-centric abstractions such as logical query plans in databases. Major cloud providers offer services based on custom pipeline abstractions [17]–[19] without making data a first-class citizen or modeling the semantics of individual operations. Instead, these services focus on flexibility and ease of deployment, treating pipelines as workflows with black-box Python operators. This focus shifts the burden of handling complex requirements, like regulatory compliance, to developers.

The data management community has recognized these shortcomings and shown how to enhance ML applications with provenance tracking, debugging, inspection, and automatic rewriting capabilities [20]–[25]. However, integrating these techniques into real-world systems is difficult, as they rely on declarative abstractions, which are not necessarily present in existing imperative data science code. Both industry and academia have proposed systems [26]–[28] for data scientists to (re)write their pipelines. Another research line enhances existing declaratively written ML pipelines without requiring code modifications [20], [21], [23], [24]. Unfortunately, the real-world adoption of these directions is still limited. Data scientists typically focus on the ML aspects of the pipeline and treat data preparation as grunt work, writing or generating messy imperative Python code to "get the job done" quickly.

The connection to data systems in production. Many severe data issues are only found after ML pipelines are deployed [4], [5] and require timely fixes. For example, protecting users from harm [29], [30] via the right-to-be-forgotten [31], [32] and security incidents [9], [33] require low-latency, fine-grained data deletion ('unlearning') from ML artifacts. However, such functionality is difficult to implement as the pipeline artifacts span different data types (relational data, tensor data, model parameters) and are produced by different operations (relational queries, linear algebra-based feature encoding, and gradient-based learning). Current cloud services and feature stores do not support functionality like unlearning as they mainly focus on feature and model artifacts and lack sufficient abstractions for end-to-end pipeline "queries" that explicitly model the semantics of individual operators. We argue that ML pipeline abstractions, like logical query plans, are crucial for integrating complex pipelines with Data-AI systems.

Vision. We propose to address the outlined challenges via *large language model (LLM)-assisted rewrite of messy pipeline code*. In contrast to previous approaches like MLflow Recipes [27] (which did not gain significant adoption), we argue that it is unrealistic to expect data scientists to write or generate their ML pipeline code in a declarative way that enables easy integration and management of complex pipelines with compound AI systems. Instead, we propose to use the promising code generation capabilities [34]–[38] of LLMs to automatically rewrite messy pipeline code to an intermediate declarative abstraction, from which we then extract a logical query plan for pipeline management in upcoming AI systems. We prototype selected aspects of our vision and make the code available at https://github.com/sscdotopen/lester.

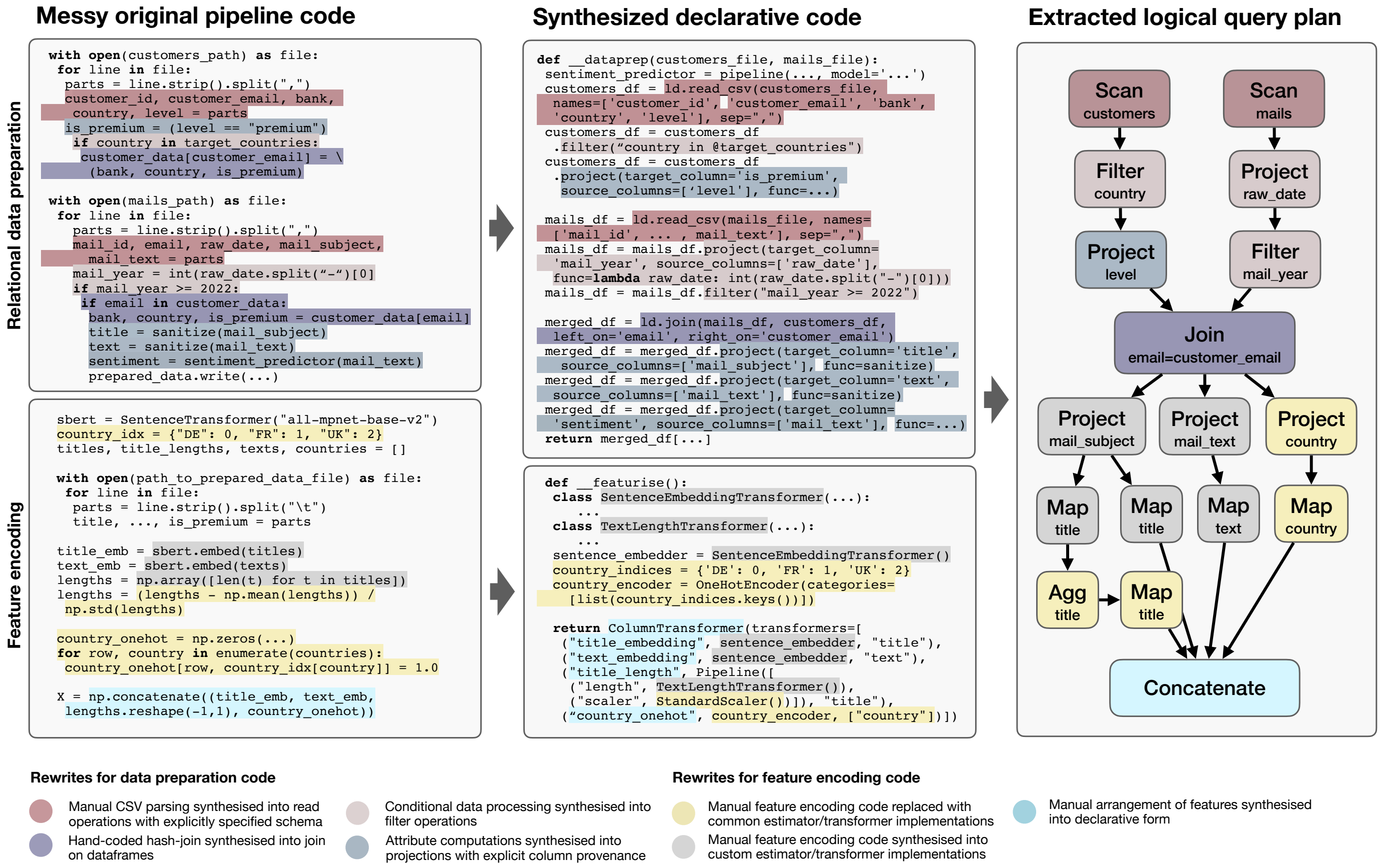

Fig. 1. Our vision – messy imperative ML pipeline code is automatically rewritten into declarative code with a custom dataframe API and feature encodings with Estimator/Transformers. The declarative code allows the extraction of a logical query plan of the pipeline, which enables data systems to manage the pipeline and its data artifacts in novel ways, e.g., to apply fine-grained provenance tracking, complex data debugging and targeted unlearning from the pipeline artifacts. *We would like to highlight that the code shown in the middle is automatically synthesized from the messy code on the left by our prototype implementation discussed in Section IV.*

II. Running Example

Imagine a financial service provider, which maintains a data lake with semi-structured customer data from several banks, including emails about questions, complaints, and service requests. Multiple ML pipelines regularly train models, e.g., for fraud detection, service request prioritization, or text completion for chatbots.

Scenario: Urgent removal of security-critical personal data. Consider a scenario inspired by reported credit card leaks [9]: Data engineers discover that credit card numbers in the email subject for customers from certain banks were not scrambled during the data import. This leak poses severe financial and reputational risks for the company, requiring immediate deletion of the leaked data. However, ML pipelines may also have consumed this data, which now could be contained in data artifacts produced by the pipelines, recoverable from encoded representations using membership inference attacks [39], or memorized by the resulting ML models!

Technical challenges. The data engineers face two key challenges: (i) identifying affected ML pipelines and models, and (ii) rectifying the affected models and pipeline artifacts without deleting all models and artifacts (due to compliance reasons) or re-executing all pipelines from scratch, which would be tedious and expensive. Unfortunately, the data engineers realize that most of the companies' ML pipelines consist of messy imperative Python code written by data scientists. This lack of structure makes it almost impossible to automatically identify affected pipelines and rectify the models and artifacts. As a result, overtime hours and tedious "detective work" are necessary to address the security issue. The data engineers must manually identify pipelines that consumed the problematic data, decipher the intended logic embedded in the messy code, and trace the flow of affected data across various pipeline stages. Finally, they have to write custom code for each affected pipeline to update their artifacts, which is both time-consuming and error-prone.

Facing this production incident, the engineers now regret directly deploying the messy, unstructured code. They wish there were dedicated systems to assist them with such incidents, e.g., a feature store with detailed provenance information and pipeline rewriting capabilities, or even a Data-AI system that can automatically identify and rectify the affected pipelines.

Messy example pipeline. We illustrate the technical challenges of this scenario with an exemplary ML pipeline that

identifies complaints from premium customers based on email content. We show the corresponding "messy" pipeline code on the left side of Figure 1. The code in the top left box prepares the training data for the ML model, based on CSV files about customers and emails from the data lake. This code has several issues: ● manual parsing of the CSV files with the schema implicitly encoded in variable names, ● data filtering with conditional statements, ● computation of new attributes in plain Python code with Python UDFs, and ● even a hand-coded hash join to map customers to their emails. The code for encoding the training data as features for model training (bottom left box) is also messy. First, ● embeddings are generated for the `title` and `text` attributes of the training data with an external model. Second, ● imperative NumPy code computes the normalized word count of the `title` as a feature, followed by a one-hot encoding of the `country` assignment. The generated features are then ● manually concatenated into a feature matrix.

Efficient removal of the security-critical data here is difficult because it requires fine-grained data provenance at both the record and feature level, and the ability to selectively re-execute pipeline operations and efficiently update artifacts. For this, one must track individual records through the pipeline, map feature matrix dimensions to their sources, selectively update the initial input data, re-encode affected records, adjust specific feature matrix ranges, and "unlearn" the data from the models without costly full retraining. All of this requires untangling messy imperative code to identify and update only the affected parts.

III. Vision

Next, we describe our vision for handling cases like our running example. Figure 2 gives an overview of the proposed approach: ❶ Data scientists develop their pipelines without any restrictions. ❷ The messy original pipeline code is automatically synthesized into declarative ML code to extract a logical query plan; ❸ The pipeline and its artifacts are deployed and managed in feature stores and data-AI systems. Our approach is built on a formal ML pipeline model, which generalises existing pipeline abstractions [21]–[24], [27], [40], [41]. This model includes common relational data processing operations and feature encoding techniques and enables provenance tracking and automatic rewriting capabilities [20]–[24].

LLM-assisted rewrite of messy pipeline code. A fundamental problem, however, is how to rewrite existing messy data science code to such a declarative form that allows extraction of logical query plans. For that, we propose to leverage the promising code transformation capabilities of LLMs [34]–[38]. Our vision is to assist data scientists in rewriting messy imperative pipeline code to a custom declarative API representing our proposed computational model. As mentioned, the API can be based on existing libraries from the data science ecosystem (or variants of them with minor changes). Current LLMs are pre-trained on gigantic web crawls and, thus, have already seen massive amounts of code and documentation (e.g., Stack Overflow questions) from

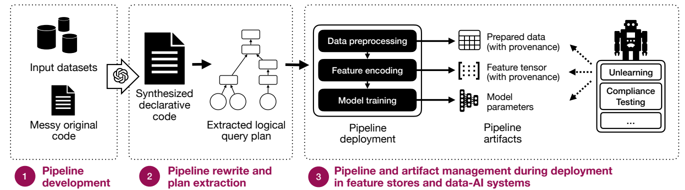

Fig. 2. Overview our envisioned approach – ❶ Data scientists develop their pipeline without any restrictions. ❷ The messy original pipeline code is automatically synthesized into declarative ML code to extract a logical query plan for the pipeline; ❸ The pipeline and its artifacts are deployed and managed in feature stores and Data-AI systems.

these libraries! We propose to exploit this fact by designing an API for our computational model that is syntactically close to existing libraries but makes it easy to identify the semantics of operations, track provenance, etc. This declarative API will be the synthesis target for the LLM-driven code transformation.

Running example. The colored lines in Figure 1 highlight some of the code transformations required to synthesize a declarative version of the example pipeline (shown in the middle) from its messy original code (shown on the left). The code that ● manually parses the CSV files with their schema implicitly encoded in variable names is now synthesized into calls to a `read_csv` method which explicitly specifies the column names. The code for ● filtering data with conditional statements is synthesized into explicit calls to `filter` operations on dataframes. The code to compute ● new attributes is synthesized into extended projections, conducted via explicit calls to a `project` operation on dataframes that contains the name of the `target_column` and `source_columns` for column provenance tracking. The ● hand-coded hash join is replaced with a `join` operation on dataframes with explicit join keys. In the feature encoding part, the manual encoding operations ●● are replaced with custom generated or existing Estimator/Transformers (ETs) (`StandardScaler` and `OneHotEncoder`) from scikit-learn, and ● scikit-learn's `ColumnTransformer` is used to specify how to combine the encoded features into a feature tensor. By executing the synthesized code built with our custom declarative API, our approach can then easily extract the final logical query plan (shown on the right) at runtime [21], [24].

Next steps. We outline the next steps for research on the LLM-assisted rewrites of messy pipeline code. The code should be synthesized step-by-step according to the different stages of the pipeline. For each stage, the system will identify the corresponding parts of the messy original code (e.g., all the code that contributes to initial data preparation) and gather additional meta information (e.g., the schemas of the involved input and output dataframes) by inspecting the input files. Next, the system will synthesize the declarative version of the code for the pipeline stage and apply a series of automated validation steps, e.g., checks for the correct parsing of the generated code and running the synthesized code on the input data (or samples of the input data) to compare its outputs with the original code. In case of potential errors, the system

will run several rounds of correction attempts [42], by feeding back error messages to the underlying LLM (with potential hints from the data scientist). As ultima ratio, the system could always ask the data scientist to manually fix the synthesized code after a series of unsuccessful correction attempts.

Quantifying the quality of pipeline rewrites. A community benchmark on logical query plan extraction will be required to evaluate various plan extraction approaches and foster further research in this area. Such a benchmark should include a large number of messy real-world ML pipelines collected from public code repositories such as GitHub or Kaggle. The benchmark should also include code from ML pipelines deployed in industry (or "proxy" reimplementations if the code cannot be shared). We plan to use our envisioned system to bootstrap the benchmark. The evaluation of various plan extraction approaches should be automated and require low effort. To evaluate different plan extraction approaches, we will manually create executable "ground-truth" query plans for the code examples. The benchmark can then execute both the manually created and automatically extracted query plans to compare their outputs to measure correctness and efficiency.

Downstream use cases. We envision our logical query plan abstraction as the foundation for integrating and managing complex ML pipelines in upcoming feature stores and Data-AI systems. This approach will enable use cases like efficient end-to-end unlearning of data, holistic debugging of complex training and serving pipelines, and compliance testing for regulations like the upcoming EU AI Act in Europe [16], [43], [44]. Our approach will also help to transpile the pipeline code into other languages [36], e.g., to SQL for execution in a database [45] or to SparkML for distributed execution [46].

IV. Prototype

We conduct a prototypical implementation of our ideas available at https://github.com/sscdotopen/lester and present some initial experiments to validate our vision.

Synthesis of declarative pipeline code. We prototype a first version of the code synthesis system. Our implementation uses OpenAI's `gpt-4o` model via the LangChain API. We create a custom prompt for each pipeline stage, with a description of the target API in our prototype, followed by step-by-step instructions for the synthesis and layout of the rewritten code. We carefully design our prompts to handle several encountered issues, e.g., to ensure that the synthesized code preserves the exact code semantics and is laid out in a way that is easy to execute. We, for example, instruct the LLM to turn global variables into function-local variables and to position imports inside the generated function code. In general, we observe that the task of identifying larger computational patterns (such as hand-coded hash joins or the computation of new data attributes) is handled well by the underlying LLM. However, it is challenging to synthesize correctly typed code, for example, for the generated UDFs for extended projections or the column assignments in the `ColumnTransformer`.

Potential of code synthesis. We evaluate the potential of our proof-of-concept code synthesis implementation. We experiment with nine different code examples (three for each pipeline stage: relational data preparation, feature encoding, and model training), including our running example and real-world code obtained from GitHub. The code examples contain challenging operations such as manual CSV parsing, handwritten joins of partitioned data, cleaning and tokenization operations for string data, low dimensional projections of featurized data, and the manual construction of feature tensors. We provide the synthesized code at https://github.com/sscdotopen/lester/blob/main/synthesised.md

Results and discussion. We find that our prototype manages to synthesize correct executable code for eight out of the nine examples (including the complex running example in Figure 1). In one case, manual adjustment of two code lines was necessary to make the code read partitioned inputs correctly (a loop over files and a function argument needed adjustments). In another case, some nonsensical dead code was produced which did not impact the final output. We encountered a case where the featurization code was improved by adding missing value imputation. Finally, there is a case, where the synthesized code even corrects a data leakage issue (as an ET was fitted on the test data in the messy original code). In summary, our initial results confirm the high potential of LLM-based code synthesis for making pipelines declarative.

Runtime benefits of provenance-based unlearning. We also prototype the code for end-to-end unlearning on the example pipeline, via provenance-based updates of the feature matrix and a first-order update of the model [33]. We evaluate the runtime benefits of unlearning for our example scenario with a small amount of credit card leaks. We compare the time to re-execute the original pipeline from scratch to the time for conducting an unlearning update on the materialized artifacts. We experiment with synthetically generated customer and mail data and ask for updates to leaked data for five customers. We use a growing number (up to 100,000) of email records and customers (up to 10,000) as pipeline inputs.

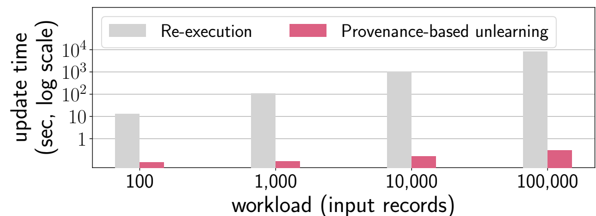

Fig. 3. Time (in logarithmic scale) to re-execute a pipeline from scratch versus time for end-to-end unlearning from the pipeline artifacts.

Results and discussion. In Figure 3, we show the resulting mean runtimes in seconds on a logarithmic scale. The Figure shows that the time to re-execute the pipeline from scratch scales linearly with the size of the input data, and that the re-execution already takes more than 140 minutes for an input size of 100,000 mails.

REFERENCES

[1] J. Stoyanovich *et al.*, "Responsible data management," *CACM*, 2022.

[2] S. Grafberger *et al.*, "Red onions, soft cheese and data: From food safety to data traceability for responsible ai." *IEEE Data Eng. Bull.*, vol. 47, no. 1, 2024.

[3] S. Shankar and A. Parameswaran, "Towards observability for production machine learning pipelines," *VLDB*, 2023.

[4] S. Shankar, R. Garcia, J. M. Hellerstein, and A. G. Parameswaran, ""we have no idea how models will behave in production until production": How engineers operationalize machine learning," *Proc. ACM Hum.-Comput. Interact.*, vol. 8, no. CSCW1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3653697

[5] K. Holstein, J. Wortman Vaughan, H. Daumé III, M. Dudik, and H. Wallach, "Improving fairness in machine learning systems: What do industry practitioners need?" in *Proceedings of the 2019 CHI conference on human factors in computing systems*, 2019, pp. 1–16.

[6] D. Xin *et al.*, "Production machine learning pipelines: Empirical analysis and optimization opportunities," *SIGMOD*, 2021.

[7] Biddle, Sam, "Facebook Engineers: We have no idea where we keep all your personal data," https://theintercept.com/2022/09/07/facebook-personal-data-no-accountability/, 2022.

[8] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[9] N. Carlini *et al.*, "The secret sharer: Evaluating and testing unintended memorization in neural networks," *USENIX Security*, 2019.

[10] Consumers International & Mozilla, "A Consumer investigation into personalised pricing," https://www.consumersinternational.org/media/369070/personalized_pricing.pdf, 2022.

[11] M. Ali, P. Sapiezynski, M. Bogen, A. Korolova, A. Mislove, and A. Rieke, "Discrimination through optimization: How facebook's ad delivery can lead to biased outcomes," *Proceedings of the ACM on human-computer interaction*, vol. 3, no. CSCW, pp. 1–30, 2019.

[12] J. Buolamwini and T. Gebru, "Gender shades: Intersectional accuracy disparities in commercial gender classification," in *Conference on fairness, accountability and transparency*. PMLR, 2018, pp. 77–91.

[13] General Data Protection Regulation, "Art. 17 GDPR - Right to erasure ('right to be forgotten')," https://gdpr-info.eu/art-17-gdpr/, 2024.

[14] California Consumer Privacy Act (CCPA), "Requests to Delete," https://oag.ca.gov/privacy/ccpa#sectiond, 2024.

[15] D. S. University, "Internet information service algorithmic recommendation management provisions." [Online]. Available: https://digichina.stanford.edu/work/translation-internet-information-service-algorithmic-recommendation-management-provisions-opinon-seeking-draft/

[16] EU AI Act, "Article 10: Data and Data Governance," https://artificialintelligenceact.eu/article/10/, 2024.

[17] Microsoft, "Azure Machine Learning Pipelines," https://learn.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines, 2018.

[18] Amazon Web Services, "SageMaker Pipelines," https://aws.amazon.com/sagemaker/pipelines/, 2020.

[19] Google, "Vertex AI Pipelines," https://cloud.google.com/vertex-ai/docs/pipelines, 2021.

[20] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer, "Vamsa: Automated provenance tracking in data science scripts," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 1542–1551.

[21] S. Grafberger *et al.*, "Data distribution debugging in machine learning pipelines," *VLDB Journal*, vol. 31, no. 5, 2022.

[22] L. Flokas *et al.*, "Complaint-driven training data debugging at interactive speeds," *SIGMOD*, 2022.

[23] Schelter *et al.*, "Proactively screening machine learning pipelines with arguseyes," *SIGMOD*, 2023.

[24] S. Grafberger *et al.*, "Automating and optimizing data-centric what-if analyses on native machine learning pipelines," *SIGMOD*, 2023.

[25] H. Mohammed, A. Yao, L. Flokas, Z. Hongbin, C. Summers, and E. Wu, "Accelerating deletion interventions on olap workload," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 5659–5659.

[26] M. Boehm *et al.*, "Systemds: A declarative machine learning system for the end-to-end data science lifecycle," *CIDR*, 2020.

[27] Databricks, "Mlflow Recipes," https://mlflow.org/docs/latest/recipes.html, 2022.

[28] Pixeltable, "Pixeltable," https://pixeltable.readme.io/, 2024.

[29] V. Warmerdam, "Beyond Broken – this is not a happy story." [Online]. Available: https://koaning.io/posts/beyond-broken/

[30] Washington Post, "Dear tech companies, I don't want to see pregnancy ads after my child was stillborn," 2018. [Online]. Available: https://www.washingtonpost.com/lifestyle/2018/12/12/dear-tech-companies-i-dont-want-see-pregnancy-ads-after-my-child-was-stillborn/

[31] S. Schelter, S. Grafberger, and T. Dunning, "Hedgecut: Maintaining randomised trees for low-latency machine unlearning," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1545–1557.

[32] S. Schelter, M. Ariannezhad, and M. de Rijke, "Forget me now: Fast and exact unlearning in neighborhood-based recommendation," in *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2023, pp. 2011–2015.

[33] A. Warnecke *et al.*, "Machine unlearning of features and labels," *NDSS*, 2023.

[34] R. C. Fernandez *et al.*, "How large language models will disrupt data management," *VLDB*, 2023.

[35] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.

[36] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, and A. Cheung, "Verified code transpilation with llms," *NeurIPS*, 2024.

[37] C. Hong, S. Bhatia, A. Haan, S. K. Dong, D. Nikiforov, A. Cheung, and Y. S. Shao, "Llm-aided compilation for tensor accelerators," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–14.

[38] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[39] C. Song and A. Raghunathan, "Information leakage in embedding models," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 377–390. [Online]. Available: https://doi.org/10.1145/3372297.3417270

[40] K. Karanasos, M. Interlandi, D. Xin, F. Psallidas, R. Sen, K. Park, I. Popivanov, S. Nakandal, S. Krishnan, M. Weimer *et al.*, "Extending relational query processing with ml inference," *CIDR*, vol. 2, no. 4, p. 7, 2020.

[41] B. Karlaš, D. Dao, M. Interlandi, S. Schelter, W. Wu, and C. Zhang, "Data debugging with shapley importance over machine learning pipelines," in *The Twelfth International Conference on Learning Representations*, 2023.

[42] M. Urban and C. Binnig, "Caesura: Language models as multi-modal query planners," *CIDR*, 2024.

[43] EU AI Act, "Annex III: High-Risk AI Systems Referred to in Article 6(2)," https://artificialintelligenceact.eu/annex/3/, 2024.

[44] ——, "Article 14: Human Oversight," https://artificialintelligenceact.eu/article/14/, 2024.

[45] M. E. Schüle, L. Scalerandi, A. Kemper, and T. Neumann, "Blue elephants inspecting pandas: Inspection and execution of machine learning pipelines in sql." in *EDBT*, 2023, pp. 40–52.

[46] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.