APEX-DAG: Library and Language independent Pipeline EXtraction

Sebastian Eggers
BIFOLD & TU Berlin
Berlin, Germany
sebastian.eggers@campus.tu-berlin.de

Nina Żukowska
BIFOLD & TU Berlin
Berlin, Germany
zukowska@tu-berlin.de

Ziawasch Abedjan
BIFOLD & TU Berlin
Berlin, Germany
abedjan@tu-berlin.de

ABSTRACT

Modern data-driven systems often rely on complex pipelines to process and transform data for downstream machine learning (ML) tasks. Extracting these pipelines and understanding their structure is critical for ensuring transparency, performance optimization, and maintainability, especially in large-scale projects. In this work, we introduce a novel system, APEX-DAG (**A**utomating **P**ipeline **EX**traction with **D**ataflow, Static Code **A**nalysis, and **G**raph Attention Networks), which automates the extraction of data pipelines from computational notebooks or scripts. Unlike execution-based methods, APEX-DAG leverages static code analysis to identify the dataflow, transformations, and dependencies within ML workflows without executing the code or the need to alter the code. Further, after an initial training phase, our system can identify pipelines that built with previously unseen libraries.

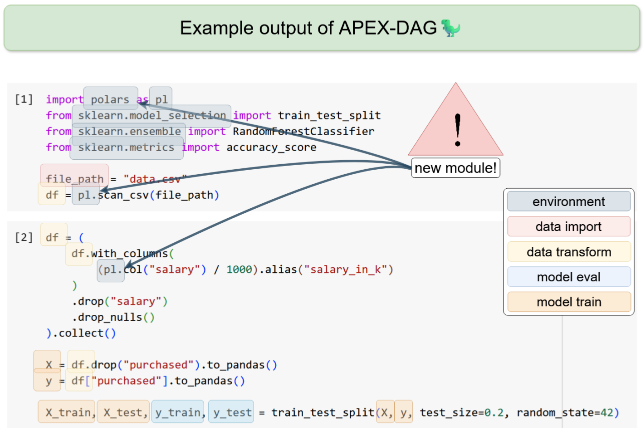
PVLDB Reference Format:
Sebastian Eggers, Nina Żukowska, and Ziawasch Abedjan. APEX-DAG: Library and Language independent Pipeline EXtraction. PVLDB, 18(12): 5375 - 5378, 2025.
doi:10.14778/3750601.3750675

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/S-Eggers/APEX-DAG.

1 INTRODUCTION

Machine learning (ML) pipelines in production are complex, often undocumented, and require automation for understanding and reuse [21]. This lack of documentation complicates compliance, particularly in regulated domains like finance and healthcare [15]. Data experts frequently face a manual and error-prone process to infer pipeline structures and dependencies [16]. For instance, tracking lineage in a cloud-based ecosystem with diverse technologies, such as ETL pipelines, Python-based models in Vertex AI [1], and legacy Java systems, requires significant efforts. Provenance tracking is especially challenging because different tools and languages use varying constructs for the same operations.

Recent approaches have focused on automating the tasks of provenance tracking and pipeline extraction [9, 11, 14, 17, 18, 20]. Yet, existing approaches either rely on code execution to infer code semantics or they are limited to a set of predefined library operations. Code execution is computationally expensive and infeasible at scale [8, 11]. Additionally, due to the dynamic nature of programming languages like Python, solving the problem of automating data lineage solely with rule-based analysis is insufficient [18]. For example, consider Figure 1. Existing approaches such as Vamsa [14] and Geyser [18], which is bound to its predefined knowledge base, would not be able to track the provenance in this program because the employed libraries are not documented within their systems, as Polars is a newer dataframe library. Such approaches would need to continuously adapt their KB with newly developed libraries. To overcome this limitation, we propose APEX-DAG, which is independent of specific versions or kinds of data and ML libraries, allowing it to generalize to unseen libraries without requiring manual updates.



```
[1]  import polars as pl
     from sklearn.model_selection import train_test_split
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.metrics import accuracy_score

     file_path = "data.csv"
     df = pl.scan_csv(file_path)

[2]  df = (
         df.with_columns(
             (pl.col("salary") / 1000).alias("salary_in_k")
         )
         .drop("salary")
         .drop_nulls()
     ).collect()

     X = df.drop("purchased").to_pandas()
     y = df["purchased"].to_pandas()

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 1: Annotated ML workflow with Polars and scikit-learn functions via APEX-DAG

In this work, we demonstrate APEX-DAG, a system designed to automatically extract data pipelines and provenance information from code, that enables the understanding of data transformations and dependencies within these pipelines. Our system does not need to alter or execute the code and does not rely on apriori defined KB or rules. To generalize across many different pipelines and languages, APEX-DAG abstracts from the original code of a program to its language-independent abstract syntax trees that are then mapped to a labeled dataflow graph (DFG). This is done in a two-step learning process. First, a self-supervised pre-training step learns the structure of a dataflow graph representation of ML pipelines. We pretrain a graph attention network (GAT) on three tasks on a DFG: (1) edge existence, (2) edge type prediction, and (3) node type classification.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750675

Then a supervised fine-tuning approach learns the abstract role of a DFG node and edge by mapping each node to relevant lineage elements, such as dataset, data transformation function, training and others. The lightweight design of APEX-DAG ensures scalability, making it suitable for large code repositories and environments with thousands of ML workflows. In this demonstration, we present APEX-DAG and the effectiveness of its training process:

(1) We show the training process of APEX-DAG, which includes dataflow extraction from ML workflow notebooks, pre-training the GAT in a self-supervised manner, and fine-tuning the GAT. In particular, it will show how static code analysis is capable to capture the semantics of code pipelines in a step-wise process
(2) We show and explain the features of APEX-DAG, particularly how it helps when building a complex machine learning pipeline in a Jupyter Notebook, and highlight how it can understand new, previously unknown libraries and functions that serve data science tasks.
(3) We enable participants to directly interact with APEX-DAG by providing our Python library and a fine-tuned model as downloadable artifacts that can be used via Jupyter notebooks and VS code.

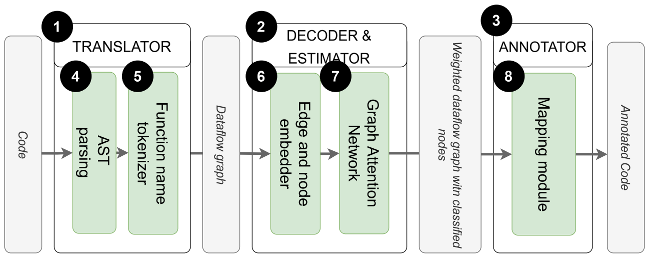


Figure 2: APEX-DAG system overview.

2 RELATED WORK

Data lineage research can be categorized into the traditional view of dataset lineage and an emerging subtopic that focuses on ML pipeline lineage, which is directly relevant to this work. Since most ML pipeline lineage systems extract pipelines from ML code, we refer to this process as pipeline extraction.

2.1 Data Lineage

Data lineage, also referred to as provenance, involves tracking the origin, transformations, and final destination of datasets [12]. It is divided into two categories: The first focuses on the origin of datasets relative to a specific output, known as where-lineage. The second investigates the transformations applied to an input to produce specific outputs, referred to as how-lineage [5].

Determining data provenance within ML pipelines first requires the extraction of the pipeline [9–11, 14, 18]. Depending on the goal of the system, either coarse-grained or fine-grained lineage is derived in a subsequent step. Extracting coarse-grained lineage requires identifying the pipeline. In contrast, detecting fine-grained lineage in ML pipelines requires replaying or executing the pipeline also [18]. Various systems rely on a fixed set of libraries to accomplish this task [9–11, 13, 19]. An alternative approach, such as that proposed by Vamsa [14], involves annotation. Vamsa, designed for Python scripts, analyzes abstract syntax trees (ASTs) and generates workflow graphs. These graphs are annotated using a knowledge base and subsequently utilized by a provenance tracker to extract lineage information. However, while Vamsa avoids reliance on a fixed set of libraries, it remains limited by the contents of its knowledge base. Additionally, it cannot handle complex dataflows like loops or conditional execution of a pipeline. The subsequent system, Geyser [18], extends the KB annotation approach to provide fine-grained lineage by additionally extracting dynamic lineage information. While operating on the same data structures, the system executes the pipeline and injects code into various parts of the pipeline to capture dynamic provenance information. Still, Geyser is limited by the extension of the KB. Another approach involves pipeline execution to identify data frame objects [22]. Although the primary goal of this work is not lineage extraction, it recommends pipelines or related tables using data lakes in a notebook environment. Nevertheless, this implementation is limited to pandas. KGLiDS semantically abstracts pipelines using static code analysis [11]. It operates on an AST to store the code and data flow for each statement corresponding to a variable. However, it relies on predefined operations, such as pandas.read_csv, to extract datasets. Our approach does not rely on predefined libraries or knowledge bases.

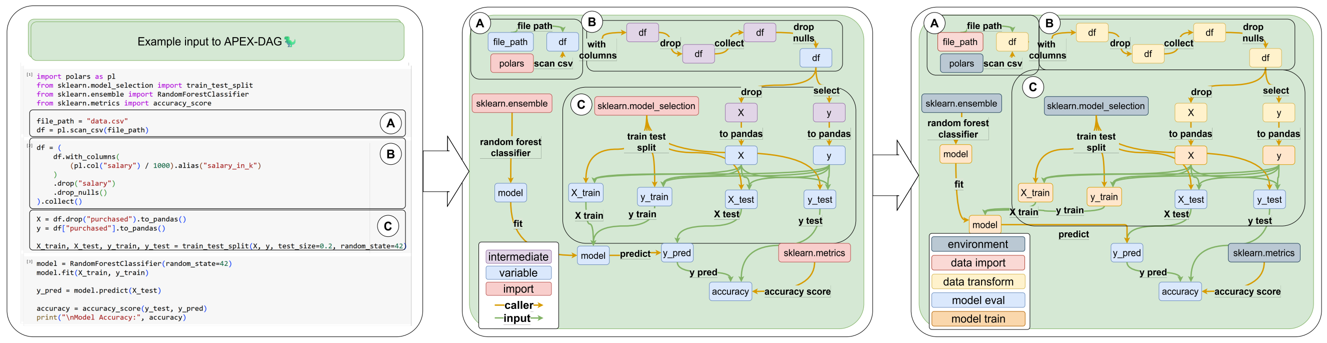
2.2 Pipeline Debugging and Understanding

Pipeline debugging and understanding depend on knowing where to locate the actual pipeline. This requires the step of extracting the pipeline itself. mlinspect is a tool designed to diagnose and address data distribution bugs and bias issues in machine learning pipelines [9], among other capabilities, the system captures dataflow and lineage during execution. However, executing a large set of pipelines, which might also involve training or fine-tuning large language models, is computationally expensive and thus infeasible. Thus, we argue that static code analysis methods are preferable, especially when dealing with a large and heterogeneous codebase. In a subsequent paper, the authors leverage dataflow tracking to automatically identify issues such as data leakage or unnormalized features in ML pipelines [20]. Notably, they avoid executing the pipeline wherever possible. Nevertheless, their approach relies on mlinspect to extract pipeline artifacts and lineage. Our approach can be a useful tool for supporting pipeline debugging.

3 SYSTEM OVERVIEW

Figure 2 depicts the process of lineage tracing inside a given piece of code by APEX-DAG. APEX-DAG comprises three main components: ❶ Translator, ❷ Detector & Estimator , and ❸ Annotator:

(1) **Translator**: Code structures vary significantly across programming languages, making it challenging to extract a unified representation suitable for neural network processing. The Translator addresses this by first converting the program into an abstract syntax tree (AST) and then into a dataflow graph (DFG). The DFG is further enriched



```
[1] import polars as pl
    from sklearn.model_selection import train_test_split
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import accuracy_score

    file_path = "data.csv"
    df = pl.scan_csv(file_path)

[2] df = (
        df.with_columns(
            (pl.col("salary") / 1000).alias("salary_in_k")
        )
        .drop("salary")
        .drop_nulls()
    ).collect()

    X = df.drop("purchased").to_pandas()
    y = df["purchased"].to_pandas()

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

[3] model = RandomForestClassifier(random_state=42)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print("\nModel Accuracy:", accuracy)
```

Figure 3: Example of a mapping between input code, the corresponding DFG and the DFG labelled by the detector and estimator.

with codeflow elements for pre-training, while variable and method names are encoded.

(2) **Detector & Estimator**: Once the DFG is constructed, it must be interpreted in the context of data or ML workflows. The Detector & Estimator processes the DFG to extract relevant structural and semantic features, enabling a graph attention network (GAT) to classify nodes and edges into lineage-relevant elements. This step ensures that lineage information is captured and structured appropriately.

(3) **Annotator**: Making lineage information accessible within the code environment is crucial for usability. The Annotator, implemented as a Jupyter plugin, visualizes lineage insights live inside Jupyter Notebooks. This interactive representation aids in pipeline development and debugging.

3.1 Translator

The input to the system can be code snippet, e.g., as a Jupyter Notebook. The translator processes the input by tokenizing all variable, method, and attribute names (❹) and converting the code into an abstract syntax tree (AST) via static code analysis (❺). For the demo, we use Python's built-in ast module, which we in time update to the TreeSitter library. Both, the AST and the tokenized names are programming language independent.

In the subsequent step, the translator traverses the AST to generate an annotated dataflow graph (aDFG), denoted as a directed multi-graph $G = (N, E)$, where N refers to the nodes and E to the edges. The nodes can be any code elements, such as variables, functions, and control flow elements inside the code. The relevant ones for tracing lineage are the variables and so-called intermediates, that describe results of cascaded function calls. Each edge from E depicts the estimated flow of data objects between two nodes in N. Since, we rely solely on static code analysis to generate the DFG, this graph is an estimation of the actual dataflow. We further categorize the edges into different types: (1) Caller, (2) Input, (3) Omitted, (4) Branch, (5) Loop, and (6) Function Call.

The "Function Call" edge type is introduced only for recursive functions; otherwise, function calls are replaced by the dataflow within the function's context. To handle operations involving branches and loops, we design `APEX-DAG` to be scope and block-aware. Note that functions and control-flow constructs, such as loops, have scopes. The translator is designed to maintain the state with respect to the individual contexts. Each context is either linked to a parent context or to the module context, which exists exactly once per script. Contexts can also be merged, such as when replacing function calls with actual dataflow or when processing branches and loops.

After generating the dataflow graph and tokenizing function names, the translator encodes this graph. The second box in Figure 3 shows one such result for the running example of this paper. Note that the nodes and edges are colored according to the predefined general types. This result is obtained without any supervision and code execution.

3.2 Detector & Estimator

The detector & estimator component of the system (❷) uses a graph attention network (GAT) (❼) that leverages node structure and edge embeddings. This enables the network to learn from the neighborhood of nodes and the structure surrounding each node, as well as from method names. We design a custom forward pass that emphasizes edges by concatenating the node embeddings, H_{nodes}, with the average edge embeddings, H_{edges} (❻):

$$H_{\text{concat}} = \text{concat}\left(H_{\text{nodes}}, \frac{1}{N_e}\sum_{i=1}^{N_e} H^{(i)}_{\text{edges}}\right)$$

Combining node and edge embeddings is crucial for inferring calls to libraries that may not be included in the training set. While the node embedding encode specifics about a variable or an intemediate, the edge embedding lets us reason about the usage of the variables, independent of the language and variable name. For instance, the APIs of `Pandas` and `Polars` are similar in their approach to reading a dataframe [3, 4]. This can be captured by the edge embedding.

The GAT is trained to classify nodes into different pipeline components, such as datasets, transformations, and exploratory data analysis parts, while also predicting edges in the graph. Additionally, the GAT addresses a weighting task, predicting how many tuples in a dataset are affected by an operation in the DFG or whether any rows are impacted. This approach results in a network capable of filtering out non-pipeline-related nodes and estimating the number of tuples affected by specific operations. This estimation supports data practitioners in debugging specific operations within large-scale codebases and emphasizes important transformations for compliance purposes. The filtered and weighted DFG as shown in the third box of Figure 3 serves as input for the annotator part. Note, that the colors encode lineage-relevant node and edge types.

3.3 Annotator

The generated dataflow graph also contains references to the original code fragment locations. The annotator uses these references to: a) highlight the relevant parts of the code (❽) referring to the different pipeline components, b) and maps it onto the input code. For a visualization of the highlighted code see Figure 1.

The results of APEX-DAG can be utilized in various ML life cycle management tasks, such as pipeline generation, optimization, or code refactoring. We implemented the system in Python, and the code is available on GitHub[1]. For pre-training the GAT, we created a dataset of 100.000 aDFGs, along with a smaller, labeled fine-tuning dataset. The labels are being obtained through a manual effort that is assisted through a paid LLM-instruct API. The label taxonomy is based on the taxonomy of the Code4ML dataset [7]. We collected scripts and code from various Kaggle [2] competitions and cleaned and filtered the JetBrains notebook dataset [6]. Both datasets will be published along the code once the labeling process is finished.

4 DEMONSTRATION

In this demonstration, we will showcase APEX-DAG and its applications. The first part of the demo focuses on interacting with APEX-DAG directly. To highlight APEX-DAG's capabilities, we initially employ a fully trained version of the system. The model and the corresponding library can also be downloaded from our github page [1].

Linage Visualization: The presenter will first develop a simple data science workflow for a simple pipeline use case, consisting multiple transformation steps on a Polars dataframe. Simultaneously, the file watcher of APEX-DAG will monitor the notebook for changes and execute APEX-DAG translator-annotator pipeline whenever a modification is detected. The generated linage DAG will be dynamically visualized as can be seen in Figure 3. The participants are encouraged to create their own pipelines to see how APEX-DAG adapts.

Generalized Linage Detection: After completing the workflow development and observing APEX-DAG's outputs, we switch to a version of APEX-DAG that has never been exposed to Polars. This way, we show that the underlying trained model is generalizable to unseen libraries at development time.

Inspecting the training process: In the final part of the demo, we will introduce the training process for our translator-annotator architecture. The participants will see the self-supervised label generation for the pre-training task and we will demonstrate how labelling the dataflow graphs for fine-tuning works. Participants will be able to see dataflow graphs generated directly from code, while the GAT is learning to predict node types and edges as can be seen in Figure 3.

[1] https://github.com/S-Eggers/APEX-DAG

5 CONCLUSION

This paper introduces APEX-DAG, a system developed to tackle the challenges of pipeline extraction and data lineage in machine learning workflows. Utilizing static code analysis and graph attention networks, APEX-DAG overcomes the limitations of execution-based methods and the reliance on predefined knowledge bases or library annotations. This approach offers scalability, adaptability to evolving library APIs and reliable inference across diverse computational contexts.

ACKNOWLEDGMENTS

This research has been supported under the grant BIFOLD24B.

REFERENCES

[1] 2025. Google Vertex AI. https://cloud.google.com/vertex-ai Accessed: 2025-16-01.
[2] 2025. Kaggle. https://www.kaggle.com/ Accessed: 2025-16-01.
[3] 2025. pandas. https://pandas.pydata.org/ Accessed: 2025-16-01.
[4] 2025. Polars. https://www.pola.rs/ Accessed: 2025-16-01.
[5] P. Buneman, S. Khanna., and T. Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT*.
[6] JetBrains Datalore. 2020. We Downloaded 10,000,000 Jupyter Notebooks from GitHub — This Is What We Learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/ Accessed: 2025-16-01.
[7] A. Drozdova, E. Trofimova, P. Guseva, A. Scherbakova, and A. Ustyuzhanin. 2023. Code4ML: a large-scale dataset of annotated Machine Learning code. *PeerJ Computer Science* (2023).
[8] S. Eggers. 2024. Automating Data Lineage and Pipeline Extraction. *VLDB* (2024).
[9] S. Grafberger, S. Guha, J. Stoyanovich, and S. Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *SIGMOD*. ACM.
[10] G. Harrison, K. Bryson, A. E. Bamba, L. Dovichi, A. H. Binion, A. Borem, and B. Ur. 2024. JupyterLab in Retrograde: Contextual Notifications That Highlight Fairness and Bias Issues for Data Scientists. CHI.
[11] M. Helali, N. Monjazeb, S. Vashisth, P. Carrier, A. Helal, A. Cavalcante, K. Ammar, K. Hose, and E. Mansour. 2024. KGLiDS: A Platform for Semantic Abstraction, Linking, and Automation of Data Science. In *ICDE*. IEEE.
[12] R. Ikeda and J. Widom. 2009. Data lineage: A survey. *Stanford University Publications* (2009).
[13] Z. G Ives and Y. Zhang. 2019. Dataset relationship management. In *CIDR*.
[14] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *SIGKDD*. ACM.
[15] Basel Committee on Banking Supervision. 2013. *Principles for Effective Risk Data Aggregation and Risk Reporting (BCBS 239)*. Technical Report. Bank for International Settlements. https://www.bis.org/publ/bcbs239.htm Accessed: 2025-01-26.
[16] J. Peuralinna. 2024. *Data Lineage in the financial sector.* Master's Thesis. Aalto University.
[17] J. F. Pimentel, L. Murta, V. Braganholo, and Juliana Freire. 2017. noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *VLDB* (2017).
[18] F. Psallidas, M. E. Leszczynski, M. H. Namaki, A. Floratou, A. Agrawal, K. Karanasos, S. Krishnan, P. Subotic, M. Weimer, Y. Wu, and Y. Zhu. 2023. Demonstration of Geyser: Provenance Extraction and Applications over Data Science Scripts. In *SIGMOD*. ACM.
[19] S. Schelter, J. Böse, J. Kirschnick, T. Klein, and S. Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. NeurIPS.
[20] S. Schelter, S. Guha, and S. Grafberger. 2024. Automated Provenance-Based Screening of ML Data Preparation Pipelines. *Datenbank-Spektrum* (2024).
[21] D. Xin, H. Miao, A. G. Parameswaran, and N. Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD*. ACM.
[22] Y. Zhang and Z. G Ives. 2020. Finding related tables in data lakes for interactive data science. In *SIGMOD*.