Addressing Data Management Challenges for Interoperable Data Science

Ilin Tolovski
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
ilin.tolovski@hpi.de

Tilmann Rabl
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tilmann.rabl@hpi.de

ABSTRACT

The development of data science pipelines (DSPs) has been steadily growing in popularity. While the increasing number of applications can also be attributed to novel algorithms and analytics libraries, the interoperability of new DSPs has been limited. To investigate this, we curated a corpus of over 494k GitHub Python repositories. We find that only 20% of the data science pipelines provide access to their input data and only 14% use a data backend. These findings highlight the key pain points in the development of interoperable DSPs. We identify five open data management challenges related to pipeline analysis, data access, and storage. We introduce Stork, a system for automated pipeline analysis, transformation, and data migration. Stork provides open data access while removing the human in the loop when reproducing results and migrating projects to different storage and execution environments. We analyze terabytes of DSPs with Stork and successfully process 72% of the pipelines, transforming 75% of the accessible datasets.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/stork.

1 INTRODUCTION

In recent years, collaborative development environments gained increasing relevance for open-source data analysis. Between 2017 and 2020, there has been a seven-fold increase in the number of publicly available data analytics notebooks on GitHub [41]. Our analysis shows that Python repositories on GitHub in general have doubled between the years of 2019 and 2021. Collaborative environments, such as Jupyter Notebooks, Google Collab, and Vizier, provide an easy-to-use interface for developing and sharing pipelines with reproducible results [7, 16, 20].

Despite the extensive use of data science frameworks, interoperability and reproducibility remain two significant challenges in the data science lifecycle [31]. Interdisciplinary environments highlight the need for interoperability and collaboration. Projects in the health domain, such as the Health Data Spaces Initiative present legal and technical barriers to the collaboration on data science projects between the scientists, medical personnel, and legislators. The pipelines and workflows are developed in a localized environment lacking open data access and significantly limiting the project interoperability [25, 26, 58].

To create a collaborative environment and interoperable pipelines, project resources need to have hosted data storage and an open execution environment. For each pipeline, the data engineer needs to locate, format, and transfer the data to a hosted storage system, such as a database, cloud storage, or remote file system. Additionally, they need to transform the pipeline to adjust the data access. Performing these steps manually incurs additional financial and technical migration cost. This prevents the interoperability and overall (re-)usability of data analysis projects [43, 54].

Chattopadhyay et al. [10] summarize the challenges for developing data science workloads, including the data setup, exploration and analysis, code management and archival, reliability, security, sharing and collaboration, reproducibility, and deployment. From the data management perspective, we identify five data management challenges. Specifically, *data setup*, *analysis*, *sharing*, *reproducibility and reusability*, and *security*.

To quantify the technical cost, we perform an analysis of 494,513 open-source GitHub projects. We determine the necessary steps to update or reproduce the pipelines in a different execution environment. When migrating to a collaborative environment, transferring the analysis scripts to a hosted solution like Jupyter Server is not sufficient for the results to be reproduced or the pipelines updated [20]. The data engineer needs to enable data access, format the data, and adjust the pipelines.

One data science repository in our corpus has three data science pipelines on average, accessing nine datasets in total. To transfer the data and adjust its access in the original pipelines, a data engineer needs to perform between 18 and 30 sequential steps, depending on the number of datasets. Executing each step manually is inefficient for pipelines ingesting more than one data file as well as for projects containing multiple pipelines. The reliance on manual pipeline adjustments presents a significant challenge for the interoperability of data science pipelines in collaborative environments.

Manual adjustments need to be performed on more than 80% of the data science repositories in our corpus. We observe a 5:1 discrepancy between the usage of analytical libraries and DBMS or cloud connectors, indicating that data science pipelines heavily rely on local data accesses. Out of the repositories accessing local datasets, only 20% provide access to the data. The rest have invalid data access even with the shared data or do not provide the data in the repository at all. These findings indicate that open-source data science projects are not designed to be reproducible or interoperable. To address the interoperability challenges and avoid the cost

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

of manual transformations, we propose developing an automated workflow based on code analysis and pipeline rewriting.

Related work has designed systems for automated code analysis based on static analyzers and large language models. Such approaches have been prominently used for automated program repair, bug detection, and class-level code generation [14, 30, 53]. To the best of our knowledge, we are the first to introduce a code analysis system designed for interoperability, modification of data access, and transformations in data science pipelines.

In this paper, we introduce Stork, a system for automated analysis, data migration, and transformation of data science pipelines. Stork uses static code analysis to detect data accesses, migrate data to a designated storage solution, and rewrite the pipeline. With Stork, we address key interoperability challenges, such as: *data access and setup*, *pipeline sharing*, as well as, *reproducibility and reusability of the data and pipelines*.

To design Stork, we performed an in-depth analysis of the open-source data science landscape. We curated a corpus of 494k Python repositories, which we use for the analysis and evaluation. Our findings show that Stork extracts data and operators on 72% of all openly available Python repositories. Stork provides a completely automated end-to-end pipeline analysis and data migration solution, replacing the human in the loop for pipeline analysis, data transformation, and migration. We compare Stork's automated analysis and code transformation to two large language models, one hosted on-premise (Llama3-8B), and GPT-3.5-Turbo, accessed via OpenAI's API [28, 36]. Stork shows three orders of magnitude faster analysis time. To the best of our knowledge, Stork is the first system that automates the complete interoperability workflow of pipeline analysis, data transformation, and pipeline rewrite. We summarize our contributions as follows:

- We curate and release a corpus of open-source repositories solving heterogeneous data processing tasks.
- We perform an in-depth analysis of the data science landscape focused on data management on over 494k Python projects.
- We identify five open data management challenges in the development of data science workflows.
- We present a system for automated pipeline analysis, data transformation, and migration. Our implementation can be accessed at https://github.com/hpides/stork.

The paper is organized as follows. We formalize the interoperability challenges in data processing pipelines in Section 2. In Section 3, we present the system design of Stork and the implementation details in Section 4. In Section 5, we evaluate Stork. In Section 7, we summarize our findings and potential future work.

2 INTEROPERABLE DATA ANALYTICS

In this section, we look into the data science landscape and investigate the surrounding interoperability challenges. We analyze a corpus of Python repositories available on GitHub from 2018 to 2023. We categorize them based on their dependencies and focus on the repositories with data analytics workloads. Based on our findings and related work, we define five data management challenges in the current data science landscape.

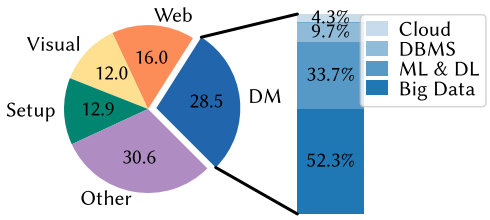


Figure 1: Library categories.

2.1 Pipeline Exploration

We curate a corpus of Python repositories published on GitHub between 2018 and October 2023 published under the MIT license [15, 18]. The corpus consists of 494,513 repositories with a compressed size of 3.5 TB. While related work analyzes individual pipelines and notebooks, we focus on complete data science repositories allowing us to observe cross-pipeline dependencies that cannot be read through processing individual files.

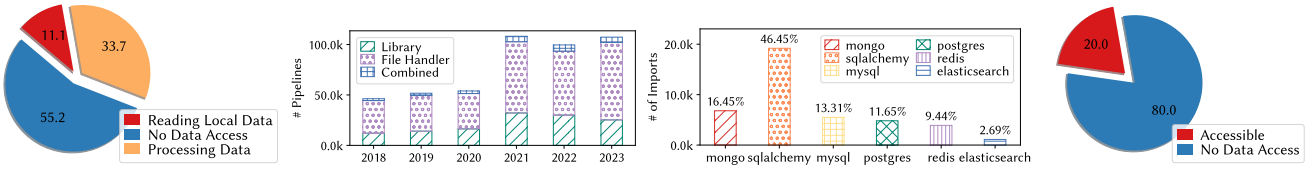
For each repository, we analyze the Python packages imported by the individual pipelines and create a minimal set of imported packages per repository. We observe that 8% of the repositories cannot be processed because of syntactical and versioning errors. Further in the analysis, we only consider repositories written in Python 3.0 and above.

We categorize the libraries based on their domain into five categories: *Data Management (DM)*, *Web Development*, *Visualization & Image Processing*, *Setup & Maintenance*, and *Other* libraries. The category distribution is shown in Figure 1. The *Data Management* libraries are the second most commonly used after the *Other* libraries, accounting for more than a quarter of all imports in our corpus of Python repositories. Related work has presented the popularity of machine and deep learning libraries, which account for a third of the *data management* libraries currently in use [41, 42]. While we confirm these findings, we also observe that the most used *DM* libraries are numerical libraries for efficient large data processing.

Additionally, we analyze the use of libraries interfacing with databases or cloud storage environments. We observe a low usage of connectors for databases (9.7%) or cloud storage (4.3%). Both categories combined have a 14% share of repositories that utilize any *DM* library. This shows a significant difference between the popularity of big data and machine learning libraries as tools for processing large data files and the necessary storage environments.

We observe a 5:1 discrepancy between repositories utilizing *DM* libraries and those storing the data in a hosted environment. This discrepancy suggests that data is often accessed locally, resulting in data access and project interoperability issues. We investigate this further by looking into the individual pipelines. In Figure 2a, we present the distribution of the pipelines reading data from storage devices compared to the pipelines processing the data and the code files that do not process data. The majority of code files (55%) are in the latter category. The read-and-write workloads constitute only 11% of the pipelines in the repositories, whereas a third of all pipelines are processing and analyzing data.

We look further into the most common ways of interacting with data in data science workloads. In our corpus, 36 % of all repositories



(a) Pipeline distribution. **(b) Data reading methods.** **(c) Distribution of DB connectors.** **(d) Dataset availability.**

Figure 2: Distribution of used data backends & availability of datasets.

contain at least one pipeline processing data. On this subset, we investigate the data reading methods used in each pipeline. In Figure 2b, we analyze the pipelines on a per-year basis. While we notice a significant increase in the number of data science pipelines created after 2020, we also observe that approximately two-thirds of pipelines read data with the native Python file manager. This analysis covers all files ingested into a Python workload. A third of the pipelines ingest data through data management libraries, such as `numpy`, `pandas`, or a database connector.

We analyze the distribution of database connectors used in data science pipelines. In Figure 2c, we observe that `SQLalchemy` accounts for almost half of the database connectors used. It provides an interface to several in-process and client-server databases. However, it is most commonly used as an `SQLite` connector, the most used in-process database, which stores data in a single file [48].

The convenience of using local files through Python file handlers and libraries operating in memory opens a set of interoperability and reproducibility questions. Specifically, the data availability, pipeline execution, and result reproducibility, are directly affected by the reliance on local file management. In our corpus, the majority of repositories in the dataset rely on local file accesses. We observe that only 20% of the accessed data is provided with the source code (see Figure 2d). This also confirms the significant discrepancy in the usage of libraries connecting repositories to storage environments. The lack of hosted data access in shared repositories makes it difficult to execute the pipelines in a new environment, and the results are difficult to reproduce. The low data availability, weak reproducibility standards, as well as the lack of interfacing with hosted storage solutions pose significant challenges that affect the overall interoperability of the current data science landscape. In Section 2.2, we define a set of data management challenges and propose an automated data science workflow.

2.2 Problem Formulation

Data and pipeline interoperability has been a significant hurdle for cross-institutional and international collaboration through the secondary use of data. Secondary data use is defined as analyzing data collected from previous studies and trials in different domains [25, 26]. A prominent example is the domain of health data with initiatives backed by the World Health Organization (WHO) and the European Union (EU). There have been several efforts to develop a framework that facilitates innovation and research by collaborating on the secondary use of health data [25, 26]. The European Union through the project Towards Europe Health Data Space (TEHDAS) has defined a set of legislative and technical barriers on the way toward a connected health data space [25]. The technical infrastructure and data management barriers include the lack of data access through research data centers, no standardized data formats, and poor data management standards for accessing and analyzing data across collaborating health centers. This leads to difficulties with interoperable data access, loss of information value, failed data analysis efforts, unreliable benchmarking, non-reproducible results, and significant financial overhead.

The infrastructure and data management barriers show a need for a framework that facilitates data interoperability as well as open access adhering to the FAIR (Findable, Accessible, Interoperable, Reusable) data principles [56]. To develop such a framework, we need to address several open interoperability challenges affecting the end-to-end data science lifecycle in a collaborative environment.

The data interoperability challenges between collaborating entities with different infrastructure and execution environments are a significant part of the data science lifecycle research in the data management (DM) and human-computer interaction (HCI) communities [10, 23, 24, 27, 31, 32, 40, 54, 55]. Both communities analyze the developments and opportunities in the data science lifecycle focusing on systems and interactivity, respectively. Chattopadhyay et al. [10] present a set of challenges from an HCI perspective.

In this work, we recognize five of them as open data management challenges. We focus on loading and processing data in multiple platforms (*Setup*), adapting data processing pipelines (*Exploration and Analysis*), code and pipeline reuse (*Share and Collaborate*, *Reproduce and Reuse*), and access control (*Security*). We look into the technical barriers through the lens of the five data management challenges and address them separately. In Listing 1 we present an example data science pipeline relevant to the health data space use case for which we present opportunities for improvement.

Setup: *Loading and processing data from multiple sources and platforms.* The management of projects in collaborative environments offered by major cloud vendors, such as Azure Notebooks, Google Collab, and Amazon SageMaker is well integrated with their object storage services [3, 16, 29]. Users synchronize their file storage by using these services, thus removing the challenge of transferring data and rewriting the pipelines. However, these setups are not suitable for locally stored data, or data with restricted access. In *Lines 7 and 8* of Listing 1, we define the source folders of two data files. Data is read in *lines 9 and 11*, in CSV and JSON format, respectively. The pipeline cannot be executed in a different environment because of the local data access in *Line 9*. To address the interoperable data access barrier, we propose opening the access and transferring local sources to a designated storage solution.

Explore and Analyze: *Adapting of data processing pipelines through repetitive steps denoting data reading and locality.* To facilitate open data access and analysis, the data engineer needs to read the data (*Lines 9* and *11*) and apply several preprocessing operations (*Lines 12-23*) before running statistical analysis. In Listing 1, we show an example with two data sources. However, doing so for several pipelines with multiple data sources becomes a repetitive and error-prone effort. To address the sequential and repetitive data adjustments, we propose an approach that analyzes the pipeline structure and automatically detects data reads to facilitate open data access.

```
1  import pandas as pd
2  import numpy as np
3  from sklearn.preprocessing import StandardScaler
4  from sklearn.linear_model import LogisticRegression
5  from sklearn.pipeline import Pipeline
6  from sklearn.metrics import accuracy_score
7  DATA_HOME = "/home/user/data"
8  DATA_WEB = 'https://example.com/user/data'
9  local_data = pd.read_csv(f'{DATA_HOME}/raw_data1.csv')
10 response = requests.get(f'{DATA_WEB}/raw_data2.json')
11 web_data = pd.read_json(StringIO(response.text))
12 train_data = pd.concat([local_data1, web_data]
13 for col in ["name", "DOB", "gender"]:
14     train_data[col] = train_data[col]
15     .apply(lambda x: f"{hash(x)}")
16 train_data["ccard_no"] = train_data["ccard_no"]
17     .apply(lambda x: str(x)[:-4] + "XXXX")
18 for col in ["ZIP"]:
19     train_data[col] += np.random.normal(0, 1)
20 train_data.to_csv("anonymized_data.csv", index=False)
21 pipeline = Pipeline([
22      ('scaler', StandardScaler()),
23      ('logreg', LogisticRegression())])
24 pipeline.fit(train_data[:, :-1], train_data[:, -1])
25 test_data = pd.read_csv(f'{DATA_HOME}/test_data.csv')
26 X_test = test_data.drop('target_column', axis=1)
27 y_test = test_data['target_column']
28 y_pred = pipeline.predict(X_test)
29 accuracy = accuracy_score(y_test, y_pred)
```

Listing 1: Data science script

Share and Collaborate: *Sharing of complete pipeline or parts of it in the form of code, data, or results.* Similarly to moving the data in a new storage solution in the environment setup, the pipeline also needs to be accessible to the users. To this end, the data must be stored in a file system, cloud object storage, or a database management system. Depending on the new storage system, the data needs to be transformed. In database management systems, there are several steps required to transform the data reads in *Lines 9 and 11* in Listing 1. The individual columns need to be transformed to database-specific datatypes, a database schema needs to be created, and the data transformed from a flat file to a relational table before writing it to the DBMS. Such steps are required for each data source in the pipeline. To address the lack of standardized data formats for sharing and collaboration, we automate the data transformation workflow and transfer the data to a hosted storage system.

Reproduce and Reuse: *Adapting pipelines and repositories to be executed in a new environment for code reuse and result replication.* Following the data and pipeline migration, data accesses and environment requirements need to be adapted before executing the pipeline in a new environment. From the data management perspective, managing the data reads is essential for the successful reuse of the pipeline and replication of results. *Lines 9 and 11* in Listing 1 need to be replaced by data reads for the transformed and relocated data. To allow data access that is independent of the execution environment, we query the new storage system. We show a DBMS example of the adjusted pipeline with a read query in Listing 4. *Lines 9 and 11* from Listing 1 are replaced by the respective code. To address pipeline interoperability, we propose a pipeline rewriter that connects to the data storage and modifies the data access.

Security: *Providing access control to localized and vulnerable data accesses in data processing pipelines.* When sharing a pipeline in a collaborative environment, the data access needs to be verified and managed to ensure authorized updates. By establishing user authentication, we prevent unwanted data corruption or pipeline changes. We address the potential vulnerabilities of open data access by allowing users to manage the authentication requirements.

We propose an automated workflow that incorporates the individual components framed in this section. The goal is to facilitate interoperable data access and collaborative pipeline development in use cases such as the open health data space. We aim to reduce the complexity of the data integration into data science pipelines and automate a sequence of manual tasks normally performed by data engineers. We present the system design in Section 3.

3 SYSTEM DESIGN

Our analysis in Section 2.1 shows that the majority of data science repositories do not provide an accessible environment to store and share the input data as well as their results.

To this end, we introduce Stork, a system for static pipeline analysis and data migration. Stork provides an automated workflow addressing the challenges presented in Section 2.2. From the system design perspective, we group these challenges into three segments: *Pipeline Analysis*, *Data Access and Formatting*, *Pipeline Rewrite and Access Management*. In Figure 3, we present the end-to-end Stork workflow. We present each stage in the following subsections.

3.1 Pipeline Analysis

We address the *Setup* and *Exploration and Analysis* by performing static code analysis on the DS pipeline. We manage the execution environment on different levels of granularity by addressing individual pipelines or projects containing multiple pipelines. We create an open and accessible execution environment for the pipelines by detecting the data sources and the data ingestion method.

The pipeline analysis consists of two phases: Abstract Syntax Tree(AST) Traversal and Generating an Intermediate Representation. In the two stages, Stork detects the data sources, verifies the access to the input data files, and records the reading and transformation of the input data. By designing the system in this way, we track the data updates throughout the pipelines and record it in a compact representation.

In the traversal phase, Stork initially reads and parses the abstract syntax tree (AST) of the complete pipeline. It then stores the operator tree of the input data and collects any input and transformation references. In Figure 4, we show a segment of the AST operator tree for *Line 9* from Listing 1. The fields necessary to track the lineage of the data read are highlighted in red. It enables Stork to process the pipeline structure and record data inputs and transformations. Stork traverses the code structure of the pipeline

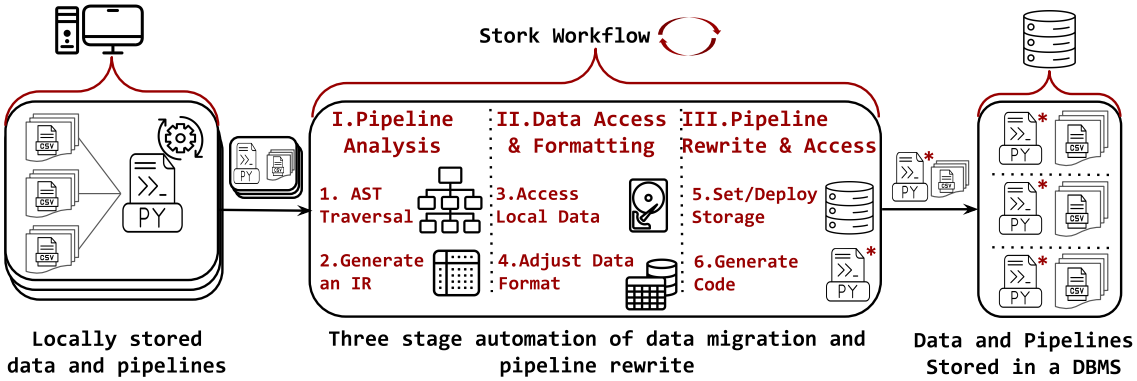


Figure 3: System overview of Stork.

and segments it into data access and execution parts. The traversal detects data reads from local files, while the segmentation stores references for the input data and tracks the transformations.

In DS pipelines, data is read from local files in several ways, including string paths, variable assignments, and references to external configuration files. The data sources are represented by a different set of nodes in the pipeline's AST. This requires adaptive filtering and variable extraction during the AST traversal. Stork differentiates between the data ingestion nodes when the data source is referenced in the pipeline. The analysis of the AST cannot capture data inputs provided in external files and as runtime arguments. Such setups are out of the scope of individual pipeline analysis.

In the second phase of the pipeline analysis, Stork stores the input and operator references in a compact intermediate representation, saving storage and subsequent traversal times. We collect the data input statements and a reference to the transformations applied to the data. Once the data reads and transformations have been recorded in our intermediate representation, the data pipeline(s) are dynamically transformed and prepared for the *Data Access and Formatting* stage of the Stork workflow. We describe the complete pipeline analysis workflow and its implementation in Section 4.1.

3.2 Data Access & Formatting

In the second stage of the Stork workflow, we access, format, and transfer the existing data to a new storage environment. We enable concurrent access to the input data and changes to the pipeline. In this stage, we address the *Share and Collaborate*, and *Reproduce and Reuse* challenges. When a storage environment is selected, Stork parses through the structure of the data and issues a data transfer. There are cases where the data needs to be formatted before it can be written to a new storage backend. One such example is transferring a flat file to a relational database management system. Upon parsing the structure of the data, Stork transforms it into a relational table before issuing the data transfer.

Stork allows for various implementations of storage backends and provides an extensible interface. In this paper, we consider relational database management systems, cloud object storage, and local file systems. Transforming the data and interfacing with multiple storage backends incurs several implementation challenges, which we present in more detail in Section 4.3.

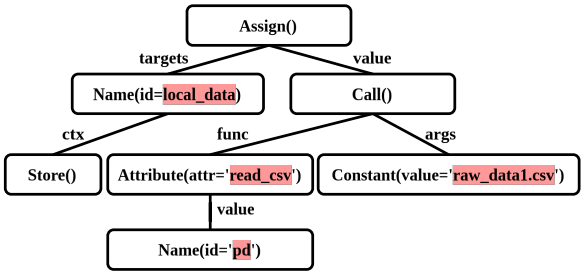


Figure 4: AST representation of a single data assignment.

3.3 Pipeline Rewrite & Access Management

In the third stage of the workflow, Stork generates the necessary connections to the new storage environment and rewrites the data read operations in the pipeline. With this stage, we enable the users to reuse the pipeline and reproduce its results, addressing the *Reproduce and Reuse* challenge.

Following the static analysis, we identify the sections of the pipelines that are ingesting data from local files. Once the data has been transferred to a new storage environment, Stork rewrites the pipeline to reroute data access to the new environment. We reuse our intermediate results from the pipeline analysis stage and access the data on the server rather than the local files. The adjusted data access enables the pipeline to be shared and executed in different environments. This translates to executing workloads on more powerful hardware, as well as sharing the pipeline with other collaborators for synchronized development.

The rewritten pipeline can be transferred to a database server, where the database also serves as an execution environment. Having the data and the pipeline co-located in a single DBMS allows us to further improve the performance and interoperability of the pipeline by utilizing the data locality.

When generating the connections to the new storage environments, Stork assumes permissive access to the environment. However, in cases when additional authentication is required, we provide managed data access to the new environments, introducing a layer of *Security* in our system. We discuss the implementation details of this workflow stage in Section 4.4.

4 SYSTEM IMPLEMENTATION

In this section, we present the implementation of Stork. In addition to the three workflow stages presented in Section 3, we discuss the data access patterns, as well as the storage backend implementation.

4.1 Pipeline Analysis

To obtain all necessary data access information in the pipeline, we perform static code analysis. We analyze the code through its abstract syntax tree (AST) in a single pass that consists of two phases. First, we traverse the AST to detect all `Import` and `Assignment` nodes. We then filter out all imports and assignments that do not contain any data artifacts and store them in an intermediate representation.

In the traversal stage, we extract all `Import` and `Assignment` nodes containing the library imports and data assignments. These nodes carry references relevant to the data analytics steps in the pipelines. From the `Import` nodes we obtain information on the libraries used for reading and processing the data. We collect the libraries to replicate the execution environment for the pipeline.

When traversing the `Assignment` nodes, we determine whether we have access to the data, where it is stored, and which methods are used to read and process it. The `Assignment` nodes also contain information orthogonal to the data reads, such as instances of classes, variables, and execution of arithmetic operations.

In the second stage, we filter such assignments by analyzing the contents of the `Assignment` nodes in a single-pass filtering stage. In Listing 2, we show a Pythonic AST representation of the operator tree shown in Figure 4. It consists of several nested nodes for variable names, method calls, attributes, and constants.

```
Assign(
    targets=[Name(id='local_data', ctx=Store())],
    value=Call(
            func=Attribute(
                value=Name(id='pd', ctx=Load()),
                attr='read_csv',
                ctx=Load()),
            args=[Constant(value='/home/user/data/
     raw_data1.csv')])
```

Listing 2: An Assignment node in AST representation

Each of the nested nodes contains additional value, attribute, and context fields, making repeated traversals costly and impractical. We traverse the AST once and filter the relevant `Assignment` nodes in a compact representation. We show the representation of the `Assignment` node in Listing 3.

We use the intermediate representation to detect the parts of the pipeline that read and process the data. Storing the information in this format provides reference points for the data reads. We track and use these reference points to process data read by different access patterns. For each access pattern, Stork creates different references for the data artifacts ingested in the pipeline.

```
{'variable':'local_data',
'data_source': {
    'func_call': {'from': 'pd', 'method': 'read_csv'},
    'data_file': ['/home/user/data/raw_data1.csv'],
    'params': []}}
```

Listing 3: Representation of an Import and Assignment node

We obtain the referenced artifacts and transform them before migrating them to a new storage environment described in Section 4.3. The data references are then used for the pipeline rewriting stage described in Section 4.4.

4.2 Data Access Patterns

During the analysis stage, we distinguish between the different data access patterns, *string path*, *variable reference*, and *runtime data access*. Each pattern results in a different operator tree of `Assignment` nodes generated in the AST. In the case of data access through *string paths*, static code analysis is sufficient to parse the respective node and retrieve the file stored on the path. In cases when the data path is passed as a reference to another variable, a macro variable, or accessed from a configuration file, we process the references leading to the data path.

We extract the data path from a local variable defined in the pipeline by mapping the variable and its latest assignment. Stork then reads the value of the data path that is referenced by the variable without executing any part of the pipeline. Once the referenced mapping has been read, the data path value is extracted the same way as in the case of accessing raw strings.

Traversing through the `Assignment` nodes and extracting the data path values from variable references allows Stork to analyze the pipeline statically, without compiling or executing the pipeline. Bypassing compilation or execution removes the need to include library imports and potentially resolve any dependency conflicts. This approach allows us to retrieve absolute data paths also in cases where partial or relative paths are concatenated to a stem path.

4.3 Data Transformations

The references to the data paths from the AST traversal are stored in an intermediate representation and used to access the data files that need to be transferred. Before initializing the data transfer, we run a data integrity check to ensure the correct data format. Depending on the new storage environment, we perform several data formatting steps. In its current implementation, Stork supports data formatting and transfers to relational database management systems, cloud object storage, and local file systems.

Conversion to Relational Data. In a DBMS scenario, the data formatting includes schema inference and table creation. Data science pipelines and notebooks process data with dataframes in Python. They are ingested in different formats, such as comma-separated values, text tables, parquet, or zip-compressed files. These formats are read into dataframes and subsequently NumPy arrays. To process them, the target format needs to be compatible with the dataframe schema used in the pipeline. The set of datatypes supported by the dataframe interface in Python is significantly more constrained than the datatypes supported by frequently used database systems. For example, the dataframe interface supported by `pandas` has 12 general-purpose datatypes, whereas `Postgres` supports 43 general-purpose datatypes.

When a data reading method, such as `read_csv` or `read_table` is called with a file path as an argument, the reader infers the datatypes for each column upon memory allocation for the created dataframe. In case of data conflicts in a column, the library uses the `object` datatype as the default. Stork uses the inferred datatypes

Table 1: Mapping between Dataframe and Postgres datatypes

DF dtype	PSQL dtype
object	varchar
(u)int64	bigint
(u)int8, (u)int16	smallint
(u)int32	integer
float16, float32	real
float64	double precision

by the dataframe interface to create a table schema suitable for the stored data and compatible with the operators in the pipeline.

For relational database systems, Stork has a datatype matching tool, that ensures each column in the dataframe is represented by an equivalent datatype in the schema. The datatype matching is depicted in Table 1. We have implemented the datatype matching for the datatypes supported by Postgres.

Transferring Flat Files. When transferring data to cloud object storage systems and local filesystems, Stork processes flat files, such as CSV files, parquet files, and zip-compressed libraries. The files are parsed to ensure the correct format. Other than checking the structure of the file, there are no additional data cleaning or preprocessing steps on the raw files at this stage.

When using a cloud object storage system, Stork prepares the data transfer by initializing the system access, verifying the user credentials and destination path. We validate the account credentials, traverse the bucket structure, and transfer the files. Transferring files to another filesystem works analogously.

4.4 Pipeline Rewrite & Access Management

Through the static analysis, Stork references the sections of the pipelines that are ingesting the data from local files. Once the data has been moved to a new storage environment, our system rewrites the pipeline to reroute the data access to the server rather than look for the data locally. We use the referenced sections from the pipeline analysis stage (see Sections 3.1 and 4.1) and access the data on the server rather than the local files.

To establish a connection to the storage environment, we generate template code in the pipeline necessary to grant read rights on the server. Depending on the data backend, we generate either a database connection through the driver or provide the access keys to the cloud storage. The template code provides access rights to the new data storage. Additionally, we adjust the data reads in the pipeline. We rewrite the local data access with a query from the new backend. An example of a rewritten pipeline code when using a Postgres DBMS as the data backend is shown in Listing 4.

4.5 Limitations of Static Pipeline Analysis

Traversing through the AST of a pipeline without compiling or executing the pipeline limits the analysis of read statements. The execution of a pipeline with external arguments cannot be resolved since the variable values are not available during the AST traversal step. Such examples include data inputs provided at runtime, from configuration files, or imported from other pipelines. In these cases, the values cannot be retrieved without a partial or complete execution of the code with the external input. Inspecting efficient ways of dynamic code execution and analysis is out of the scope of this paper.

```
dbms_connector = DBMSConnector(pipeline, config)
train_data_1 = pd.read_csv(dbms_connector
    .execute("SELECT * FROM raw_data1"))
train_data_2 = pd.read_json(dbms_connector
    .execute("SELECT * FROM raw_data2"))
```

Listing 4: Rewritten data read operation in a script

5 EVALUATION

In this section, we evaluate the performance of Stork on a curated corpus of open-source data science pipelines. We outline our experimental setup and the criteria for forming the corpus. We then present the main findings from our evaluation.

5.1 Experimental Setup

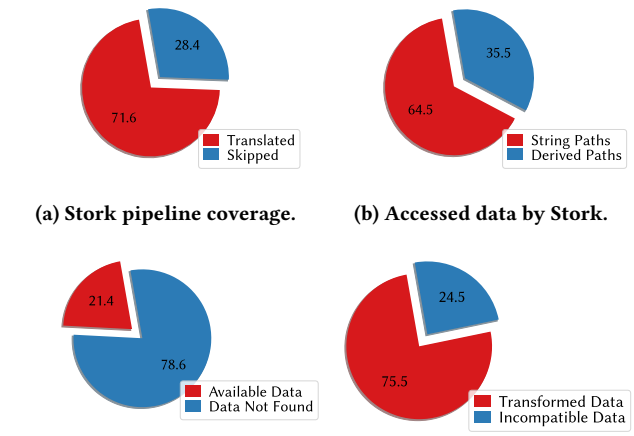
We crawled GitHub for projects with varying numbers of pipelines and data sources. Before curation, the corpus included 494,513 repositories for a compressed size of 3.5TB. In Section 2, we present the curation and analysis of the complete set of repositories. In this section, we utilize our curated corpus of 54,757 repositories, which include a total of 152,554 pipelines. We use it to evaluate the effectiveness of Stork in the following subsections. We ran the complete corpus analysis with Stork, the relational data transformations, and the storage backend comparison on two different x86 server configurations, 1) 2x Intel Xeon Gold 5220S with 18 cores and 96GB of RAM, and 2) 2x AMD EPYC 7742 with 64 cores and 512GB of RAM. All experiments were executed using 36 threads. For hosting a Llama3-8B model, we use a server with AMD Ryzen Threadripper PRO 3995WX, 64GB of RAM, and two NVIDIA RTX A5000 GPUs with 24GB HBM2 memory.

5.2 Analyzing the GitHub Corpus

We analyze Stork's workflow on a curated dataset of 54,757 GitHub repositories defined in Section 2. In this experiment, we analyze the complete corpus and run the end-to-end workflow. We evaluate Stork in three separate stages: *end-to-end performance, data availability, and data transformations.* Stork ingests 91% of the repositories. Upon inspection of a sample of the remaining repositories, we found that the analysis had failed due to syntactical and import errors, rendering the repositories not executable or translatable.

End-to-end Performance. In Figure 5a, we present the results for measuring the success rate of Stork on all pipelines from the repository corpus. We analyze the repositories on a per-year basis. We evaluated Stork on 56,444 pipelines, ingesting data from strings as well as derived paths from variable references. Stork successfully processes 71.6% of the pipelines (see Figure 5a). For this set of pipelines, Stork detects the data ingestion in the pipeline and transforms the pipeline. The majority (65%) of the analyzed pipelines are reading data through string paths, whereas 35% access the data through paths derived from variable references (see Figure 5b). External arguments are the limiting factors for analyzing the pipeline without compiling or executing the code, as shown in Section 4.5.

Data Availability. In the second stage, Stork accesses the detected datasets and transfers them to a new storage medium, where the pipeline can be reproduced. However, our analysis in Section



(a) Stork pipeline coverage. (b) Accessed data by Stork.

(c) Data availability in pipelines. (d) Data transformations.

Figure 5: Analysis of the GitHub Corpus with Stork.

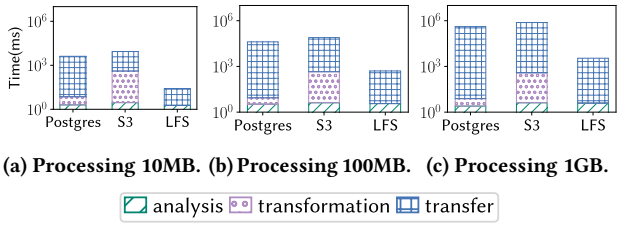
2.1 shows that only 20% of datasets referenced in the pipelines are available in the repository. In Figure 5c, we present the percentage of available datasets found and accessed by Stork. We observe that only 21.4% of the datasets recognized by Stork are accessible in the repositories. From the data reads, we observe that the rest of the accessed data are stored locally, thus rendering the pipelines uploaded on GitHub not reproducible. For the detected datasets, Stork transfers the data to a hosted storage and rewrites the pipeline to access the data from the new storage medium.

Data Transformations. In this experiment, we transform flat files into relational tables and transfer the data to a DBMS. We evaluate the schema inference on the set of available datasets in our corpus. Stork detects 82% of all datasets as structured data that can be transformed into a relational format. It translates and transfers 75% of this subset (see Figure 5d). We observe several recurring errors in the data that can be improved. Date encoding issues and using special characters in column names account for the majority of errors, for which we perform encoding checks and column name sanitation. Data cleaning operations on the raw data of individual datasets are out of the scope of this paper.

5.3 Storage Backend Comparison

We break down Stork's end-to-end runtime for three storage environments: DBMS (*Postgres*), cloud object storage (*AWS S3*), and a local file system. We run the workflow on three pipelines processing 10MB, 100MB, and 1GB of data, respectively. In Figures 6a - 6c, we show the duration breakdown for the three storage backends.

We observe that the pipeline analysis stage has a mean runtime of 3ms, amounting to a maximum of 4% of the total runtime when using a local file system and transferring 10MB (see Figure 6a). The data transformation and schema inference for Postgres take up to 3x longer than the translation across all data sizes, with a maximum runtime of 6ms. When using S3, creating the HTTP request takes up to 5% of the total runtime, or 400ms on average. In this experiment, the bottleneck is the data transfer stage, with the end-to-end time scaling linearly with the increase in data size.



(a) Processing 10MB. (b) Processing 100MB. (c) Processing 1GB.

Figure 6: Runtime breakdown of Stork.

Stork's system overhead of pipeline analysis and data transformation reaches a maximum of 5% of the total runtime (see 6a), i.e, 2ms and 405ms when using LFS and S3, respectively. The maximum overhead when using Postgres is 8ms, or less than 1% of the runtime for all data sizes.

5.4 Analyze Pipelines with LLM

Open-access large language models (LLMs) have been used for code generation from user input. In this section, we evaluate the effectiveness and efficiency LLMs for analyzing the code structure of a pipeline. Specifically, we evaluate Stork's performance against LLMs when analyzing data science pipelines. We focus only on the pipeline analysis from the end-to-end Stork workflow since the LLM has no execution rights, nor access to the dataset. We analyze a set of 52 representative pipelines reading data via `strings`, `variables`, and `external arguments`. We compare Stork against two LLM versions, the open-sourced Llama-3 with 8B parameters hosted on-premise, and GPT-3.5-Turbo, accessed via the online API [28, 36].

The LLM workflows consist of 1) creating a structured prompt to the model, 2) loading the model weights, and 3) inference of the output. When creating the prompt, we set a code analysis context for the model. In Listing 5 we show the context, prompt message, and a placeholder for the pipeline code.

```
1 ["role": "system", "content": {"You are a code analyzer
      that detects data inputs in Python workloads.}"
2 "role": "user", "content": "Given the following code,
      detect the lines where data is read from external
      files, and return a list of all paths to the data."
3 "role": "user", "content":{PIPELINE_CODE}]
```

Listing 5: Evaluation prompt.

The desired output from this prompt is a structured list of data paths, that can be used as an input to the rewriting and data transfer workloads. We measure the inference time of the LLMs to generate the list of data inputs in the pipeline. In Figure 7 we show the inference times for both LLMs and the analysis time for Stork.

We observe that Stork is up to three orders of magnitude faster for the pipeline analysis workflow. For our hosted LLM, we reload the weights to minimize the variance between the LLM outputs. The reloading time is not measured towards the total inference time. For GPT-3.5-Turbo, we report the request return time after the prompt has been generated. In terms of coverage and accuracy of the answers, both LLM solutions detect the data read operations in the pipeline, albeit at a significantly higher runtime.

We note that using an LLM for such a workload requires an additional manual effort. The data transfer, as well as the insertion of the new data location in the pipeline, needs to be done by the data

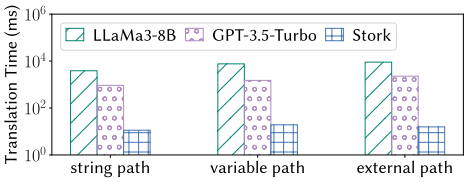


Figure 7: Code Analysis with Stork and LLMs.

scientist. In comparison, Stork automates the complete analysis, transformation, rewrite, and migration workflow. By utilizing Stork instead of an LLM, we achieve an analysis speedup of three orders of magnitude. While LLMs have found great use in code generation, completion, and workflow automation tasks, in our use case the benefits come at significant runtime and interoperability costs.

6 RELATED WORK

Bommarito and Bommarito analyze the usage of the complete Python Package Index (PyPI) [5]. Other case studies provide insights into the development of the data science landscape for open-source ML products, library usage, and software engineering practices [1, 4, 13, 33, 38, 42, 46]. Our analysis is focused on the combined usage of data analytics and database libraries and their contribution to an interoperable data science lifecycle. We utilize these findings to increase the overall interoperability in the data science lifecycle.

Data science pipelines have been analyzed and processed through automated provenance tracking [17, 34, 37, 39, 45]. Furthermore, there have been several approaches for static analysis of data processing pipelines to detect data leakages and analyze data transformations [6, 35, 50, 51]. Compared to these approaches, Stork utilizes the AST of the pipeline to detect data reads and input operations, as well as to recursively rewrite the pipeline.

In the area of data and pipeline migration, proprietary and open-source ETL solutions have been available [2, 9, 12, 22, 44, 49]. Such tools provide a low- or no-code development environment for data science pipelines with a semi-automated storage integration. Defining the data inputs and their interaction with the pipeline are done manually, together with the formatting and management of the data. While such solutions ease the development of data science pipelines, Stork enables fully automated data migration and pipeline rewrite of completed projects to new storage mediums.

Static code analysis and generation solutions have been introduced for the use cases of code repair, error detection, as well as class-level code generation [11, 57]. Large Language Models (LLMs) have also been used for code repair, analysis, and generation [14, 30, 52]. We utilize AST analysis for managing data accesses and transformations in the pipelines. To the best of our knowledge, we are the first to build an end-to-end system for this use case.

Collaborative environments, such as Jupyter Server, and Google Collab offer integration with cloud storage backends [16, 21] and to a lesser extent with distributed cloud systems [8, 19, 20]. Cloud providers have also developed data science tools offering exclusive access to their storage backends [3, 29, 47].

Stork bridges the gap between semi-automated ETL tools and cloud-dependent notebook environments. By performing static code analysis, Stork automates the data migration, provides open access to it, and rewrites the pipeline. It increases project interoperability and facilitates collaboration in a fully automated fashion.

7 CONCLUSION

In this paper, we analyze the landscape of data science repositories from 2018 to 2023. While observing an increase in the number of data science pipelines, we recognize limited data and pipeline interoperability, summarized in five data management challenges. Based on our analysis, we propose Stork, a system for automated pipeline analysis and data migration on different storage backends. Stork works out of the box for more than 70% of the existing data science pipelines while outperforming LLMs as an alternative code analysis and workflow automation tool. In future work, we will further extend the pipeline translation coverage of Stork and research opportunities of migrating parts of the pipeline automatically into database systems to enable efficient hybrid execution.

REFERENCES

[1] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. 2019. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. arXiv:1909.00084 [cs.DB]

[2] Sajid Alam, Nok Lam Chan, Laura Couto, Yetunde Dada, Ivan Danov, Deepyaman Datta, Tynan DeBold, Jitendra Gundaniya, Jannic Holzer, Stephanie Kaiser, Rashida Kanchwala, Ankita Katiyar, Ravi Kumar Pilla, Amanda Koh, Andrew Mackay, Ahdra Merali, Antony Milne, Huong Nguyen, Vladimir Nikolic, Nero Okwa, Juan Luis Cano Rodríguez, Joel Schwarzmann, Dmitry Sorokin, Jo Stichbury, and Merel Theisen. 2023. *Kedro*. https://github.com/kedro-org/kedro

[3] Amazon. 2023. *AWS Sagemaker*. Amazon. Retrieved March 16, 2023 from https://aws.amazon.com/pm/sagemaker/

[4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[5] Ethan Bommarito and Michael Bommarito. 2019. An Empirical Analysis of the Python Package Index (PyPI). arXiv:1907.11073 [cs.SE]

[6] Laurent Boué, Pratap Kunireddy, and Pavle Subotić. 2023. Automatically Resolving Data Source Dependency Hell in Large Scale Data Science Projects. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 1–6. https://doi.org/10.1109/CAIN58948.2023.00009

[7] Mike Brachmann, Carlos Bautista, Sonia Castelo, Su Feng, Juliana Freire, Boris Glavic, Oliver Kennedy, Heiko Müeller, Rémi Rampin, William Spoth, et al. 2019. Data debugging and exploration with vizier. In *Proceedings of the 2019 International Conference on Management of Data*. 1877–1880.

[8] Michael Brachmann and William Spoth. 2020. Your notebook is not crumby enough, REPLace it. In *Conference on Innovative Data Systems Research (CIDR)*.

[9] ByteHub. 2023. *ByteHub Feature Store*. ByteHub AI. Retrieved September 29, 2023 from https://github.com/bytehub-ai/bytehub

[10] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–12. https://doi.org/10.1145/3313831.3376729

[11] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, Vasteras, Sweden, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

[12] Nachiket Deo, Boris Glavic, and Oliver Kennedy. 2022. Runtime provenance refinement for notebooks. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*. 1–4.

[13] Helen Dong, Shurui Zhou, Jin L.C. Guo, and Christian Kästner. 2021. Splitting, Renaming, Removing: A Study of Common Cleaning Activities in Jupyter Notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 114–119. https://doi.org/10.1109/ASEW52652.2021.00032

[14] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the*

IEEE/ACM 46th International Conference on Software Engineering. ACM, Lisbon Portugal, 1–13. https://doi.org/10.1145/3597503.3639219

[15] GitHub. 2023. *GitHub Official Website*. GitHub. Retrieved August 30, 2023 from https://github.com/

[16] Google. 2023. *Google Collab*. Google. Retrieved March 16, 2023 from https://colab.research.google.com/

[17] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. 2021. Lightweight inspection of data preprocessing in native machine learning pipelines. In *Conference on Innovative Data Systems Research (CIDR)*.

[18] Open Source Initiative. 2023. *The MIT License*. open source initiative. Retrieved August 30, 2023 from https://opensource.org/license/mit/

[19] Jupyter. 2023. *Jupyter Hub*. Jupyter. Retrieved September 29, 2023 from https://jupyter.org/hub

[20] Jupyter. 2023. *Jupyter Notebooks*. Jupyter. Retrieved September 29, 2023 from https://jupyter.org/

[21] Kaggle. 2023. *Kaggle Notebooks*. Kaggle. Retrieved March 16, 2023 from https://www.kaggle.com/code

[22] Kedro. 2023. *Kedro Official Website*. Kedro. Retrieved September 29, 2023 from https://kedro.org/

[23] David Koop and Jay Patel. [n.d.]. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. ([n. d.]).

[24] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Dunedin, New Zealand, 1–11. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[25] Abboud Linda, Cosgrove Shona, Kesisoglou Irini, Richards Rosie, Soares Flavio, Pinto Cátia, Bogaert Petronille, and Bowers Sarion. 2021. *Summary of results: case studies on barriers to cross-border sharing of health data for secondary use*. Technical Report. TEHDS Consortium.

[26] Kalliola Markus, Drakvik Elina, and Nurmi Maria. 2023. *Advancing Data Sharing to Improve Health For All in Europe*. Technical Report. SITRA.

[27] Andrew M Mcnutt, Chenglong Wang, Robert A Deline, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–16. https://doi.org/10.1145/3544548.3580940

[28] Meta. 2024. *Llama-3-8B*. Meta. Retrieved May 30, 2024 from https://huggingface.co/meta-llama/Meta-Llama-3-8B

[29] Microsoft. 2023. *Azure Notebooks*. Microsoft. Retrieved March 16, 2023 from https://visualstudio.microsoft.com/vs/features/notebooks-at-microsoft/

[30] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2023. SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models. arXiv:2310.18532 [cs]

[31] Michael Muller and Angelika Strohmayer. 2022. Forgetting Practices in the Data Sciences. In *CHI Conference on Human Factors in Computing Systems*. ACM, New Orleans LA USA, 1–19. https://doi.org/10.1145/3491102.3517644

[32] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for Engineered Software Projects. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[33] Nadia Nahar, Haoran Zhang, Grace Lewis, Shurui Zhou, and Christian Kästner. 2023. A Dataset and Analysis of Open-Source Machine Learning Products. *arXiv preprint arXiv:2308.04328* (2023).

[34] 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* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1542–1551. https://doi.org/10.1145/3394486.3403205

[35] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Orlando, FL, USA) *(SOAP 2023)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3589250.3596145

[36] OpenAI. 2024. *GPT-3.5-Turbo*. OpenAI. Retrieved May 30, 2024 from https://platform.openai.com/docs/models/gpt-3-5-turbo

[37] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2019. A Survey on Collecting, Managing, and Analyzing Provenance from Scripts. *ACM Comput. Surv.* 52, 3, Article 47 (jun 2019), 38 pages. https://doi.org/10.1145/3311955

[38] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. *Empirical Softw. Engg.* 26, 4 (jul 2021), 55. https://doi.org/10.1007/s10664-021-09961-9

[39] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[40] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 507–517. https://doi.org/10.1109/MSR.2019.00077

[41] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (jul 2022), 30–37. https://doi.org/10.1145/3552490.3552496

[42] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. 2019. Data Science through the looking glass and what we found there. arXiv:1912.09536 [cs.LG]

[43] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. 2022. Workflow Analysis of Data Science Code in Public GitHub Repositories. *Empirical Software Engineering* 28, 1 (Nov. 2022), 7. https://doi.org/10.1007/s10664-022-10229-z

[44] Rivery. 2023. *Rivery Cloud ELT Tool Official Website*. Rivery. Retrieved September 29, 2023 from https://rivery.io/

[45] Lukas Rupprecht, James C Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. 2020. Improving reproducibility of data science pipelines through transparent provenance capture. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3354–3368.

[46] Marius Schlegel and Kai-Uwe Sattler. 2023. Management of Machine Learning Lifecycle Artifacts: A Survey. *SIGMOD Rec.* 51, 4 (jan 2023), 18–35. https://doi.org/10.1145/3582302.3582306

[47] Snowflake. 2023. *Snowpark API*. Snowflake. Retrieved February 15, 2023 from https://docs.snowflake.com/en/developer-guide/snowpark/index

[48] SQLite. 2024. *SQLite*. SQLite. Retrieved February 28, 2024 from https://www.sqlite.org/mostdeployed.html

[49] Stitch. 2023. *Stitch Official Website*. Stitch. Retrieved September 29, 2023 from https://www.stitchdata.com/

[50] Pavle Subotić, Uroš Bojanić, and Milan Stojić. 2022. Statically detecting data leakages in data science code. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 16–22.

[51] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A Static Analysis Framework for Data Science Notebooks. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3510457.3513032

[52] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2024. Bugs in Large Language Models Generated Code: An Empirical Study. arXiv:2403.08937 [cs]

[53] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture?: A Structural Analysis of Pre-Trained Language Models for Source Code. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2377–2388. https://doi.org/10.1145/3510003.3510050

[54] Dakuo Wang, Q. Vera Liao, Yunfeng Zhang, Udayan Khurana, Horst Samulowitz, Soya Park, Michael Muller, and Lisa Amini. 2021. How Much Automation Does a Data Scientist Want? arXiv:2101.03970 [cs]

[55] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. 2019. Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–24. https://doi.org/10.1145/3359313

[56] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data* 3, 1 (2016), 1–9.

[57] Zhaogui Xu, Ju Qian, Lin Chen, Zhifei Chen, and Baowen Xu. 2013. Static Slicing for Python First-Class Objects. In *2013 13th International Conference on Quality Software*. IEEE, Najing, China, 117–124. https://doi.org/10.1109/QSIC.2013.50

[58] Amy X. Zhang, Michael Muller, and Dakuo Wang. 2020. How Do Data Science Workers Collaborate? Roles, Workflows, and Tools. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (May 2020), 1–23. https://doi.org/10.1145/3392826