




**SciPy 2025**

July 7 - July 13, 2025

Proceedings of the 24<sup>th</sup>  
Python in Science Conference  
ISSN: 2575-9752

# Python is all you need: an overview of the composable, Python-native data stack

Deepyaman Datta<sup>1</sup>  

<sup>1</sup>Kedro

## Abstract

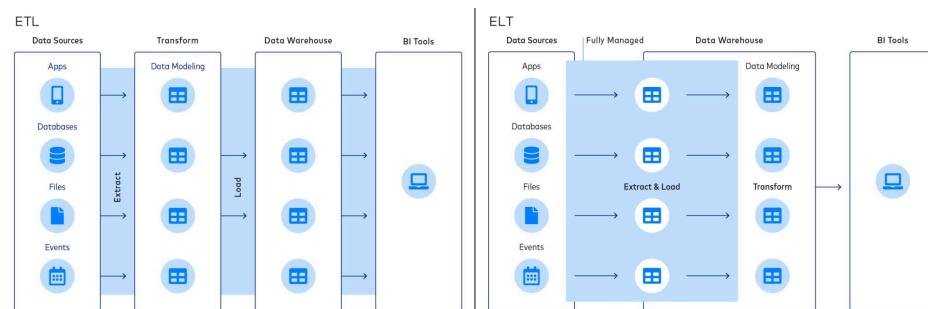
For the past decade, SQL has reigned king of the data transformation world, and tools like dbt have formed a cornerstone of the modern data stack. Until recently, Python-first alternatives couldn't compete with the scale and performance of modern SQL. Now Ibis can provide the same benefits of SQL execution with a flexible Python dataframe API.

Through new integrations, Ibis supercharges existing open-source libraries like Kedro and Pandera. Combined, these technologies enable building and orchestrating scalable data engineering pipelines without sacrificing the comfort and other advantages of Python.

**Keywords** data engineering, modern data stack, composable data stack, data ingestion, data transformation, data validation

## 1. INTRODUCTION

By 2016, the modern data stack was a well-defined idea [1] that had begun to take hold of the data engineering community. Specifically, it represented a set of products that enabled end-to-end data transformation *in the cloud* using SQL. This mindset shift was driven by the advent of cloud-based analytics databases like Amazon Redshift, Google BigQuery, and Snowflake [2] and the subsequent rise of the extract, load, transform (ELT) paradigm. Compared to the extract, transform, load (ETL) workflows that had become ubiquitous since their invention in the 1970s [3], ELT leveraged the power and efficiency of cloud data warehouses in performing transformations [4], combined with the simplicity of managed compute and processes. The differences between ETL and ELT can be seen in [Figure 1](#) below.



**Figure 1.** In ETL workflows (left), data is modeled and transformed before loading it into the data warehouse. In contrast, in ELT workflows (right), data sources are extracted and loaded into the data warehouse before building transformations to create data models for analytics [3].

Over the next few years, the tooling surrounding the modern data stack matured. Products like Fivetran and Airbyte solidified their position as data ingestion tools, and various data quality and observability vendors offered solutions for ensuring data correctness and completeness as part of the end-to-end process. Business intelligence (BI) tools that offered

**Published** Jul 10, 2025

**Correspondence to**  
Deepyaman Datta  
[deepyaman.datta@utexas.edu](mailto:deepyaman.datta@utexas.edu)

**Open Access** 

Copyright © 2025 Datta. This is an open-access article distributed under the terms of the [Creative Commons Attribution 4.0 International](#) license, which enables reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator.

cloud-native analytics workflows, such as Looker, rose in popularity, while established players like Tableau evolved (out of necessity) to become more cloud-native. Last but not least, dbt became the de facto standard for SQL-based transformation in the cloud. By 2024, the tenets of the modern data stack were so widely accepted and adopted that Tristan Handy, the individual who initially coined the term, wrote that the idea of distinguishing the “modern” data stack had outlived its usefulness:

When I was a consultant, helping small companies build analytics capabilities, I would only work with MDS tooling. It was so much better that I simply wouldn’t take on a project if the client wanted to use pre-cloud tools. I, and others like me, needed a way to talk about this preference, to differentiate between these new products and all of the other ones that we didn’t want to use. The term actually conveyed important information.

Today, most data products are built for the cloud. Either they have been built in the past ten years and therefore baked in cloud-first assumptions, or they have been re-architected to do so.

— T. Handy [1]

However, when data and analytics engineers talk about the modern data stack, they invariably mean SQL-based workflows. Most *SQL-oriented* data products are built for the cloud. Despite Python’s explosive growth—by some measures, overtaking SQL in popularity among developers [5]—the Python data ecosystem has lagged in supporting data engineering best practices. Python is often relegated to the fallback option for when a use case can’t be solved by SQL, and support can be limited [6] (if available at all).

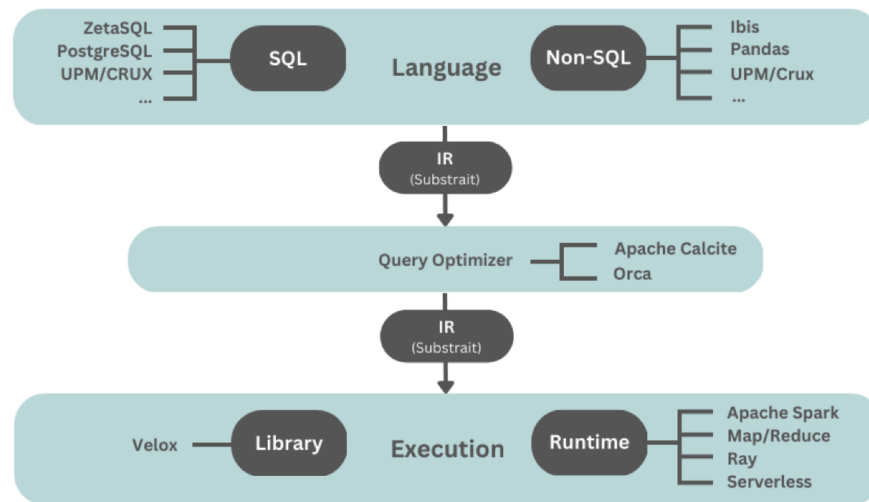
In this paper, we present a set of new software integrations that form the basis for the composable Python analytics stack<sup>1</sup>. First, we will introduce the concept of composable data management systems and Ibis [7], the data processing workhorse of the Python analytics stack. Then, we will present two key components of the emerging stack: Kedro as the core transformation framework and Pandera for data validation. In both cases, we will highlight how Ibis extends the capabilities of the existing, established tool. Finally, we will step back and look at the remaining pieces of the composable analytics stack. We will fill out the picture with Python-native recommendations for ingestion and orchestration. We will also be transparent about some of the current gaps compared to the more established SQL-first approach and ongoing work to address them.

## 2. IBIS AND THE RISE OF COMPOSABLE DATA MANAGEMENT SYSTEMS

In 2021, engineering leaders from Meta, Voltron Data, Databricks, and Sundeck published a joint paper outlining a vision for a modular data stack of reusable components built on open standards [8], as seen in [Figure 2](#). In short, they posited that modern data systems all share a common set of logical components, and developers of modern data management systems should take advantage of these similarities instead of reinventing the wheel and further fragmenting the ecosystem. They argued that a paradigm shift towards composable data systems would not only benefit those building such products but also user experience by improving consistency and reducing the learning curve across libraries.

---

<sup>1</sup>Given Handy suggests “analytics stack” as a more appropriate term for the ecosystem of cloud-first data tools [1], and to avoid confusion with the existing set of technologies closely associated with the phrase “modern data stack,” we prefer to use the name “Python analytics stack” or “Python data stack.”



**Figure 2.** The emerging modular data stack can be distilled into a general model comprised of five layers. A language component generates an intermediate representation (IR) that serves as input to a query optimizer. The optimizer produces query fragments that are passed to an execution engine.

One of the key results of this architecture is that users should be empowered to use the language frontend of their choice with the execution engine that they need. Whereas SQL provides one standardized<sup>2</sup> interface for modern data systems, many use cases demand non-SQL APIs. Pushed by the popularity of Python, coinciding with the increased prevalence of data processing for machine learning and artificial intelligence, more and more data products developed Python dataframe APIs: PySpark and the pandas API on Spark, BigQuery DataFrames, Snowpark and the Snowpark pandas API. However, many database system developers don't have the resources or expertise to build dataframe-like APIs on top of execution engines. Furthermore, P. Pedreira *et al.* [8] point out the merits of the model popularized by Apache Spark [10], where SQL (Spark SQL) and non-SQL (DataFrames, Datasets, PySpark, and the pandas API on Spark) APIs all generate the same underlying intermediate representation (IR), meaning that the following queries should be indistinguishable from an execution standpoint:

```
SELECT a, b, c FROM t WHERE a > 1
```

**Program 1.** Select a subset of rows and columns using SQL.

```
t.filter(t.a > 1).select("a", "b", "c")
```

**Program 2.** Select the same subset of rows and columns using a dataframe API.

This need prompted the creation of open-source libraries like Ibis, the portable Python dataframe API. Ibis provides a lightweight, universal interface for data wrangling that supports 20+ query engines, from local backends like Polars, DuckDB, and DataFusion to remote databases and distributed computation frameworks like BigQuery, Snowflake, and Spark [7]. Ibis constructs a query plan, or IR, that it evaluates lazily (i.e. as and when needed) on the execution engine [11]. Because Ibis code is functionally equivalent to SQL—in fact,

<sup>2</sup>The SQL standard isn't a well-adhered-to standard in that each database supports a slightly different version of the SQL standard, often referred to as a SQL dialect. G. Forsyth [9] shows a few common examples of queries that need to be expressed differently based on the database or execution engine being used.

Ibis produces and executes SQL under the hood for all but the Polars backend—it is well suited for ELT workflows as part of a Python-first data stack.

### 3. BUILDING THE COMPOSABLE, PYTHON-NATIVE DATA STACK

In order to propose a valid Python-first alternative to the SQL-based analytics stack, we break the existing stack into a set of primary functions and address each in turn. As per T. Krantz and A. Jonker [12], the modern data (or analytics) stack consists of several core components including:

- Data storage
- Data ingestion
- Data transformation
- BI and analytics
- Data observability

Some definitions of the analytics stack include specific callouts for reverse ETL, orchestration, and other aspects, but we will focus on the above five layers.

#### 3.1. *Data storage*

The storage layer of the analytics stack provides a centralized, most often cloud-based, medium for managing data. Traditionally, online analytical processing (OLAP) databases like Redshift, BigQuery, and Snowflake were the primary storages associated with the modern data stack. Data lakes that supported managing large data volumes in cloud object storage have also been popular options; distributed query engines like Spark and Trino are well-suited for workloads accessing such file systems. Finally, recent interest in data lakehouse architectures has resulted in increased support for working with data stored in Iceberg, Delta, and other related table formats.

Ibis supports a wide range of file formats, including Parquet files, CSV files, and newline-delimited JSON data. It also allows reading from and writing to Delta Lake tables. If some Ibis execution backend natively implements a more efficient path for working with a particular format, Ibis will leverage it; otherwise, Ibis falls back to a unified solution for handling data in PyArrow record batches. For data warehouses, the storage engine doubles as the execution engine, so Ibis can create, read, update, and delete tables by running the appropriate SQL statements under the hood. Ultimately, data storage layer support comes down to the underlying engine, and Ibis attempts to keep up to date with new developments in its supported backends.

#### 3.2. *Data ingestion*

Data ingestion is the process of extracting data from various sources and loading it into a data storage system (like those discussed in the previous section) for processing and analysis [12]. Because of the activities performed, this process is also often referred to as the extract and load (EL) stage.

dlt is an open-source, production-ready Python framework that has become a popular choice as part of the analytics stack [13]. Since many data engineers are already opting for dlt over established like Fivetran and Airbyte, we don't find it necessary to justify dlt's space in the Python analytics stack. That said, it is worth mentioning that dlt also provides Pythonic reverse ETL [14] (one of the categories that we chose not to cover explicitly, given the broader industry trend favoring general data movement tools over dedicated reverse ETL frameworks). dlt also supports basic transformation capabilities, including built-in Ibis support. However, for all but the simplest use cases, we recommend a dedicated transfor-

mation framework, much in the way dlt is frequently used alongside dbt in the existing analytics stack.

```
from pathlib import Path

import dlt
from dlt.sources.filesystem import filesystem, read_csv_duckdb

# Create a list of dlt resources corresponding to each of the raw files.
readers = []
for name in ["raw_customers", "raw_orders", "raw_payments"]:
    files = filesystem(
        bucket_url=(Path(__file__).parent / "data").as_uri(),
        file_glob=f"01_raw/{name}.csv",
    )
    reader = (files | read_csv_duckdb()).with_name(name)
    readers.append(reader)

# Create a new dlt pipeline configured to use DuckDB as the destination.
pipeline = dlt.pipeline(
    pipeline_name="jaffle_shop", dataset_name="main", destination="duckdb"
)

# Run the pipeline to load data into DuckDB, and output run information.
info = pipeline.run(readers)
print(info)
```

**Program 3.** A dlt pipeline to load data from local CSV files into a DuckDB database.

### 3.3. Data transformation

After data lands in the centralized data storage, it needs to be cleaned, processed, and combined before it's useful for analytics, reporting, or machine learning. Before dbt became the standard for SQL data transformation, the SQL data transformation layer often consisted of an unstructured mess of scripts, stored procedures, or—even worse—GUI-based data pipelines. dbt changed the game by providing a “SQL-first transformation workflow that lets teams quickly and collaboratively deploy analytics code following software engineering best practices” [15].

Kedro is an open-source Python framework that empowers data teams to write reproducible, maintainable, and modular data pipelines [16]. Kedro joined the Linux Foundation AI & Data Foundation in 2021, graduating from Incubation in 2024 [17]. Like dbt, Kedro helps apply software engineering best practices to data transformation code. Both projects also model transformation workflows as directed acyclic graphs (DAGs). A former dbt Labs product manager remarked on the similarities between the two frameworks:

When I learned about Kedro (while at dbt Labs), I commented that it was like dbt if it were created by Python data scientists instead of SQL data analysts (including both being created out of consulting companies).

— Cody Peterson, Technical Product Manager @ Voltron Data [11]

Our work integrating Kedro and Ibis opens the door to Python transformation workflows on par with SQL-based transformation with tools like dbt. To illustrate, we reimplement the canonical Jaffle Shop dbt demo project [18] using this stack.

#### 3.3.1. Configuring the Data Catalog

One of the core features of Kedro is the Data Catalog, a per-Kedro-project repository of data sources and sinks configured for use by the project [16]. Kedro also supports a concept of data connectors called datasets. To enable reading and writing Ibis tables in Kedro pipelines, we contributed Ibis dataset implementations for handling data in tables and in

files to Kedro-Datasets, a Python package containing commonly-used dataset implementations maintained by the Kedro team.

The standard approach for using a dataset is to add it to the `catalog.yml` configuration file:

```
_duckdb:
  backend: duckdb
  # `database` and `threads` are parameters for `ibis.duckdb.connect()`.
  database: jaffle_shop.duckdb
  threads: 24

seed_customers:
  type: ibis.FileDataset
  filepath: data/01_raw/raw_customers.csv
  file_format: csv
  connection: ${_duckdb}

raw_customers:
  type: ibis.TableDataset
  table_name: raw_customers
  connection: ${_duckdb}
  save_args:
    materialized: table

...

```

**Program 4.** A subset of the dataset definitions used to implement a port of the Jaffle Shop dbt demo project [18] using Kedro and Ibis.

With the Kedro-Ibis integration, data is represented in memory as Ibis tables, analogous to other lazy dataframe implementations like Polars LazyFrames and Spark DataFrames. The main difference is that Ibis can work with a much wider array of backend data systems; for example, the above example connects to a DuckDB database. The implementations of the Ibis datasets also abstract other functionalities that data engineers require, including support for different materialization strategies (seed\_customers stores loaded data in a view, whereas raw\_customers overrides the default behavior and persists data as a table) [11].

Other optimizations include caching connections for reuse. Similar to how dbt configures connection details in `profiles.yml` [19], we leverage variable interpolation supported by the Kedro configuration loader to define and reuse connection information [11].

### 3.3.2. Building pipelines

Kedro pipelines are composed of nodes, which themselves are simply Python functions. Given the datasets described in the above section load and save data as Ibis tables, the function needs to take as input and return as output any number of Ibis tables.

```

from __future__ import annotations

import ibis

def process_orders(
    orders: ibis.Table, payments: ibis.Table, payment_methods: list[str]
) -> ibis.Table:
    total_amount_by_payment_method = {}
    for payment_method in payment_methods:
        total_amount_by_payment_method[f"{payment_method}_amount"] = ibis.coalesce(
            payments.amount.sum(where=payments.payment_method == payment_method), 0
        )

    order_payments = payments.group_by("order_id").aggregate(
        **total_amount_by_payment_method, total_amount=payments.amount.sum()
    )

    final = orders.left_join(order_payments, "order_id")[
        [
            orders.order_id,
            orders.customer_id,
            orders.order_date,
            orders.status,
            *[
                order_payments[f"{payment_method}_amount"]
                for payment_method in payment_methods
            ],
            order_payments.total_amount.name("amount"),
        ]
    ]
    return final

```

**Program 5.** The same logic as the `orders.sql` model from dbt Labs, Inc. [18] expressed using Ibis in a Python function.

One of the advantages of native Python is cleaner parametrization. In the above example, we iterate over `payment_methods` using a `for` loop. By comparison, a templating language like Jinja is necessary to extend the capabilities of SQL to support control structures, variable interpolation, and more in dbt [20]. Another benefit of Python as a data transformation language is that it supports unit tests. While dbt added support for unit tests in 2024, it has many limitations [21].

A complete port of the Jaffle Shop project using Kedro and Ibis is available on GitHub [22].

### 3.4. BI and analytics

The BI and analytics layer includes everything from dashboards (with tools like PowerBI, Tableau, Looker, and Streamlit) to machine learning and data science [12]. While SQL is well supported choice by most BI tools, Python holds a significant edge in AI and ML workflows. In fact, we believe that data scientists, machine learning engineers, and data engineers who work alongside data science teams are likely to be the biggest beneficiaries of the emergence of the Python-first analytics stack. Kedro is already established as a popular framework for authoring data science pipelines, and these developments help enable cross-functional teams to write end-to-end data transformation pipelines using a single technology.

That said, Ibis does integrate with a number of data visualization and dashboarding tools, including Streamlit and Plotly [7]. Furthermore, most BI tools support Python to some extent, and Ibis can produce data in a supported dataframe format.

### 3.5. Data observability

Data observability is a broad category that includes data quality, freshness, and other metrics [12]. In our work, we focused on providing a solution for data validation as part of

the composable Python analytics stack. Pandera originated as a lightweight-yet-expressive API for validating pandas dataframes [23]. As it evolved, Pandera added support for other pandas-compatible APIs, including GeoPandas, Dask, Modin, and the pandas API on Spark. Later, Pandera loosened its tight coupling with the pandas API in order to support other execution engines like Polars [24].

We take advantage of this prior work to support validating Ibis tables using Pandera—and, by extension, enable data validation across the full suite of Ibis-supported backends. For built-in checks (i.e. the wide range of common data quality checks that Pandera supports out of the box), the fact that *Ibis* enables validation on a previously-unsupported data processing framework or database can be almost transparent to the user. For more bespoke checks, the Ibis backend (similar to all of the previously-existing Pandera backends) supports defining custom checks, and these do need to be written using Ibis syntax.

```
import ibis
import pandera.ibis as pa
from ibis import _
from pandera.ibis import IbisData

def total_amount_positive(data: IbisData) -> ibis.Table:
    w = ibis.window(group_by="order_id")
    with_total_amount = data.table.mutate(total_amount=data.table.amount.sum().over(w))
    return with_total_amount.order_by("order_id").select(_.total_amount >= 0)

schema = pa.DataFrameSchema(
    columns={
        "order_id": pa.Column(int),
        "amount": pa.Column(float),
        "status": pa.Column(
            str,
            pa.Check.isin(
                ["placed", "shipped", "completed", "returned", "return_pending"]
            ),
        ),
    },
    checks=[pa.Check(total_amount_positive)],
)

con = ibis.duckdb.connect("jaffle_shop.duckdb")
orders = con.table("orders")

schema.validate(orders)
```

**Program 6.** An example of validating data in DuckDB using the Ibis backend for Pandera, including both built-in and custom checks.

## 4. THE FUTURE OF THE COMPOSABLE, PYTHON-NATIVE DATA STACK

In this paper, we have laid out the foundations for the composable, Python-native data stack. As with any new development in the open-source data ecosystem, getting to this point would not have been possible without the contributions of many people and projects over the past decade—maintainers, committers, contributors, and users across dlt, Kedro, Pandera, and, of course, Ibis.

At the same time, the Python data stack is not (yet) a product category that the data engineering community has coalesced around in the way that the modern data stack went from an idea to the accepted way to do data engineering using SQL and the cloud years prior. This means that there are gaps waiting to be addressed and hardening that needs to happen. dlt has quickly gained traction among data engineers. Kedro is a mature data transformation framework that benefits from an active community and usage across academia, research institutions, startups, and Fortune 500 companies. The Ibis integration has



been widely adopted by users, despite being relatively new; this usage has also highlighted functionalities that need adding, such as support for insert and upsert operations, both of which are in progress. Pandera itself is a very popular choice for data validation, but the Ibis integration is new. Other integrations, usually driven by the community (between Kedro and Pandera, as well as between Kedro and workflow orchestrators like Airflow and Dagster), show promise as part of a future iteration of the stack.

The future is bright for the Python analytics stack. For one, the pieces fell together not because we wanted to create a Python-based alternative to the already-successful SQL data stack, but rather because users were performing data engineering tasks using Python and didn't have the tooling to do it the "right" way. Furthermore, the growth in popularity of Python means that more users will face the same challenges we already do. While not a focus of this paper, some of the frameworks presented also have technical advantages that come from the composable nature of the proposed stack. Because Ibis provides a unified API, data transformation and data validation using Ibis is portable, meaning you can develop using local engines and run in production using other engines. By extension, the Python-first analytics stack is well-positioned with increasing interest in multi-engine stacks.

## REFERENCES

- [1] T. Handy, "Is the "Modern Data Stack" Still a Useful Idea?." [Online]. Available: <https://roundup.getdbt.com/p/is-the-modern-data-stack-still-a>
- [2] T. Handy, "What are the steps / tools in setting up a modern, SaaS-based BI infrastructure?." [Online]. Available: <https://www.linkedin.com/pulse/what-steps-tools-setting-up-modern-saas-based-bi-tristan-handy/>
- [3] C. Wang, "ETL vs. ELT: Choose the right approach for data integration." [Online]. Available: <https://www.fivetran.com/blog/etl-vs-elt>
- [4] I. a. i. a. Amazon Web Services, "Data Warehousing on AWS: AWS Whitepaper," 2021. [Online]. Available: <https://docs.aws.amazon.com/pdfs/whitepapers/latest/data-warehousing-on-aws/data-warehousing-on-aws.pdf>
- [5] Stack Exchange, Inc., "Stack Overflow Developer Survey 2023." [Online]. Available: <https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages>
- [6] dbt Labs, Inc., "Python models." [Online]. Available: <https://docs.getdbt.com/docs/build/python-models>
- [7] Ibis maintainers, "Ibis: the portable Python dataframe library." [Online]. Available: <https://ibis-project.org/>
- [8] P. Pedreira *et al.*, "The Composable Data Management System Manifesto," *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2679–2685, 2023, doi: 10.14778/3603581.3603604.
- [9] G. Forsyth, "Designing Interfaces is hard." [Online]. Available: <https://ibis-project.org/presentations/datafusion-meetup-nyc-2024/talk#/sql-questions-with-no-single-answer>
- [10] M. Zaharia *et al.*, "Apache Spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016, doi: 10.1145/2934664.
- [11] D. Datta, "Building scalable data pipelines with Kedro and Ibis." [Online]. Available: <https://kedro.org/blog/building-scalable-data-pipelines-with-kedro-and-ibis>
- [12] T. Krantz and A. Jonker, "What is the modern data stack?." [Online]. Available: <https://www.ibm.com/think/topics/modern-data-stack>
- [13] dlthub, "dlthub: ELT as Python Code." [Online]. Available: <https://dlthub.com/>
- [14] A. Brudaru, "dlt adds Reverse ETL - build a custom destination in minutes." [Online]. Available: <https://dlthub.com/blog/reverse-etl-dlt>
- [15] dbt Labs, Inc., "What is dbt?." [Online]. Available: <https://www.getdbt.com/product/what-is-dbt>
- [16] S. Alam *et al.*, "Kedro." [Online]. Available: <https://github.com/kedro-org/kedro>
- [17] T. L. Foundation, "Kedro." [Online]. Available: <https://lfaidata.foundation/projects/kedro/>
- [18] dbt Labs, Inc., "The Jaffle Shop." [Online]. Available: [https://github.com/dbt-labs/jaffle\\_shop\\_duckdb](https://github.com/dbt-labs/jaffle_shop_duckdb)
- [19] dbt Labs, Inc., "About profiles.yml." [Online]. Available: <https://docs.getdbt.com/docs/core/connect-data-platform/profiles.yml>
- [20] dbt Labs, Inc., "Jinja and macros." [Online]. Available: <https://docs.getdbt.com/docs/build/jinja-macros>
- [21] dbt Labs, Inc., "Unit tests." [Online]. Available: <https://docs.getdbt.com/docs/build/unit-tests>

- [22] D. Datta, “Jaffle Shop.” [Online]. Available: <https://github.com/deepyaman/jaffle-shop>
- [23] N. Bantilan, “pandera: Statistical Data Validation of Pandas Dataframes,” in *Proceedings of the 19th Python in Science Conference*, in SciPy. 2020, pp. 116–124. doi: [10.25080/majora-342d178e-010](https://doi.org/10.25080/majora-342d178e-010).
- [24] N. Bantilan, “Pandera: Going Beyond Pandas Data Validation,” in *Proceedings of the 22nd Python in Science Conference*, in SciPy. 2023, pp. 124–129. doi: [10.25080/gerudo-f2bc6f59-010](https://doi.org/10.25080/gerudo-f2bc6f59-010).