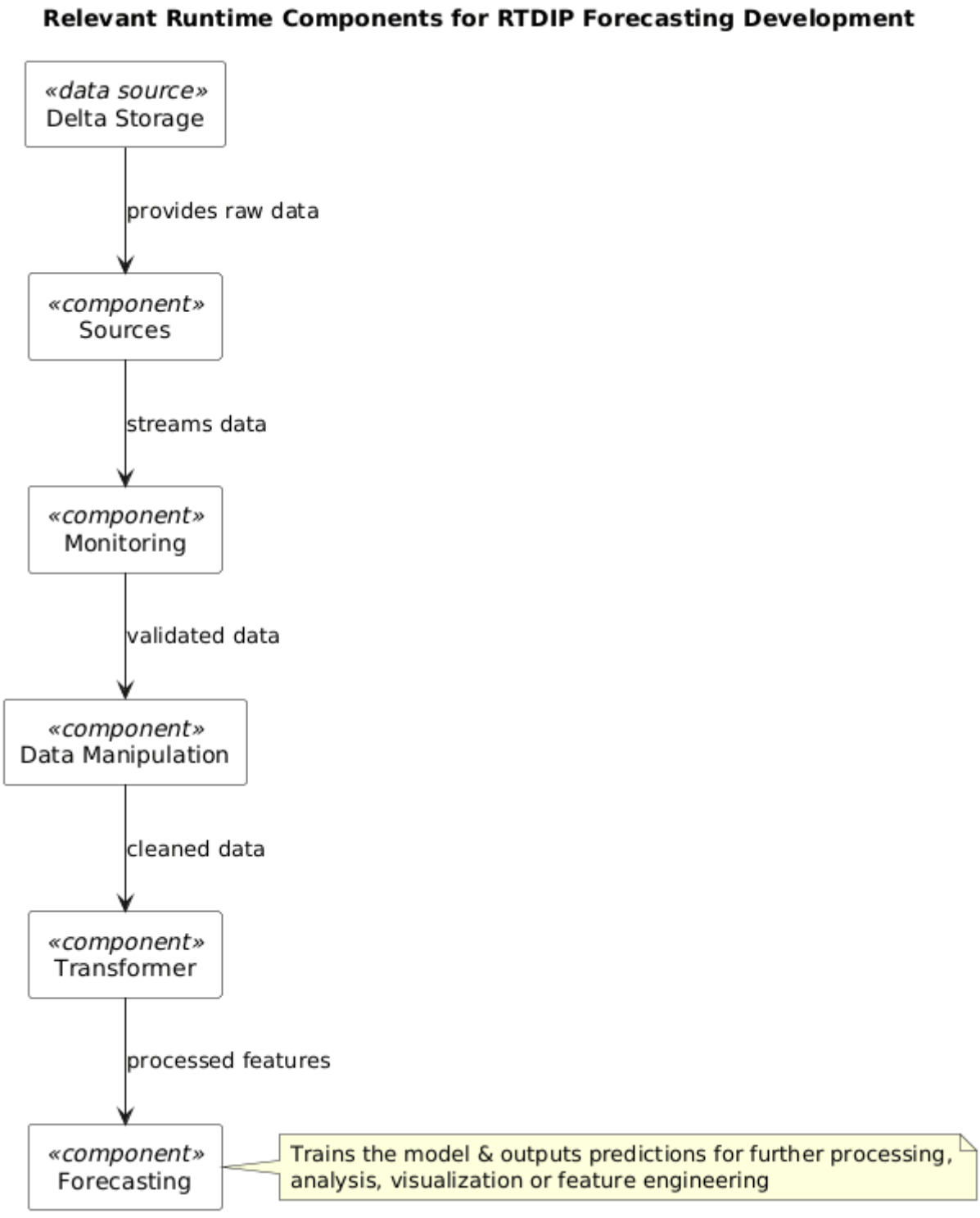
Runtime Components



Code Components

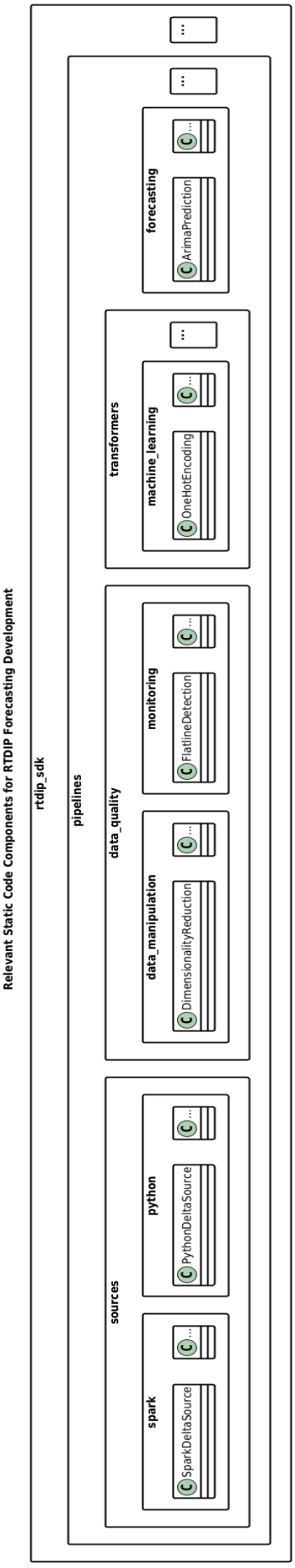


Diagram Explanation

As RTDIP is an existing project that has been developed over multiple years, its code base has grown too large to clearly depict on one page, especially on the class level. Instead, we decided to show only those components that are relevant for our project scope, the (local) development of forecasting methods. For now, this includes data storage and streaming, pre-processing, as well as model training, testing and evaluation. We chose two complementary diagram styles to clearly distinguish between runtime behavior and static code structure, using UML-inspired notation adapted for practical documentation rather than strict standard compliance.

1. Runtime Components

The runtime diagram illustrates the dynamic behavior of the system during execution. It shows the flow of data through various processing stages, from retrieving data from Delta Storage, through quality monitoring and preprocessing, to generating forecasts. This diagram is intended to demonstrate how components interact at runtime and the sequence of operations during a typical forecasting workflow.

2. Code Components

The code diagram depicts the static structure of the RTDIP SDK codebase. It shows the hierarchical organization of packages, along with example classes from relevant packages for the development of forecasting models. The "..." notation indicates additional components (packages, modules and classes) exist but are omitted for clarity. Similarly, the tests, documentation, API server code and web code are also excluded. This diagram is intended to show how the code is organized, and which modules contain the functionality used in our forecasting development workflow.

Technology Stack

The Technology Stack also focuses on the forecasting model development scope. The repo uses many more libraries that are not mentioned here.

The RTDIP forecasting development stack is built on PySpark (3.3.0-3.5.x) as the core distributed processing engine, enabling scalability from local development to production clusters.

The pipeline begins with Delta Storage using Delta Lake (2.2.0-3.2.x) for ACID-compliant data storage, which stores data in Apache Parquet format internally. CSV files may be used as initial input format for rapid prototyping.

The Sources component streams data using delta-spark as the connector, with pandas (<3.0.0) handling local file operations. Data quality is ensured through Great Expectations (0.18.8+) for validation rules and PySpark SQL for query-based checks, supported by NumPy (<2.0) for statistical computations in the Monitoring component. Data Manipulation leverages PySpark's distributed transformation capabilities with NumPy for numerical operations and pandas for preprocessing.

The Transformer component uses PySpark MLlib with built-in feature engineering tools (e.g., OneHotEncoder). The Forecasting component currently employs two complementary approaches: statsmodels (0.14.1+) provides industry-standard ARIMA time series forecasting with interpretable statistical models, while pmdarima (2.0.4+) automates ARIMA parameter selection. For distributed machine learning, PySpark MLlib currently implements Linear Regression and other algorithms. The stack uses pandas as a bridge between PySpark and statsmodels for ARIMA models.

Key technology decisions prioritize interoperability (pandas/NumPy bridge PySpark and statsmodels), scalability (PySpark enables seamless local-to-cluster deployment), interpretability (statistical models over black-box deep learning), and CPU efficiency (no GPU requirements). The entire stack is implemented in Python and supports Python 3.9-3.12. Testing uses pytest (7.4.0) throughout the codebase.