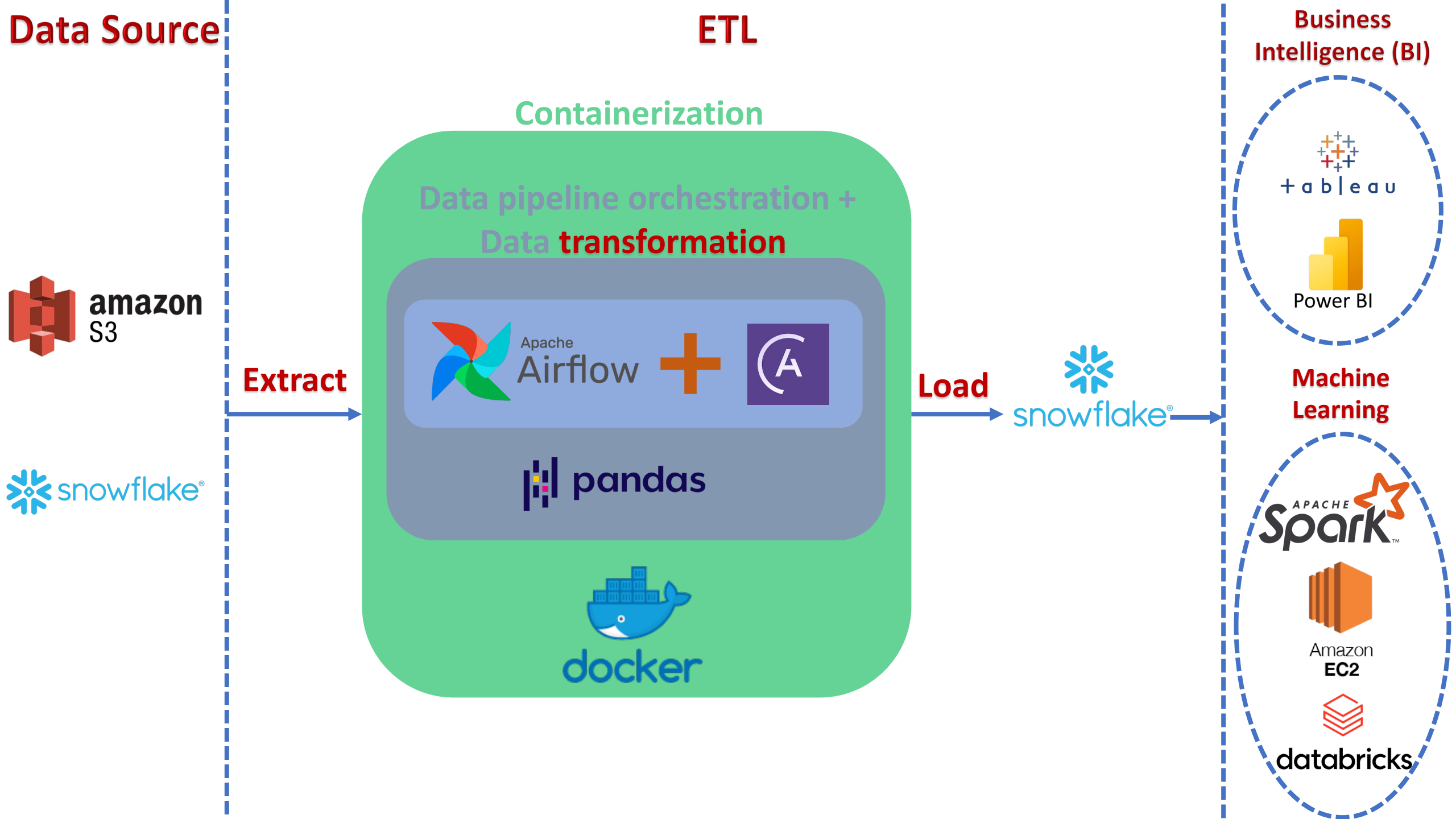
Data Source
ETL
Business Intelligence (BI)
Containerization
Data pipeline orchestration +
Data transformation
amazon S3
snowflake
Extract
Apache Airflow
pandas
docker
Load
snowflake
tableau
Power BI
Machine Learning
APACHE Spark
Amazon EC2
databricks

Load

```
AWS_CONN_ID = "aws_default"
SNOWFLAKE_CONN_ID = "snowflake_default"
SNOWFLAKE_ORDERS = "orders_table"
SNOWFLAKE_FILTERED_ORDERS = "filtered_table"
SNOWFLAKE_JOINED = "joined_table"
SNOWFLAKE_CUSTOMERS = "customers_table"
SNOWFLAKE_REPORTING = "reporting_table"
```

Airflow provider connections.

```
with DAG(dag_id='purchase_orders_pipeline', start_date=datetime(2023, 1, 1),
         schedule_interval='@daily', catchup=False) as dag:

    ## OPERATOR_1:
    # Allows to transfer data from a file into a table (load the files). Here we load the data from an S3 bucket:
    order_data = aql.load_file(
        input_file = File(
            path = "s3://airflow-astro-sdk-project/orders_data_header.csv",
            conn_id = AWS_CONN_ID
        ),
        # Creates a temporary table on Snowflake with the loaded data.
        # The conn_id name is not specified in order to create a temporary table,
        # because the purpose is to store the data but not store it after the DAG is completed.
        output_table = Table(conn_id = SNOWFLAKE_CONN_ID)
    )

    ## OPERATOR_2:
    # Allows to interact with the customers table in Snowflake from the data pipeline.
    customers_table = Table(
        name = SNOWFLAKE_CUSTOMERS,
        conn_id = SNOWFLAKE_CONN_ID
    )
```

Loading a file from an S3 bucket and connecting to an existing table on Snowflake.

Transform (operators)

```python
## OPERATOR_3:
# Bellow the "join_orders_customers" function dependencies is defined.
# The first table on the join is the "order_data" from the S3 bucket but transform by
# the function "filter_orders", then the second table is directly the "customers_table" in Snowflake.
joined_data = join_orders_customers(filter_orders(order_data), customers_table)

## OPERATOR_4:
# This operator allows to add data to an existing table (from "customers_table" added to "reporting_table")
# with "ignore and update" conflict resolution technique.
reporting_table = aql.merge(
    target_table=Table(
        name = SNOWFLAKE_REPORTING,
        conn_id = SNOWFLAKE_CONN_ID
        ),
    source_table = joined_data,
    # Merge on the "order_id" column.
    target_conflict_columns = ["order_id"],
    # If there is a conflict on "customer_id" or "customer_name"(...)
    columns = ["customer_id", "customer_name"],
    # (...)use the data from the "joined_data" table.
    if_conflicts = "update",
)

## OPERATOR_5:
# Creates a Dataframe from the "reporting_table" on Snowflake.
purchase_dates = transform_dataframe(df=reporting_table, output_table=Table(name="pandas_transformed_reporting_table"))

## OPERATOR_6:
reporting_table = create_reporting_table(conn_id=SNOWFLAKE_CONN_ID)

## OPERATOR_7:
# Append transformed data to the reporting table.
# Dependency is inferred by passing the previous `transformed_data` task to `source_table` param.
# Here the freshly created Snowflake table "reporting_table_essential" is fulfilled with the Pandas
# trasnformed data from "pandas_transformed_reporting_table".
record_results = aql.append(
    source_table=purchase_dates,
    target_table=Table(name="reporting_table_essential", conn_id=SNOWFLAKE_CONN_ID),
    columns=["customer_name", "purchase_date", "avg_purchase_per_year", "total_purchase_amount"],
)
```

Uses the “`join_orders_customers`” to join the S3 “orders” table (that is firstly filtered using “`filter_orders`”) and Snowflake “customers” table. (SQL transformation)

Uses “aql” function from Astro to merge both Snowflake tables “`REPORTING`” and a query table stored on “`joined_data`” and solving conflict issues between tables.

Uses “`transform_dataframe`” function for data transformations and outputs back to Snowflake a table called “`pandas_transformed_reporting_table`”

Activates the python function “`create_reporting_table`” to create a table in Snowflake.

Uses “aql” function “append” to insert data to a final “`reporting_table_essential`” table in Snowflake.

Transform (functions)

```python
## Functions responsible for transforming the data:
# The transform decorator beblow allows the transformation of the data, so implements the "T" of the ETL process
# by running an SQL query. Each time the pipeline does a transformation that creates a new table from the
# SELECT statement that will be passed easily to the next task.
@aql.transform
def filter_orders(input_table: Table):
    return """SELECT * FROM {{input_table}} WHERE amount > 150"""


# Join beetween tthe table from "filter_orders" and the "customers_table" in Snowflake.
@aql.transform
def join_orders_customers(filtered_orders_table: Table, customers_table: Table):
    return """SELECT c.customer_id, customer_name, order_id, purchase_date, amount, type
    FROM {{filtered_orders_table}} f JOIN {{customers_table}} c
    ON f.customer_id = c.customer_id"""

# Transform the data from a "reporting SQL table" to a DataFrame.
# The bellow decorator allows to transformation SQL table into Dataframe, so it can be manipulated in a
# Python environment with Pandas.
@aql.dataframe
def transform_dataframe(df: pd.DataFrame):
    df.columns = df.columns.str.lower()  # Convert column names to lowercase.
    # Calculate the total purchase amount for each customer.
    total_purchase_amount = df.groupby('customer_id')['amount'].sum().reset_index()
    total_purchase_amount.rename(columns={'amount': 'total_purchase_amount'}, inplace=True)
    # Convert 'purchase_date' to datetime to allow the next code line.
    df['date'] = pd.to_datetime(df['purchase_date'], format='%Y-%m-%d %H:%M:%S')
    # Create a new column 'year' based on 'purchase_date'.
    df['year'] = df['date'].dt.year
    # Calculate the average purchase amount per year.
    avg_purchase_per_year = df.groupby(['customer_id', 'year'])['amount'].mean().reset_index()
    avg_purchase_per_year.rename(columns={'amount': 'avg_purchase_per_year'}, inplace=True)
    # Filter for customers with a total purchase amount greater than 300.
    high_value_customers = total_purchase_amount[total_purchase_amount['total_purchase_amount'] > 300]
    # Combine all transformations and return the final DataFrame.
    final_df = df.merge(avg_purchase_per_year, on=['customer_id', 'year'], how='left')
    final_df = final_df.merge(high_value_customers, on='customer_id', how='left')
    print(final_df.head(2))

    return final_df
```

"`transform_dataframe`" is a function with the purpose of transforming SQL quarriable data to a dataframe data capable of being transformed by Pandas and performing some transformations with it, returning at the end a final dataframe.

Transform (functions)

```
# Run a raw SQL statement to create the reporting table if it doesn't already exist.
@aql.run_raw_sql
def create_reporting_table():
    """Create the reporting data which will be the target of the append method"""
    return """
    CREATE OR REPLACE TABLE reporting_table_essential (
      customer_name VARCHAR,
      purchase_date DATE,
      avg_purchase_per_year NUMBER,
      total_purchase_amount NUMBER
    );
    """
```

Creates the final table "`reporting_table_essential`" schema in Snowflake.

Final

```
## OPERATOR_8:
# Dependency is inferred by passing the 'reporting_table' only before 'record_results'
# and running in parallel with 'purchase_dates'
reporting_table >> record_results
# Clean up the temporary tables, mainly the ones on "operation_1".
purchase_dates >> record_results >> aql.cleanup()
```

After this setp, the data can be used/load for:

- BI using tools lke Power BI or Tableau:
- ML using tools like Apache Spark, AWS EC2 or Databricks