Pydiverse Pipedag

A library for data pipeline orchestration optimizing high development iteration speed

https://pydiversepipedag.readthedocs.io/en/latest/

Who’s talking?

Martin Trautmann (@windiana42 on GitHub)

- IOI 2000 / Jugend forscht (https://t.ly/DmvHG)
- Electrical Engineering at KIT, Stanford University, and IMEC/KU-Leuven
  - Research: Mapping software descriptions on hardware targets + Transactional Memory
- QuantCo – Optimizing high stakes business decisions by data analytics

QuantCo uses and contributes to open source

https://github.com/QuantCo

- conda-forge / pixi
- apache arrow / parquet / duckdb
- polars / polarify
- ONNX / spox / ndonnx / plonnx
- datajudge / sqlcompyr / tabular delta
- tabmat / glum / metalearners
- pydiverse

Agenda

- Introduction
- Iterating quickly between:
  - Showing example code of pipedag
  - Motivating important aspects of pipeline orchestration
- Summarizing how to achieve high iteration speed

What is pipeline orchestration?

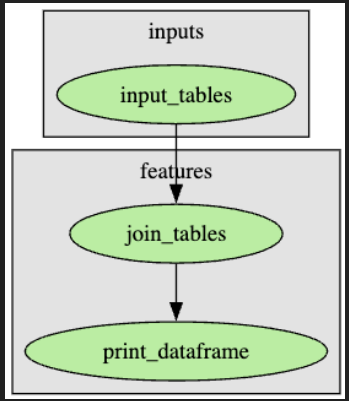
- ***Data pipeline:*** a series of potentially expensive steps often built around training some machine learning model
- ***Orchestration:*** the process of executing steps of a pipeline
- The term ***workflow engine*** is sometimes used interchangeably
- ***Data pipelines*** often focus on transporting/transforming ***tables*** (table = rows conforming to typed columns)
- A dataframe would also classify as a table

Example Data Pipeline:

Pipedag and its competitive landscape

- How to get started:
  - `conda install pydiverse-pipedag` # also works with pip
  - https://pydiversepipedag.readthedocs.io/en/latest/quickstart.html
- Features of pipedag:
  - Wraps many transformation coding styles with minimum boilerplate
  - Automatic cache invalidation
  - Supports high iteration speed (for insight generation loop)
- Competitive landscape:
  - airflow
  - prefect
  - dagster
  - dbt
  - hamilton
  - …

Hello World Pipedag




```python
import pandas as pd
import sqlalchemy as sa

from pydiverse.pipedag import Flow, Stage,
materialize

def main():
    # Define how the different tasks should be wired
    with Flow("flow") as flow:
        with Stage("inputs"):
            names, ages = input_tables()

        with Stage("features"):
            joined_table = join_tables(names, ages)
            print_dataframe(joined_table)

    flow.run()
```

```python
@materialize(version="1.0", nout=2)
def input_tables():
    names = pd.DataFrame({
        "id": [1, 2, 3],
        "name": ["Alice", "Bob", "Charlie"],
    })
    ages = pd.DataFrame({
        "id": [1, 2, 3],
        "age": [20, 40, 60],
    })
    return names, ages

@materialize(lazy=True, input_type=sa.Table)
def join_tables(names: sa.Alias, ages: sa.Alias):
    return (
        sa.select(names.c.id, names.c.name, ages.c.age)
        .join_from(names, ages, names.c.id == ages.c.id)
    )

@materialize(input_type=pd.DataFrame)
def print_dataframe(df: pd.DataFrame):
    print(df)
```

Materialization / Dematerialization

The task decides the library *X* in which it wants to describe a data transformation:

```
@materialize(input_type=X)
def task(in_tbl: X):
    out_tbl = f(in_tbl)
    return out_tbl
```

This effectively does the following:

- Dematerialize input *in_tbl*
  - *Dematerialize = Loading from Table Store*
  - *Table Store is often a SQL Database*
- Run:
  `out_tbl = f(in_tbl)`
- Materialize output *out_tbl* as table
  - *Materialize = Writing to Table Store*

You can choose from many transformation languages

This is important to be able to wrap existing code in the form it already exists

https://github.com/Quantco/vectorization-tutorial/blob/main/vectorization06.ipynb

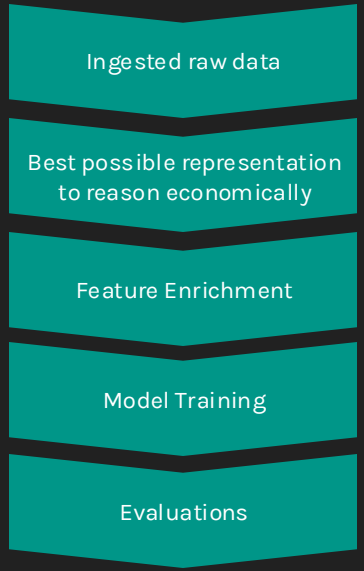
```python
@materialize(input_type=pd.DataFrame, version="1.0.0")
def task_pandas(a: pd.DataFrame, b: pd.DataFrame):
    return a.merge(b, on="pk", how="left").assign(x2=lambda df: df.x * df.x)

@materialize(input_type=pl.LazyFrame, version=AUTO_VERSION)
def task_polars(a: pl.LazyFrame, b: pl.LazyFrame):
    x = pl.col("x")
    return a.join(b, on="pk", how="left").with_columns(x2=x * x)

@materialize(input_type=pdt.SQLTableImpl, lazy=True)
def task_transform_sql(a: pdt.Table, b: pdt.Table):
    return a >> left_join(b, pk_match(a, b)) >> mutate(x2=b.x * b.x)

@materialize(input_type=ibis.api.Table, lazy=True)
def task_ibis(a: ibis.api.Table, b: ibis.api.Table):
    return a.left_join(b, pk_match(a, b)).mutate(x2=b.x * b.x)
```

```python
@materialize(input_type=sa.Table, lazy=True)
def task_sqlalchemy(a: sa.Alias, b: sa.Alias):
    return sa.select(
        *a.c, *b.c,
        (b.c.x * b.c.x).label("x2"),
    ).select_from(a.outerjoin(b, pk_match(a, b)))
```

```python
@materialize(input_type=sa.Table, lazy=True)
def task_sql(a: sa.Alias, b: sa.Alias):
    return sa.text(f"""
        SELECT
            a.*, b.*, b.x * b.x AS x2
        FROM {ref(a)} AS a
        LEFT JOIN {ref(b)} AS b
        ON a.pk = b.pk
    """)
```

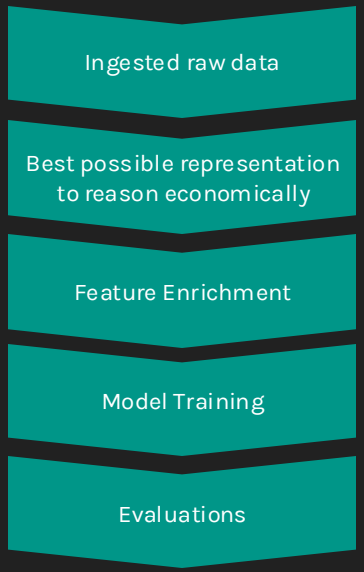
Data Pipeline in Code and Prose

```
with Flow() as flow:
    with Stage("raw_input"):
        raw_a, raw_b = read_input_data()

    with Stage("economic_representation"):
        econ = economic_representation(raw_a, raw_b)

    with Stage("features"):
        features = compute_features(econ)

    with Stage("model_training"):
        input_data_train, target_train, encoding_parameters = model_encoding(
            econ, features, train=True
        )
        model, output_train = model_train(input_data_train, target_train)

    with Stage("model_evaluation"):
        input_data_test, target_test, _ = model_encoding(
            econ, features, encoding_parameters=encoding_parameters
        )
        output_test = model_predict(model, input_data_test)
        evaluation_result = evaluate_model(output_test, target_test)
```



Parallelization under constant Change

Many parallelization techniques limit changes that can be made to algorithms – two stand out
[Most data pipelines are not dealing with BIG Data]



SQL

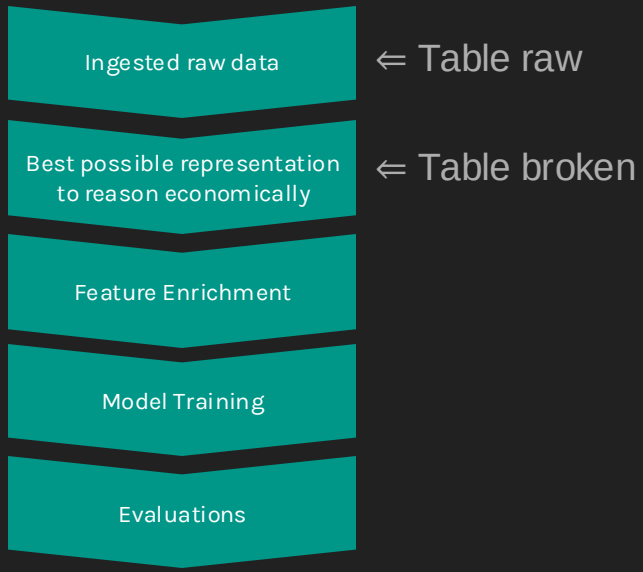
```
create table as select …
```

SQL can describe a transformation that is executed fully within the database and can be parallelized there

Full dataset in memory of high-RAM-machine (Pandas / Polars)

Pandas/polars are dateframe libraries that are often used on >128GB RAM machines to train + evaluate models

Explorative SQL: Tracing down problems really fast



⇐ Table raw

⇐ Table broken

- In/Out diagnostic query:

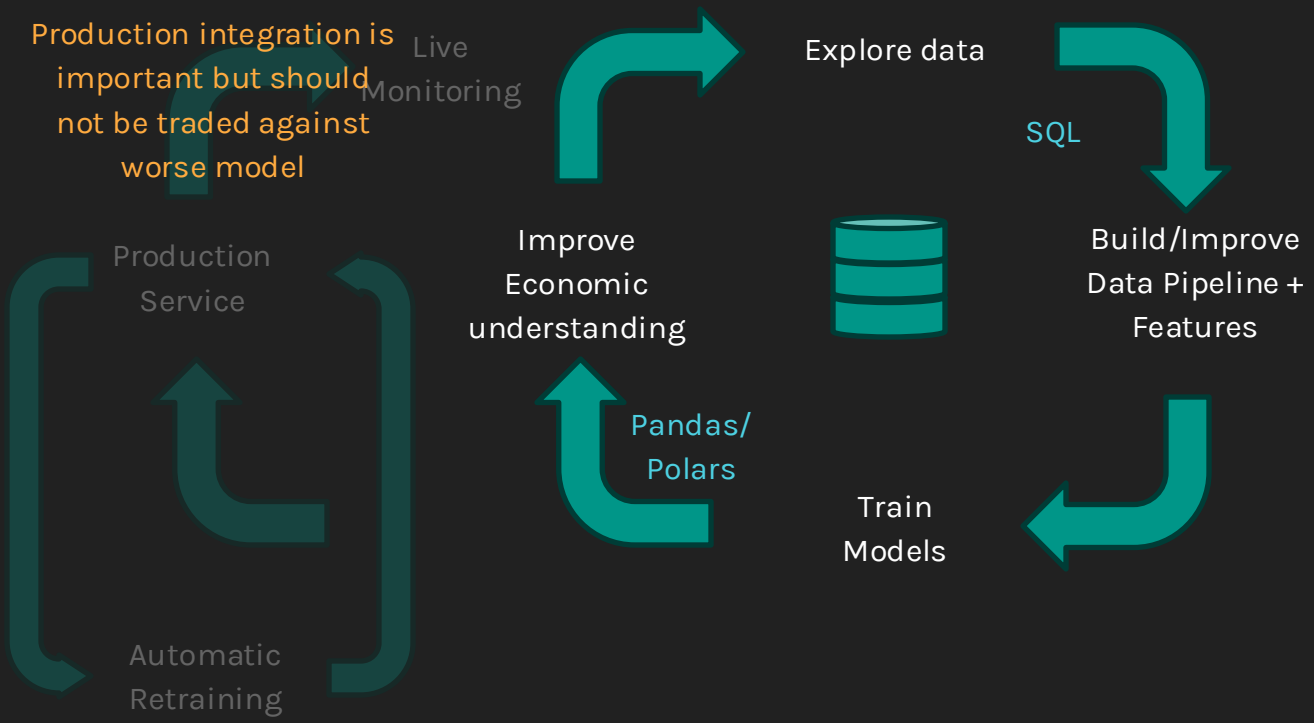
```
SELECT * FROM raw
LEFT JOIN broken on raw.id=broken.id
WHERE broken.col not in ('a', 'b', 'c')
-- LIMIT 10
```

- Transformation diagnostic query

```
-- CREATE TABLE broken AS
SELECT TRIM(raw.x) as col
      ,raw2.y      as irrelevant
LEFT JOIN raw2
on raw.id=raw2.id
WHERE raw.z > 0

-- Debugging:
AND TRIM(raw.x) not in ('a', 'b', 'c')
```

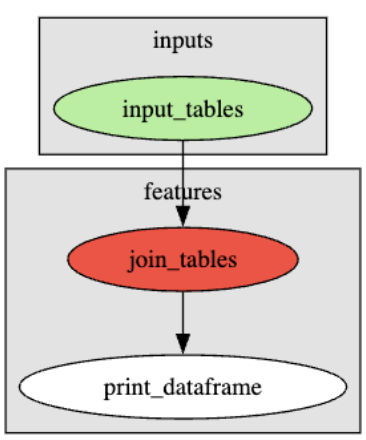
High development iteration speed

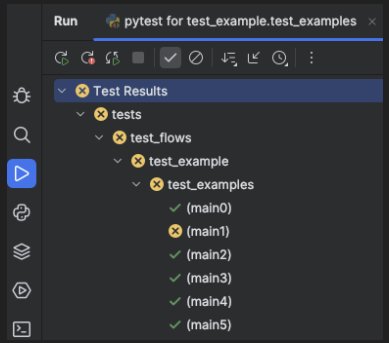
The faster the insight generation loop, the better the models you may build

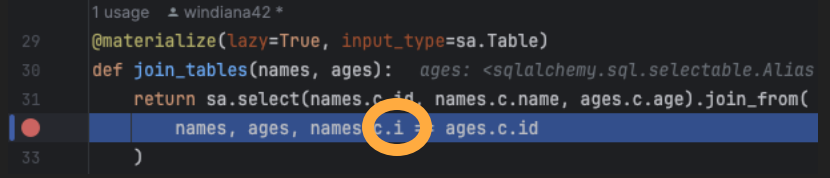


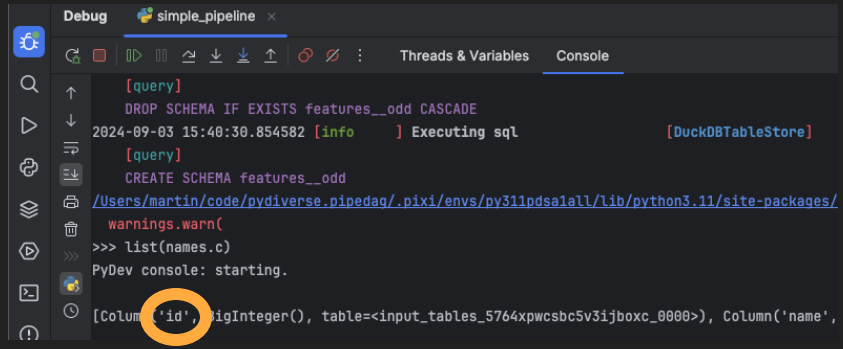
Use of IDEs / Debuggers in Data Science work

- We believe that Data Scientists and Engineers should work with same tools



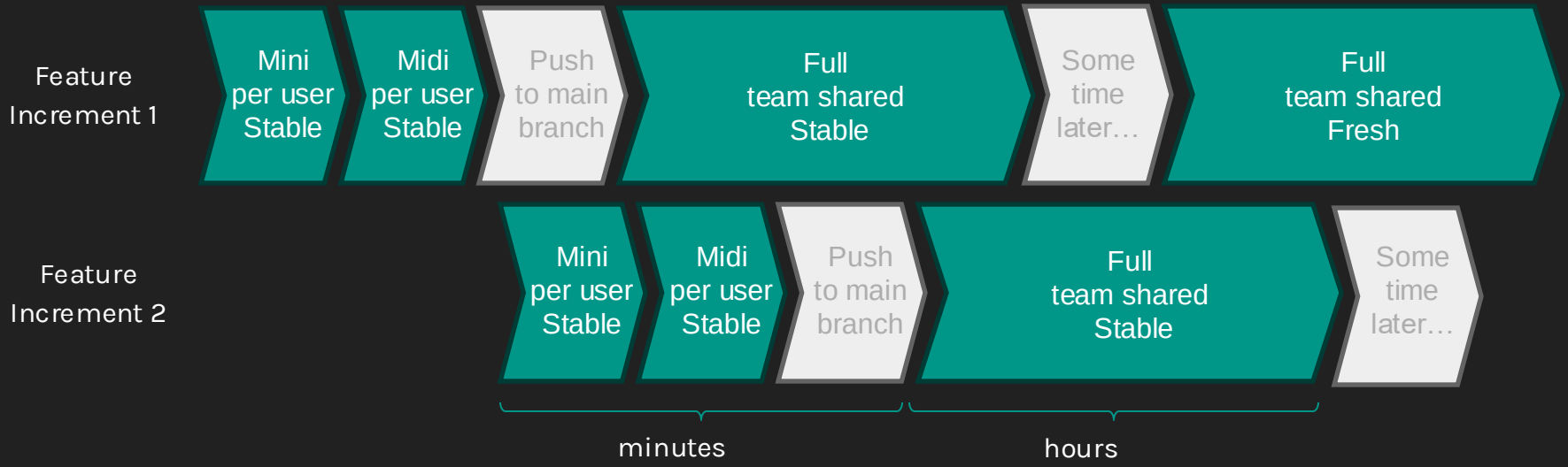




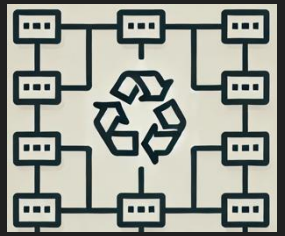


Pipeline instances (Manifestations with different input)

- Full / Midi / Mini Datasets
- Per user / Team shared
- Fresh inputs / Stable inputs



Automatic Cache Invalidation

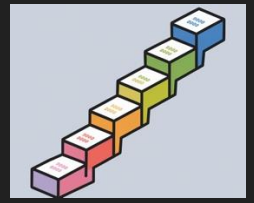


- Run what needs rerunning
- Manual updating reduces pipeline instances you can manage

```python
@materialize(version="1.0")
def input_table():
    return pd.DataFrame({
        "id": [1, 2, 3],
        "age": [20, 40, 60],
    })

@materialize(lazy=True, input_type=sa.Table)
def join_tables(names, ages):
    return sa.select(names, ages.c.age).join_from(
        names, ages, names.c.i == ages.c.id
    )

@materialize(version=AUTO_VERSION, input_type=pl.LazyFrame)
def enrich(ages):
    return ages.with_columns(age2=pl.col("age") * 2)
```

- version="1.0" ⇒ Manual versioning
  - We considered automatic python code change tracing too high a penalty when modifying shared code
- lazy=True ⇒ Automatic versioning
  - The code producing queries always runs
  - Queries will only be executed when they change
- version=AUTO_VERSION ⇒ Automatic versioning
  - The code will be executed twice:
    - Once with empty input data frames to avoid loading time
    - If lazy expression tree changed, again with full input

Gradual Improvement of code bases



```python
@materialize(lazy=True)
def all_tbls():
    return RawSql(f"""
        CREATE TABLE tbl_a AS
        SELECT 1;

        CREATE TABLE out AS
        SELECT
            a.*, a.x * a.x AS x2
        FROM tbl_a AS a;
    """)

with Flow() as flow:
    with Stage("s0"):
        all_tbls()
```

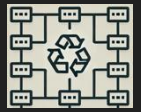
```python
@materialize(lazy=True)
def get_tbl_a():
    return sa.select("SELECT 1")

@materialize(input_type=sa.Table, lazy=True)
def get_out(a: sa.Alias):
    return sa.text(f"""
        SELECT
            a.*, a.x * a.x AS x2
        FROM {ref(a)} AS a
    """)

with Flow() as flow:
    with Stage("s0"):
        tbl_a = get_tbl_a()
        out = get_out(tbl_a)
```

```python
@materialize(lazy=True)
def get_tbl_a():
    return sa.select("SELECT 1")

@materialize(input_type=pdt.SQLTableImpl, lazy=True)
def get_out(a: pdt.Table):
    return a >> mutate(x2 = a.x * a.x)

with Flow() as flow:
    with Stage("s0"):
        tbl_a = get_tbl_a()
        out = get_out(tbl_a)
```

Summary: High iteration speed in insight generating loop

Multiple pipeline instances (mini/midi/full) to develop and push fast

IDE support with debugger on all (mini - full) pipeline instances



Automatic cache invalidation: only rerun what is affected by changes

Explorative SQL / Schema swapping / stage level transactionality

Gradual improvement of code bases (i.e. Raw SQL -> programmatic SQL)

Summary: High iteration speed in insight generating loop

⇒ Pydiverse pipedag provides you all the above with a syntax that allows you to focus on your main task:

```
# Writing data transformation code
@materialize(input_type=X)
def task(in_tbl: X):
    out_tbl = f(in_tbl)
    return out_tbl
```

Pipedag docs

Q & A

GitHub /
Feature requests

Notebook with
supported Libraries

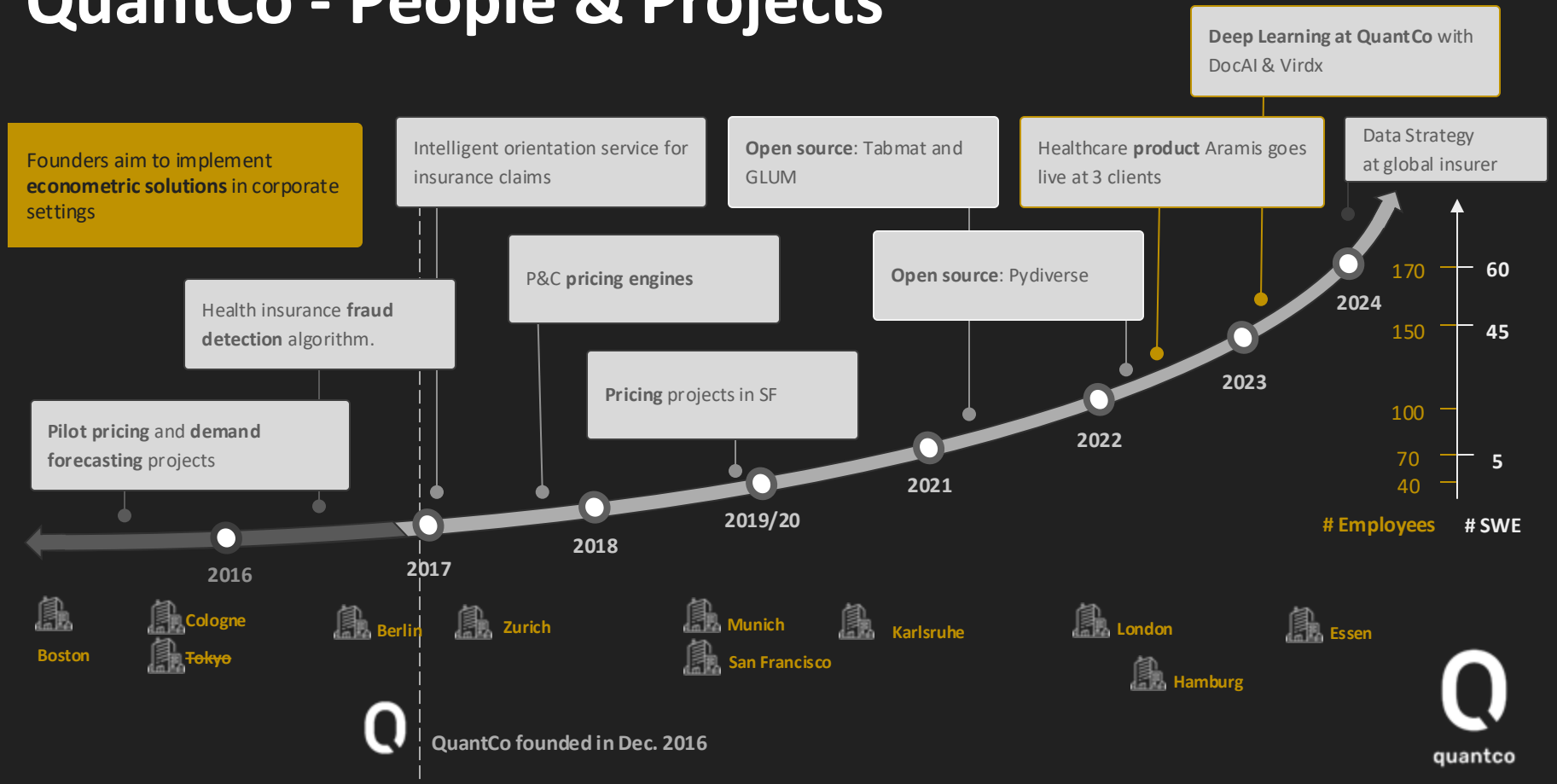
These slides

QuantCo - Overview

Company	Started in Cambridge, MA, by **PhDs** from Harvard and Stanford, QuantCo comprises more than **200 professionals** with extensive quantitative, engineering, and business experience
Locations	Boston (HQ), San Francisco, Berlin, Hamburg, London, Munich, Zürich, Karlsruhe
Value Proposition	QuantCo enters **long-term partnerships** with its clients to identify and realize customized **data-driven solution** with the most significant **economic impact and potential**: 1. Rigorous focus on value drivers in the business model 2. Close collaboration with the organization 3. Unique combination of econometric, statistical, and machine learning skills
Business Model	**Entrepreneurial deal structures** align the incentives of QuantCo and its clients, enabling the long-term collaboration needed to realize the full potential from analytical solutions

quantco

QuantCo - People & Projects



QuantCo is constantly looking for new colleagues to join us.

Internships

We offer off-cycle and summer internships for bachelor, master and PhD students throughout the entire year in data science and software engineering.

Full-time positions

We are currently focusing on our European offices. We offer positions in data science, software and machine learning engineering (among others).

Contact

carolin.thomas@quantco.com; please attach your CV and a recent copy of your transcript to an informal email (no cover letter needed)

quantco

Schema swapping / Stage level transactionality

Feature Increment 1

Mini per user Stable

Midi per user Stable

Push to main branch

Full team shared Stable

Data Scientist

Explorative SQL

Ingested raw data

Best possible representation to reason economically

Feature Enrichment

Model Training

Swap

Pipeline Run

Temporary Schema for new version of tables

Future developments

- Batched execution of dataframe tasks (with nice syntax)
- Extraction of production pipeline subgraph
  - Avoid writing back to table store (i.e. SQL Database) when it is not desirable
  - Arrow / Parquet backed table store
  - Syntax improvements for Arrow backed interactions of Pandas / Polars / DuckDB
- Pydiverse transform offers one nice syntax to describe data transformation that execute on SQL, dataframes, and ONNX
- Please reach out for your wishlist: https://github.com/pydiverse/pydiverse.pipedag/issues
- Contributions welcome!

Pipeline Template

Differences to pipeline shown in prev. slides:

- Early cleaning stage added
- Table dictionaries on highest level
- Test/train set split is computed early but tables included data about both (dangerous but convenient)
- CalibrationState offers communication link between training and production runs
- run_id related function calls can document runs in experiment tracking tools (ML Flow)
- InOutSpecification carries information about how to carve out subtree that runs in production (from input, ignore, const to output)

```
def get_pipeline(attrs):
    with Flow("typical_pipeline") as flow:
        with Stage("01_raw_input"):
            raw_tbls = read_input_data()
        with Stage("02_early_cleaning"):
            clean_tbls = clean(raw_tbls)
        with Stage("03_economic_representation"):
            train_test_set = mark_train_test_set(clean_tbls, **attrs["train_test_set"])
            tbls = economic_representation(clean_tbls, train_test_set)
        with Stage("04_features"):
            feature_tbls = features(tbls, train_test_set)
            # calibration_state holds dictionaries of parameters and parameter tables
            # which are computed during training based on train set but already applied
            # to rows of test set. For deployment, the calibration_state will be loaded
            # and injected as constant input into the pipeline subgraph that runs in
            # production.
            calibration_state = CalibrationState()
            feature_tbls2 = calibrated_features(tbls, feature_tbls, train_test_set, calibration_state)
            feature_tbls.update(feature_tbls2)  # will be executed lazily in consumer tasks
        with Stage("05_model_training"):
            run_id = get_run_id()
            input_data_train, target_train, encoding_paramters = model_encoding(
                tbls, feature_tbls, train_test_set, run_id, train=True
            )
            model, model_run_id, output_train = model_train(input_data_train, target_train, run_id)
        with Stage("06_model_evaluation"):
            run_id = get_run_id(run_id)  # get a run id if only evaluation is running
            document_link(run_id, model_run_id)
            input_data_test, target_test, _ = model_encoding(
                tbls, feature_tbls, train_test_set, run_id, encoding_parameters=encoding_paramters
            )
            output_test = model_predict(model, input_data_test, run_id)
            evaluation_result = evaluate_model(output_test, target_test, run_id)
            document_evaluation(evaluation_result, run_id)
    prod_in_out_spec = InOutSpecification(
        input=clean_tbls, ignore=[train_test_set, run_id],
        const=[model, model_run_id, feature_parameters, parameter_tbls, encoding_paramters], output=output_test,
    )
    return flow, prod_in_out_spec
```

High development iteration speed

You should always be able to turn the best possible representation upside down (Rely on CI not stability)