Using Trino and Airflow for (almost) all your data problems

Trino Summit 2022 @ The Commonwealth Club, San Francisco

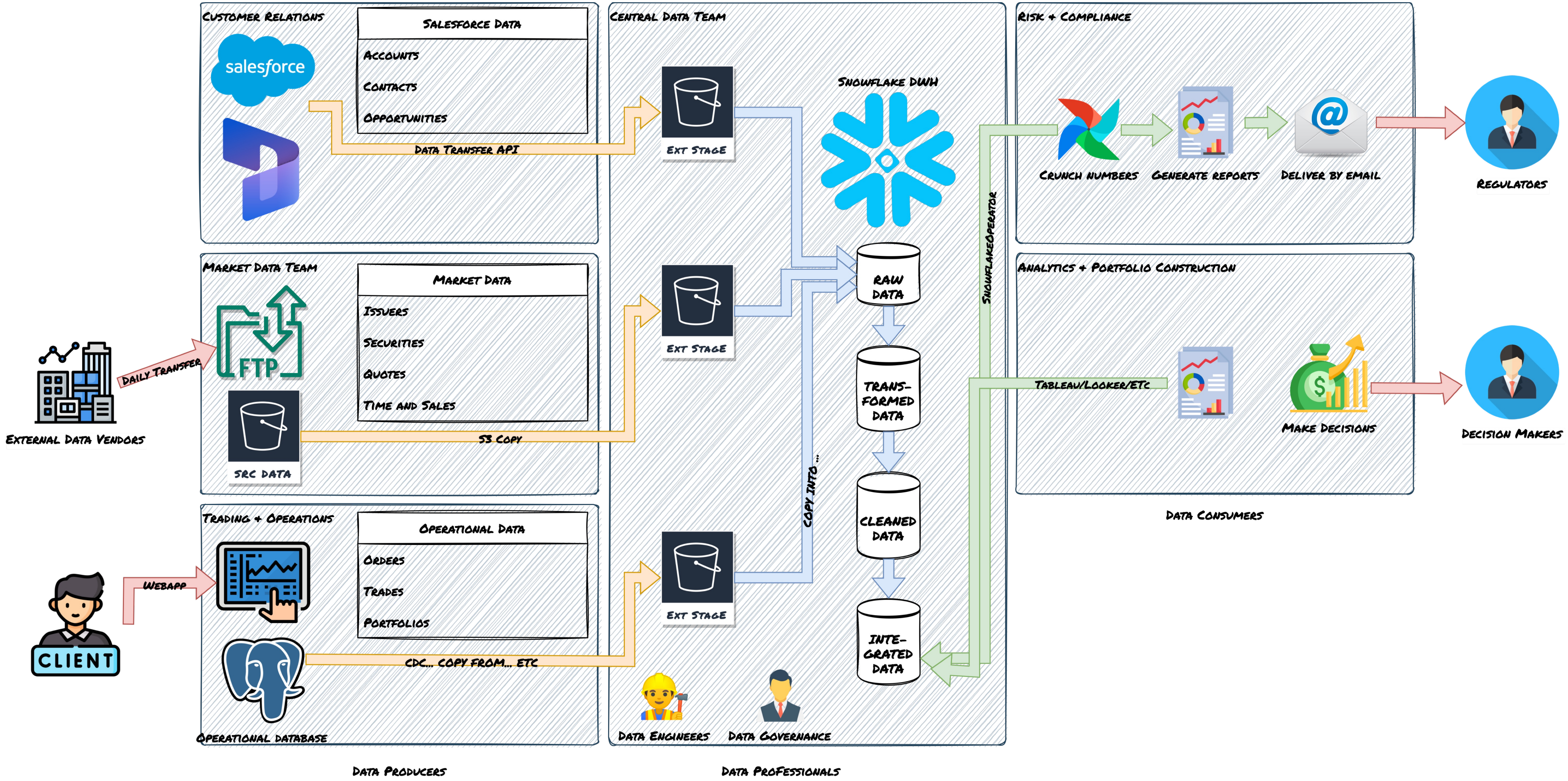
Philippe

- Your speaker this afternoon!
- Solutions Architect @ Astronomer, Inc. (we develop Apache Airflow commercially)
- Previously data engineering in the financial sector
- Last even attended pre-covid: Presto Summit NYC

Our agenda today

- The transition from a traditional to a federated data model
- Trino is not just for analytics
- Introducing Apache Airflow to orchestrate Trino queries
- Structuring Trino workloads on Apache Airflow

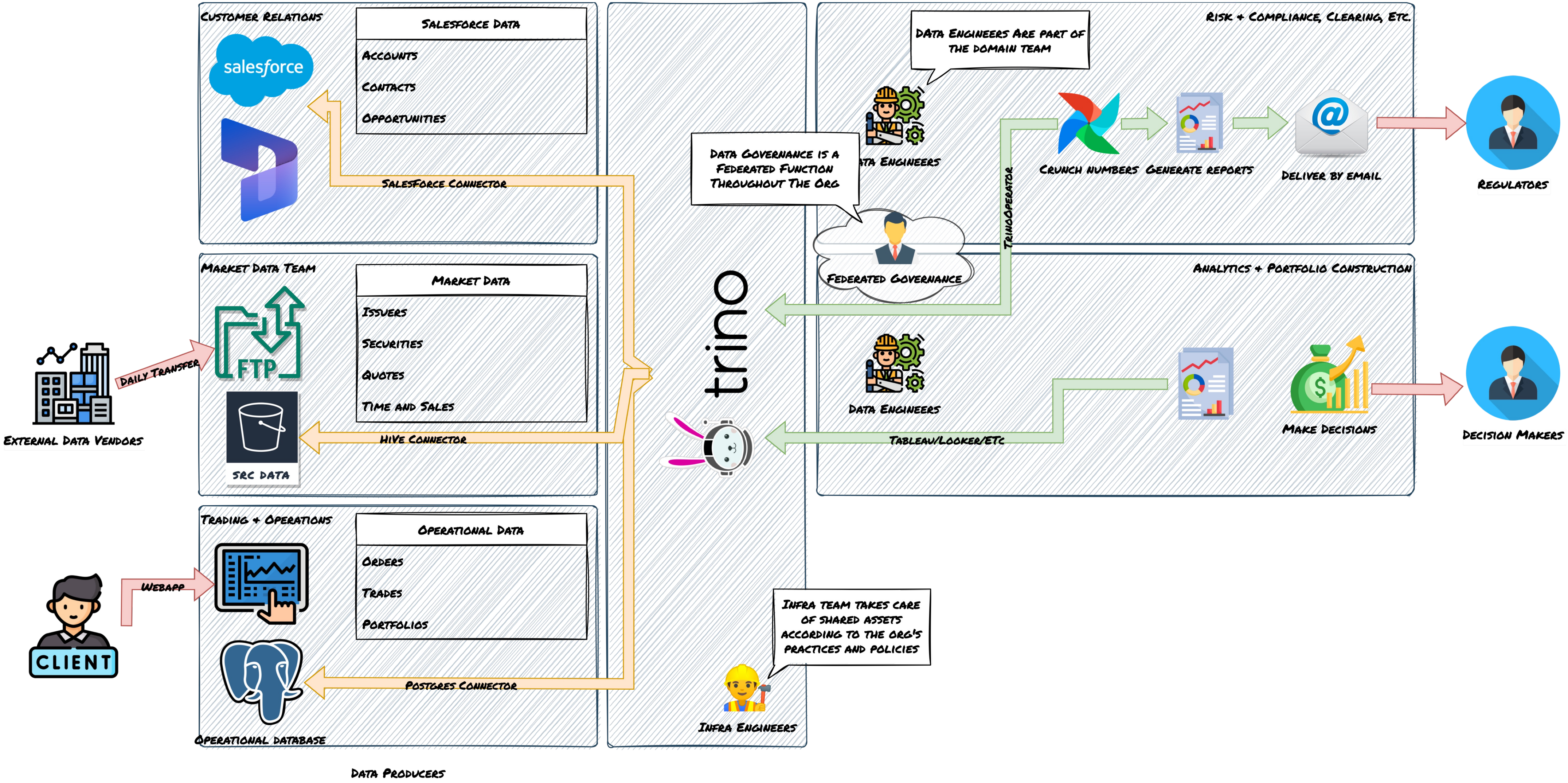
Traditional Approach



Traditional approach

- A central team has to be responsible for building an integration between a producer team and a central data platform.
- The data team views the producer's data from an external point of view and is further removed from the business context.
- The integrations they build are exposed to unpredictable changes in the source database, and while attempting to keep up with said changes, the data team can easily become a bottleneck for the business.

Federated data layer approach



Modest cluster, ~$12 a day

Instances (8) Info

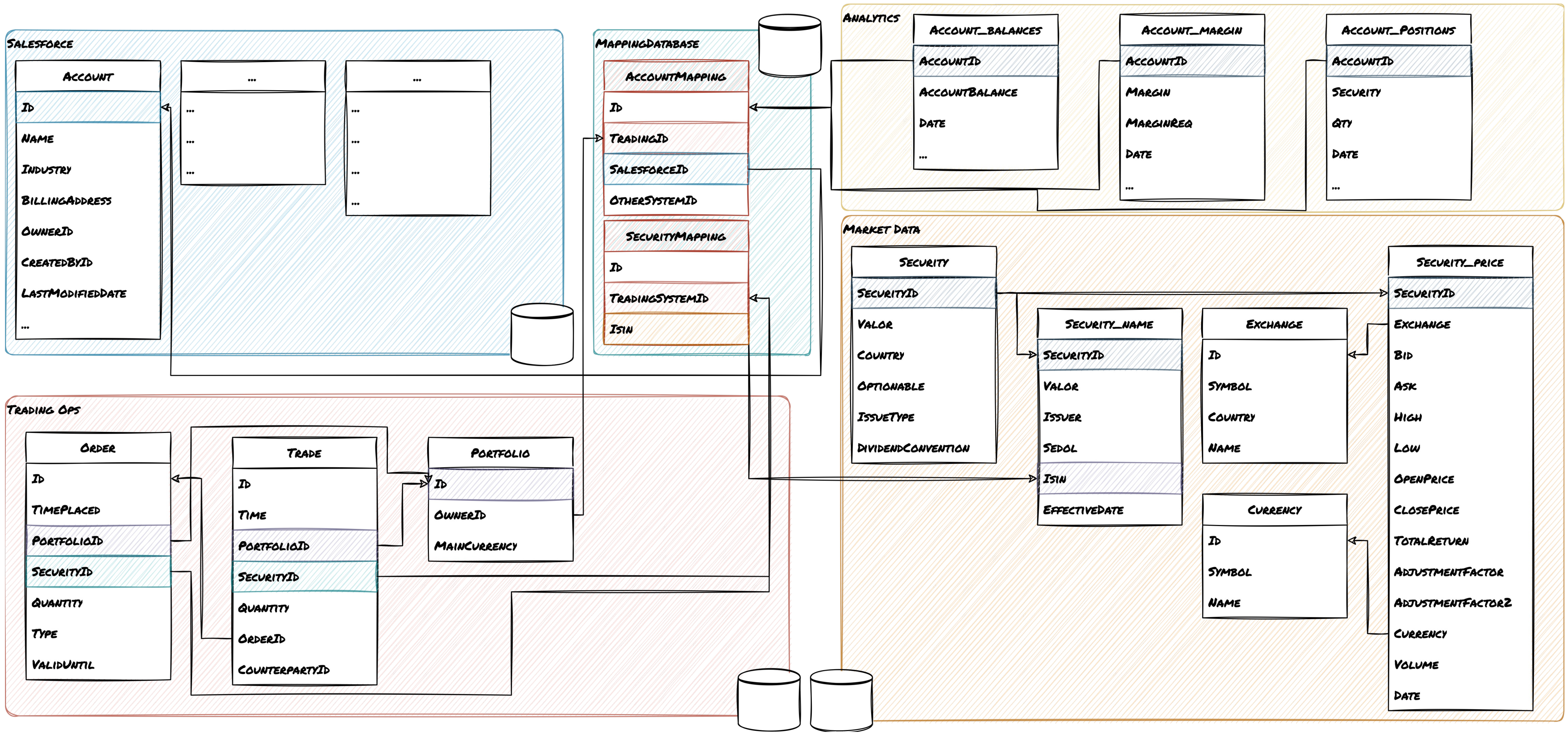
Connect | Instance state ▼

Find instance by attribute or tag (case-sensitive)

Instance state = running ✕ | Clear filters

	Name	Instance ID	Instance state	Instance type	Status check
☐	amazing-mon...	i-0ab0f2ff0745eb508	Running	r5b.xlarge	2/2 checks passed
☐	amazing-mon...	i-0936c887bbbc1983b	Running	r5dn.xlarge	2/2 checks passed
☐	amazing-mon...	i-04416d1480efd1c24	Running	r5n.xlarge	2/2 checks passed
☐	amazing-mon...	i-0cfd6785b0ade7cd5	Running	i4i.xlarge	2/2 checks passed
☐	amazing-mon...	i-067edeceab0df81ea	Running	r6id.xlarge	2/2 checks passed
☐	amazing-mon...	i-0ac35246bb81d0741	Running	r6id.xlarge	2/2 checks passed
☐	amazing-mon...	i-072f0c0a290d9eb2f	Running	r5.xlarge	2/2 checks passed
☐	amazing-mon...	i-0fa2d8d1d37699a64	Running	r5.xlarge	2/2 checks passed

Our federated data model



Trino is not just for analytics

- Fast, in-memory processing engine with newly introduced fault-tolerant functionalities for queries.
- Lots of connectors built-in and flexible SPI allows users to roll their own as long as data can be represented in tabular format.
- If built-in SQL functions are not good enough, it's possible to implement transformations using user defined functions.
- Run transformations that add value without having to explicitly move data to intermediate systems

create table catalog.schema.table as select * from <...> ... or insert into catalog.schema.table select * from <...> ...

But sometimes it needs a hand

Designing heavy batch workflows to run on Trino was challenging and required teams with specific skillsets.

Batch workloads often have complex interdependencies and sequencing requirements.

They are also often mission-critical processes, and their failure needs to be logged, alerted and handled.

In order to do this we use an orchestrator such as Apache Airflow.

What is Airflow?

An open-source platform for developing, scheduling, and monitoring batch-oriented workflows.

Originally developed at Airbnb by Max Beauchemin to orchestrate their batch workloads.

Open-sourced since 2015 under the Apache foundation umbrella.

It is a platform to programmatically define, author, schedule and monitor **workflows**.

Introduced the concept of defining orchestration workflows as python code.

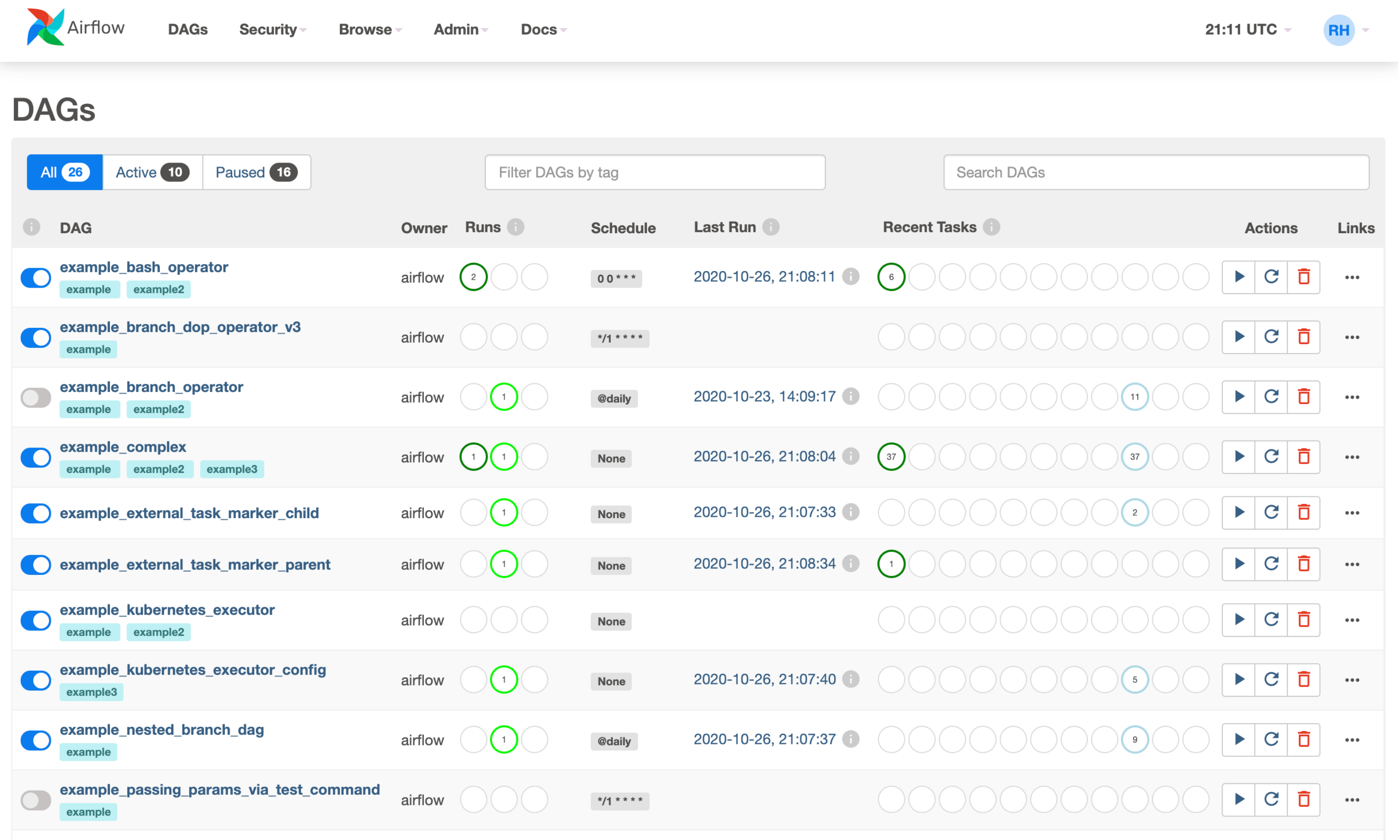
Strong community, constantly evolving. (28.1k github 🌟s, 10M downloads a month on PyPI).

Used by organizations everywhere, from small startups to F500 companies.

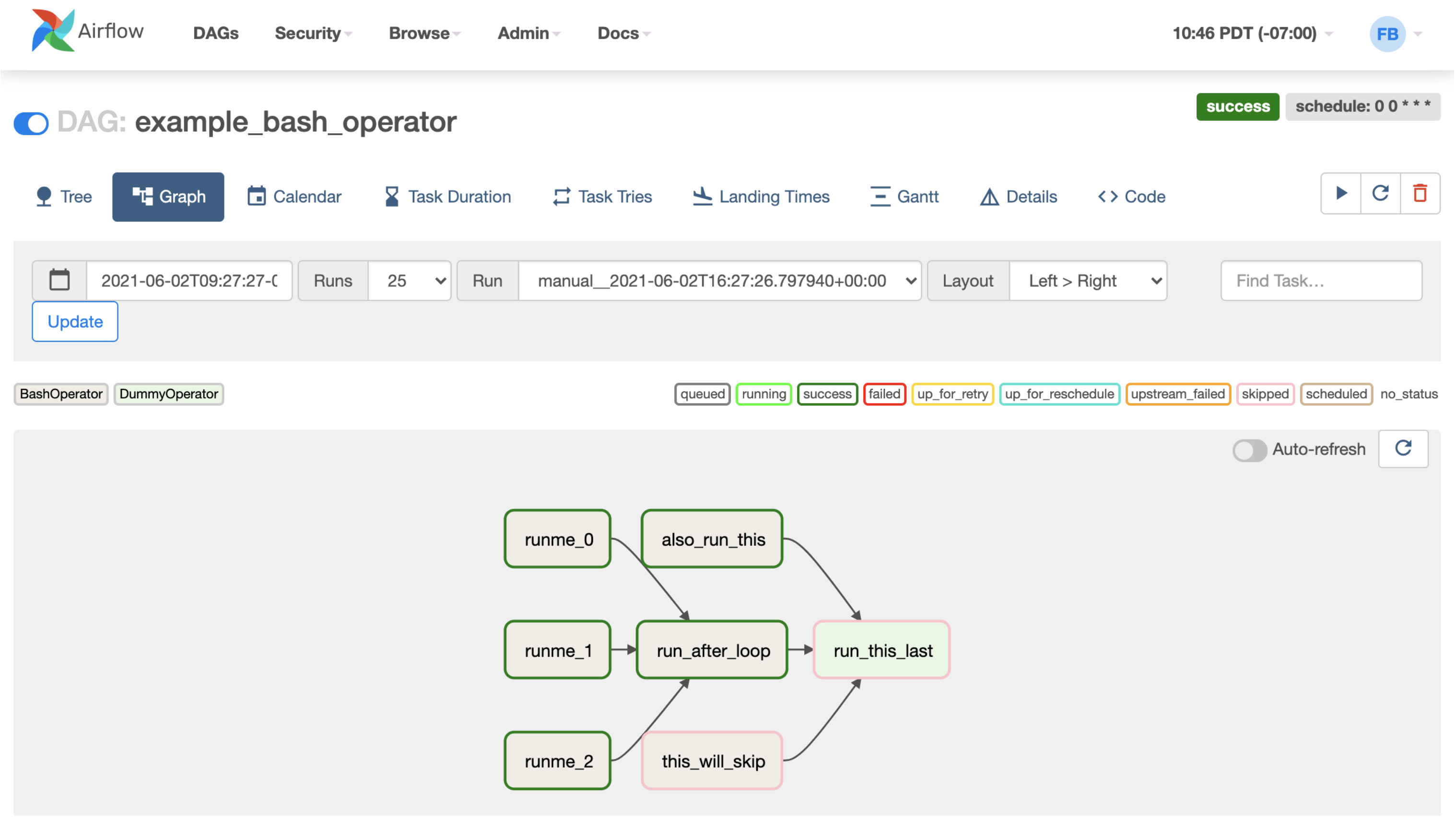
What is a DAG? Hello world.

```
from datetime import datetime

from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:

    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

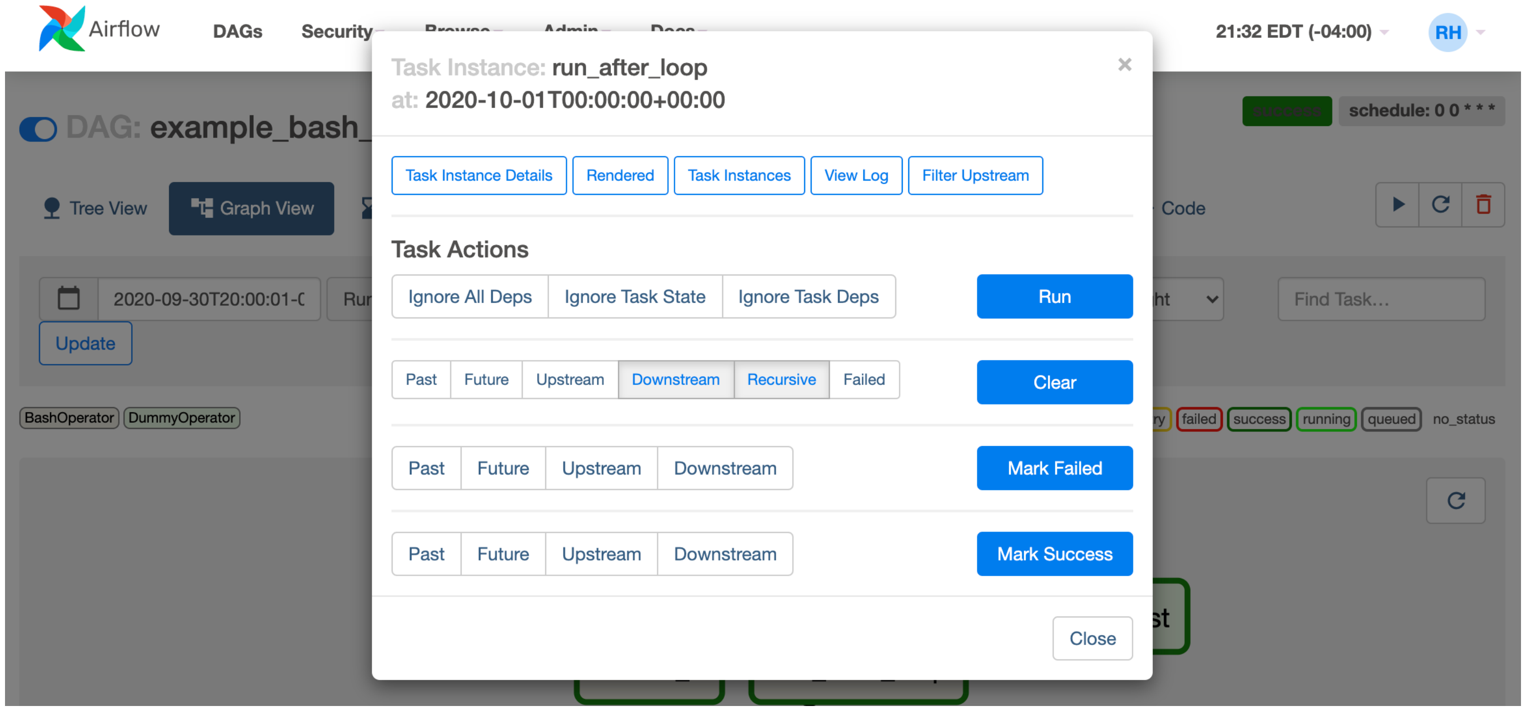
Monitor your DAGs



Monitor your tasks



Task actions

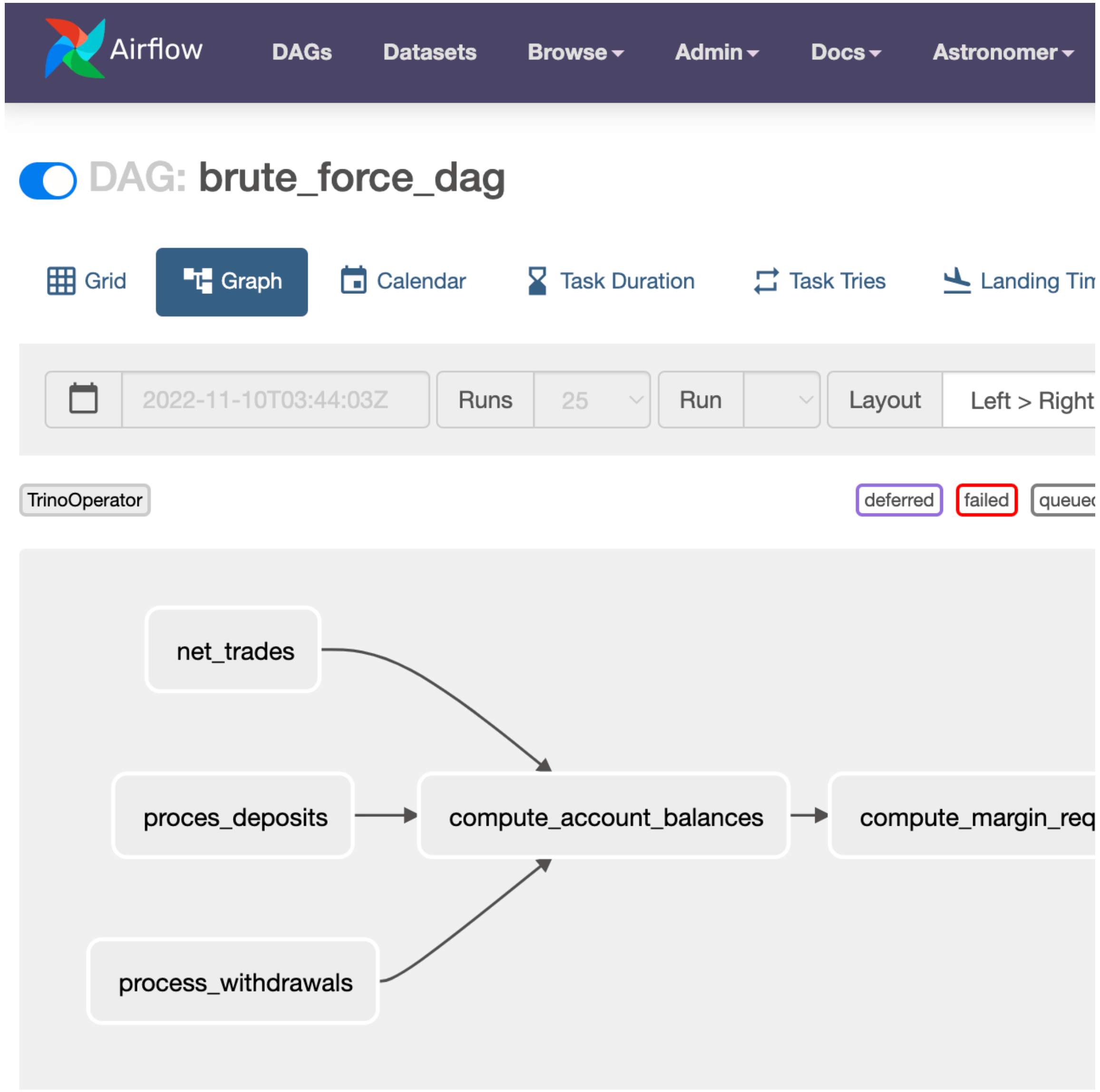


Structuring Trino workloads on Airflow

- A basic DAG
- Sharded DAG
- Dynamic task mapping
- Is this necessary with fault-tolerant execution?
- Data-aware scheduling

Basic DAG

- This is the simplest approach.
- Consists of running long, expensive queries on Trino as single Airflow task.
- Task failures are handled by the built-in Airflow retry mechanism.
- Main problems with this approach are that a lot of compute resources can be wasted if a task fails, and unreliable landing times.



The basic DAG

```
default_args = {
    "owner": "me",
    "start_date": pendulum.datetime(2021, 1, 1, tz="UTC"),
    "retries": 3,
    "retry_delay": timedelta(minutes=15),
    "catchup": False,
    "email_on_failure": True,
    "template_searchpath": "templates",
}
with DAG(dag_id="simple_dag",
        schedule_interval="@daily",
        default_args=default_args
        ) as dag:
```

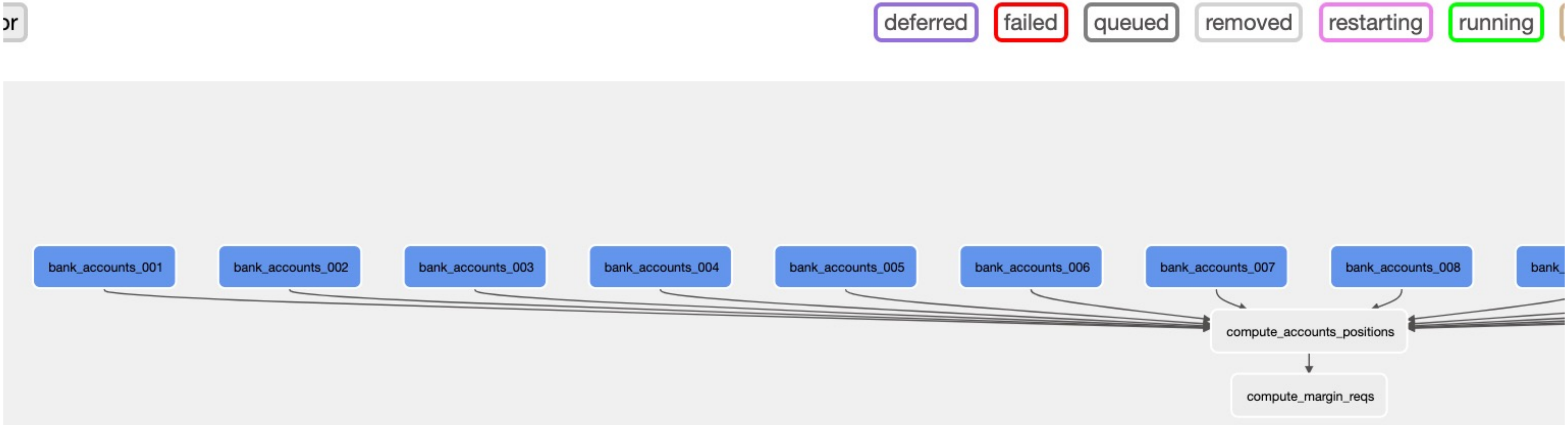
```
process_deposits = TrinoOperator(
        task_id="process_deposits",
        trino_conn_id="trino_default",
        sql="templates/process_deposits.sql",
        handler=list,
    )

    <...>

    [process_deposits, process_withdrawals, net_trades] >>
compute_account_balances >> compute_margin_reqs
```

process_deposits.sql:

```
insert into lake.banking.cash_position_offsets
select trading.id as account_id,
        trans.date, as date,
    sum(trans.credits) as credits,
    sum(trans.debits) as debits
from bankteam_app.public.transactions trans
join mappingdb.public.account_mapping m on trans.id = m.bank_id
join trading_db.account trading on trading.id = m.trading_id
where trans.date >= {{logical_date}}
group by trading.id, trans.date
```

Sharded structure

- This technique consists of splitting a long, expensive query into logical components, which output to durable storage.
- These "query components" are orchestrated by an orchestrator such as Apache Airflow.
- This allows the orchestrator to retry a smaller set of tasks in case of failure.



Sharded DAG

```
for bank_account_group in ["001", "002", "003", "004", "005",
"006", "007", "008", "009", "010"]:
    with TaskGroup(group_id=f"bank_accounts_{bank_account_group}") as group_:
        deposits_task, withdrawals_task = _create_bank_tasks(bank_account_group)


        group_ >> compute_account_balances

for trade_account_group_prefix in ["A", "B", "C", "D"]:
    with TaskGroup(group_id=f"trading_accounts_{trade_account_group_prefix}") as group_:
        net_trades_task = _create_trade_group_tasks(trade_account_group_prefix)


        group_ >> compute_account_balances

compute_account_balances >> compute_margin_reqs
```

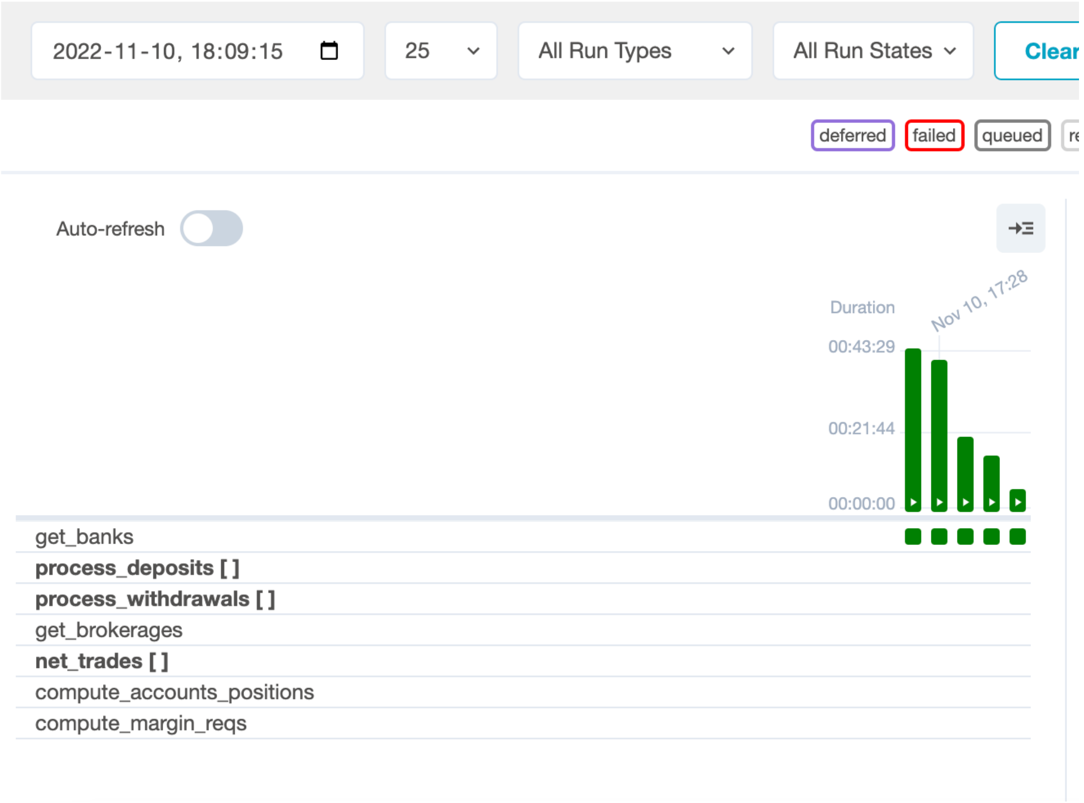
```
def _create_bank_tasks(account_group):
    process_deposits = TrinoOperator(
        task_id=f"process_deposits_{account_group}",
        trino_conn_id="trino_default",
        sql="templates/process_deposits.sql",
        handler=list,
        params={"account_group": account_group},
    )

    process_withdrawals = TrinoOperator(
        task_id=f"process_withdrawals_{account_group}",
        trino_conn_id="trino_default",
        sql="templates/process_withdrawals.sql",
        handler=list,
        params={"account_group": account_group},
    )
```

Templated SQL query

```
create table lake.banking.cash_position_offsets-{{params.account_group}}-{{run_id}} as
select trading.id as account_id,
        trans.date, as date,
     sum(trans.credits) as credits,
     sum(trans.debits) as debits
from bankteam_app.public.transactions trans
join mappingdb.public.account_mapping m on trans.id = m.bank_id
join trading_db.account trading on trading.id = m.trading_id
where trans.date between {{data_start_interval}} and {{data_end_interval}}
and trans.id like '{{ params.account_group }}%'
group by trading.id, trans.date
```

Dynamic task mapping

Allows DAG authors to generate tasks at runtime based on current data, rather than having to know ahead of time how many tasks would be needed.



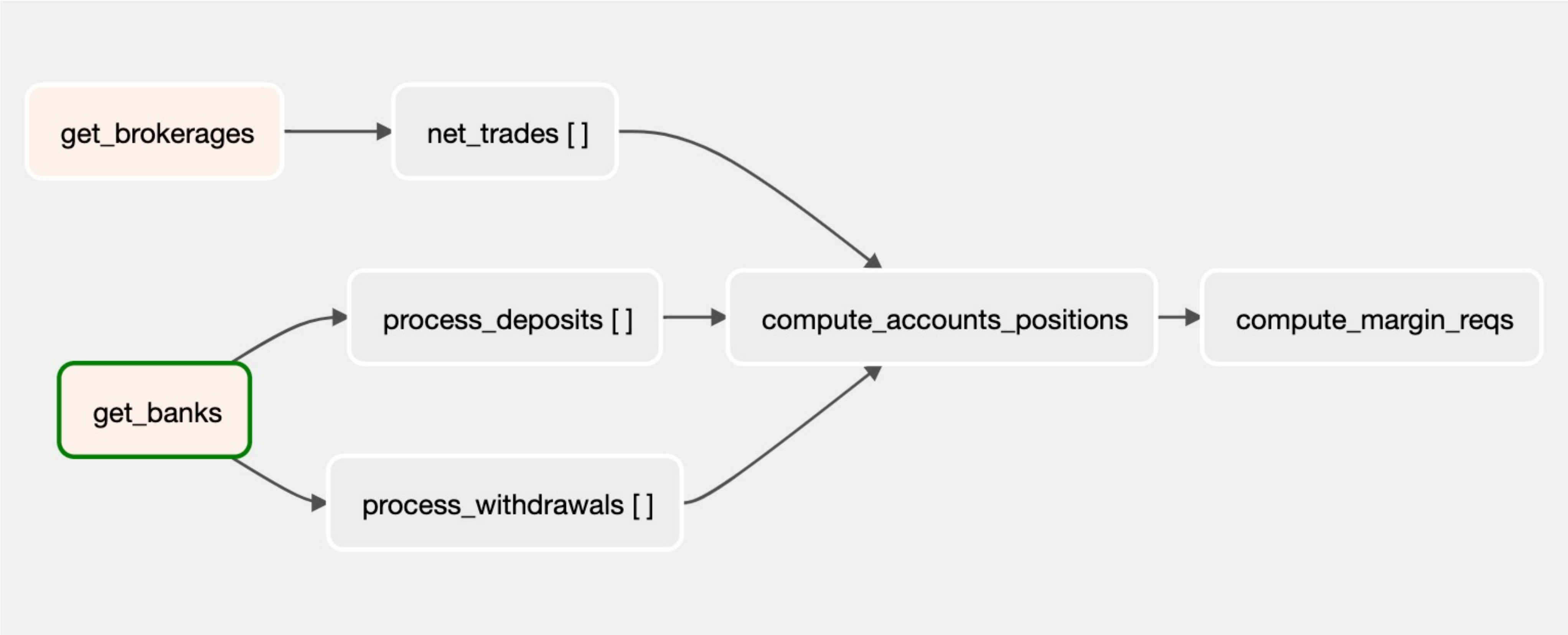
DAG code

```python
with DAG(dag_id="dtm_dag", schedule_interval=None, default_args=default_args) as dag:
    @task
    def get_banks():
        return TrinoHook().get_records(
            "select bank_id from portfolio_ops_db.public.banks"
        )

    process_deposits = TrinoOperator.partial(
        task_id=f"process_deposits",
        trino_conn_id="trino_default",
        sql="templates/process_deposits.sql",
        handler=list,
    ).expand(parameters=get_banks())
```

```python
(
    [process_deposits, process_withdrawals, net_trades]
    >> compute_account_balances
    >> compute_margin_reqs
)
```

```sql
insert into lake.banking.cash_position_offsets
select trading.id as account_id,
       trans.date, as date,
    sum(trans.credits) as credits,
    sum(trans.debits) as debits
from bankteam_app.public.transactions trans
join mappingdb.public.account_mapping m on trans.id = m.bank_id
join trading_db.account trading on trading.id = m.trading_id
where trans.date >= {{logical_date}}
and trans.counterparty_bank = ?
group by trading.id, trans.date
```

Graph view

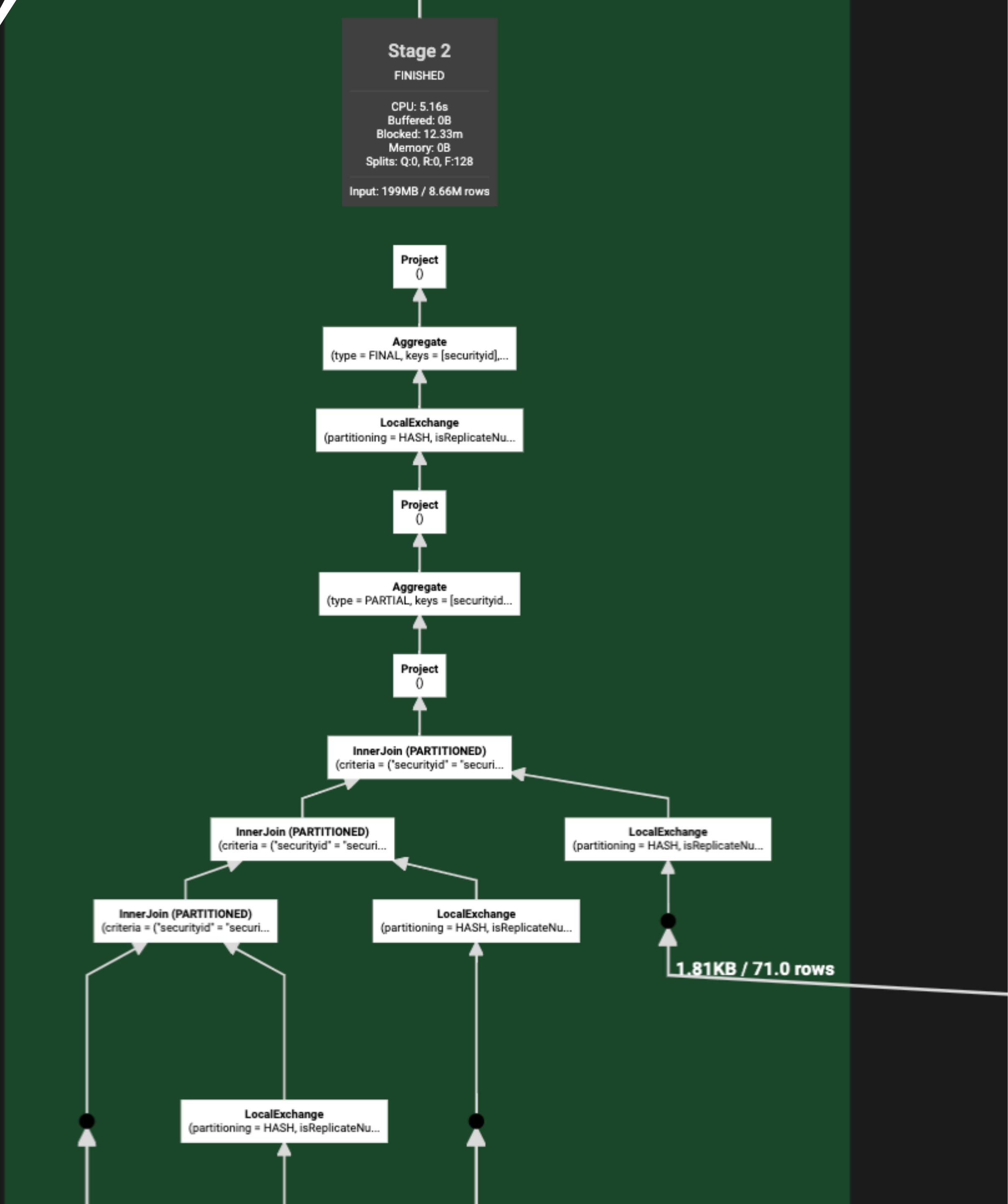

get_brokerages
net_trades [ ]
process_deposits [ ]
compute_accounts_positions
compute_margin_reqs
get_banks
process_withdrawals [ ]

Fault-tolerant execution on Trino

- Introduces task and query based retries in Trino
- Retry policy configures whether Trino retries whole queries, or individual tasks within a query
- Task-based retries are appropriate for large batch workloads, but can introduce overhead for small queries
- Task-based retries require an exchange manager to be configured. This component is responsible for spooling task data for fault-tolerant execution.
- The exchange manager should use object storage as a backend for scalability

Trino

- Trino queries are split into a series of stages
- These stages are split into tasks which are the actual execution units of a Trino query
- With a proper exchange manager configured task output is spooled to shared storage



Data-aware scheduling

- In version 2.4, Airflow introduced "data-aware scheduling" as a feature.
- A dataset is a stand-in for a logical grouping of data.
- Allows DAGs to be scheduled based on another task updating a dataset.
- In a Trino setting, this allows a team to launch a batch job that consumes a dataset produced by another team based on interdependent transformations in a decoupled yet explicit way.

Data-aware DAG code

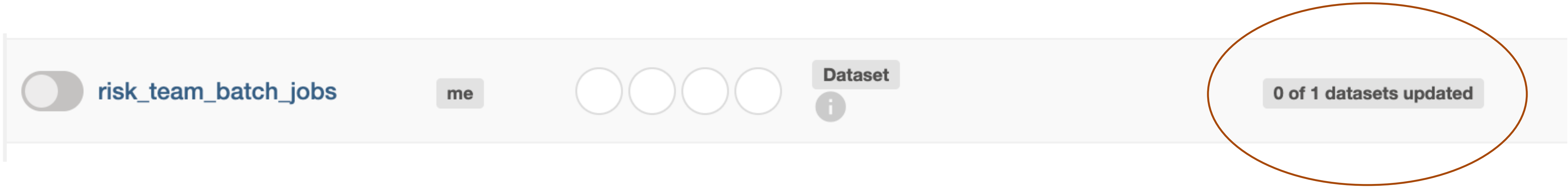
Defining outlets

```
compute_account_balances = TrinoOperator(
    task_id="compute_accounts_balances",
    sql="sql/compute_accounts_balances.sql",
    handler=list,
     outlets=[Dataset("trino://lake.analytics.account_balances")],
)

@task(outlets=[Dataset("trino://lake.analytics.account_margin_reqs")])
def compute_margin_requirements():

...
```

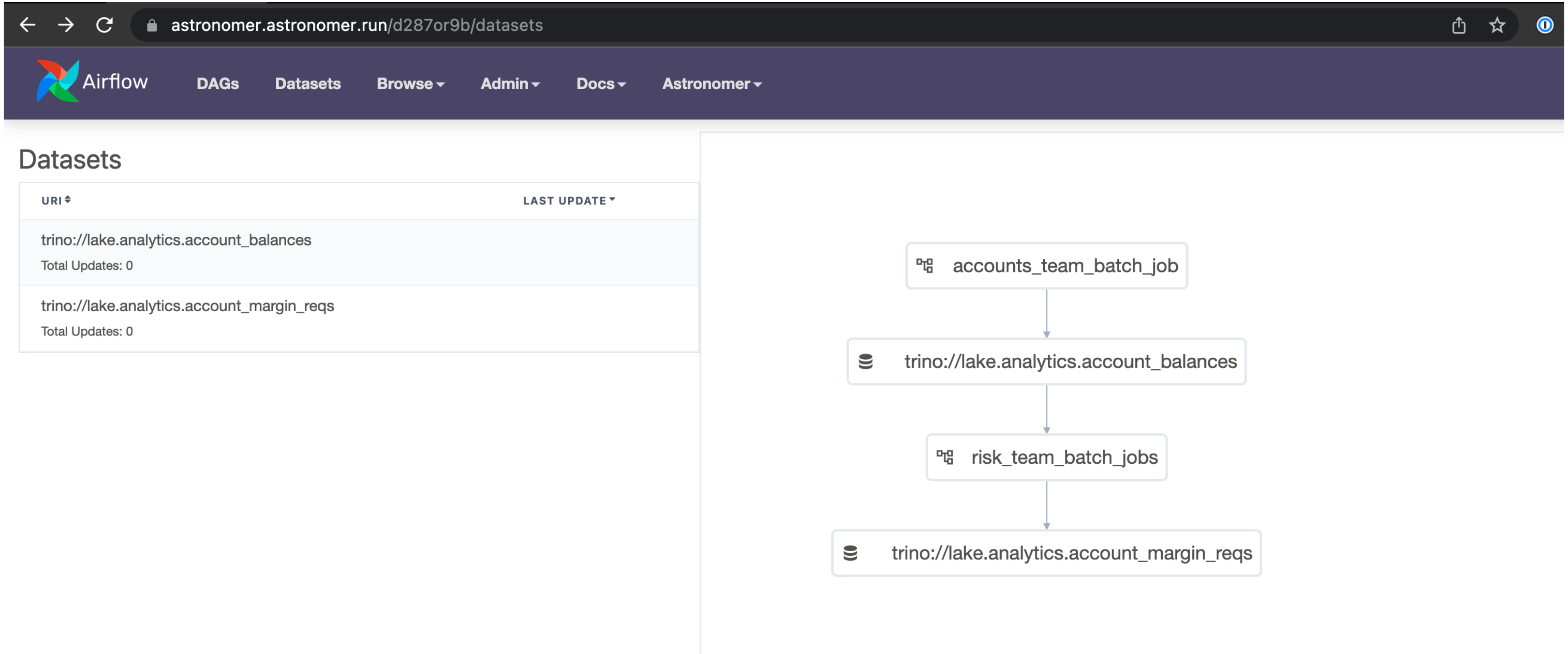
Consuming datasets

```
with DAG(
    dag_id="risk_team_batch_jobs",

schedule=[Dataset("trino://lake.analytics.account_balances")],
    default_args=default_args,
) as dag:

...
```

Data-aware DAG schedule



Datasets view



Takeaways

- The "brute force" method to running Trino is viable, but task-based fault tolerance should be enabled on your cluster.
- In fact, I would recommend enabling task-based fault tolerance by default if your tasks run for over fifteen minutes on average.
- Trino task-based fault tolerance reduces the need for shards in your code.
- Dynamic task mapping is a great way to structure your workflows if you need to adapt their structure at runtime.
- You can produce "datasets" so that other Airflow users within your org can use your data products efficiently with data-aware scheduling.

Questions? I can answer.⁉️

Thank you! 🦾

Reach out to me on:

Slack, Twitter, LinkedIn, Email, Phone, Signal, Telegram, Mastodon, Facebook, Instagram, or even offline.