

A photograph of the Golden Gate Bridge in San Francisco, California, taken at dusk. The bridge's red-orange towers and suspension cables are silhouetted against a dark, cloudy sky. The water of the bay is dark, and the foreground shows a sandy beach with some rocks. The overall mood is serene and iconic.

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

Trino Summit 2022 @ The Commonwealth Club, San Francisco

Philippe

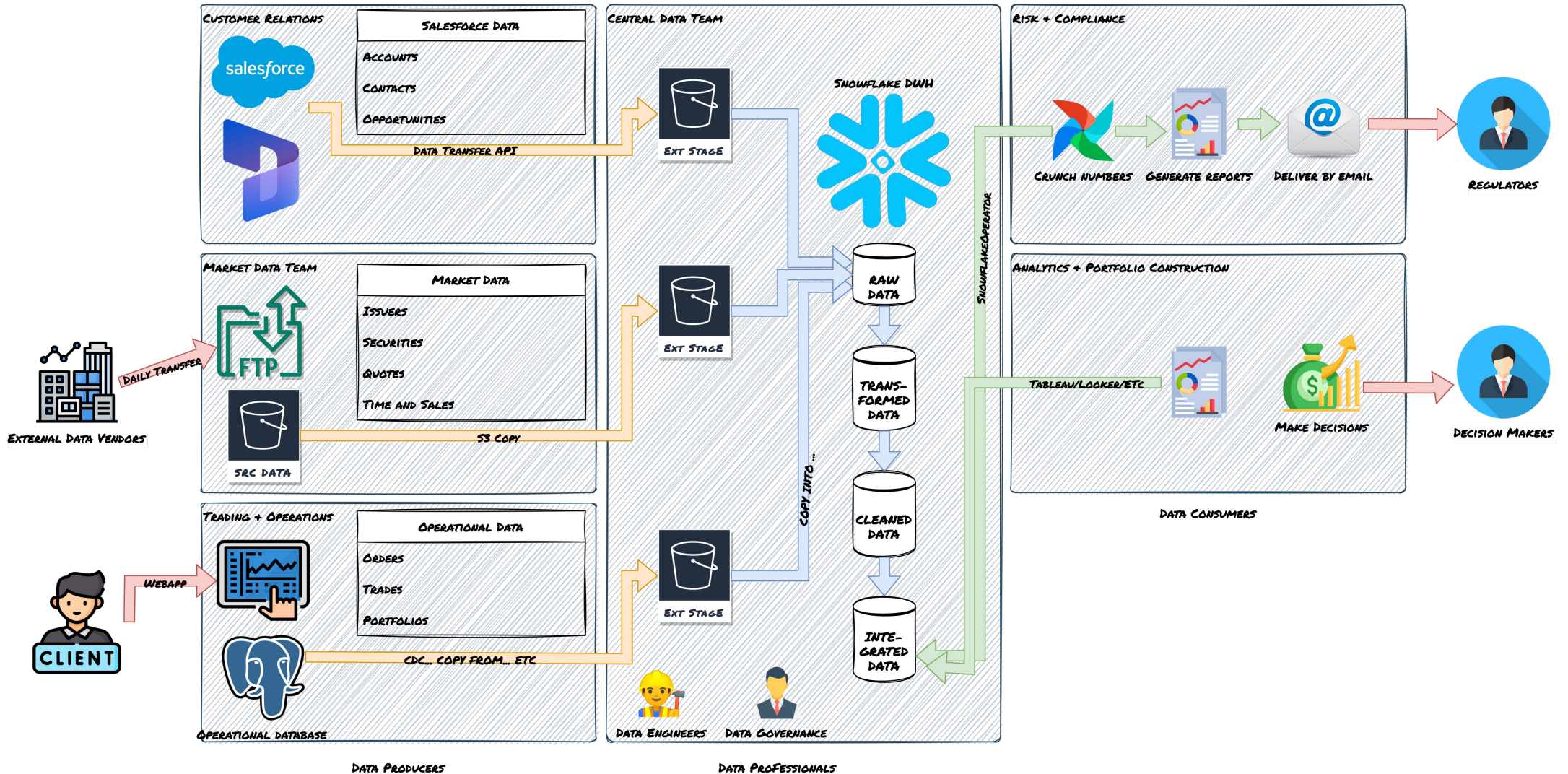
- Your speaker this afternoon!
- Solutions Architect @ [Astronomer, Inc.](#) (we develop Apache Airflow commercially)
- Previously data engineering in the financial sector
- Last even attended pre-covid: Presto Summit NYC



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

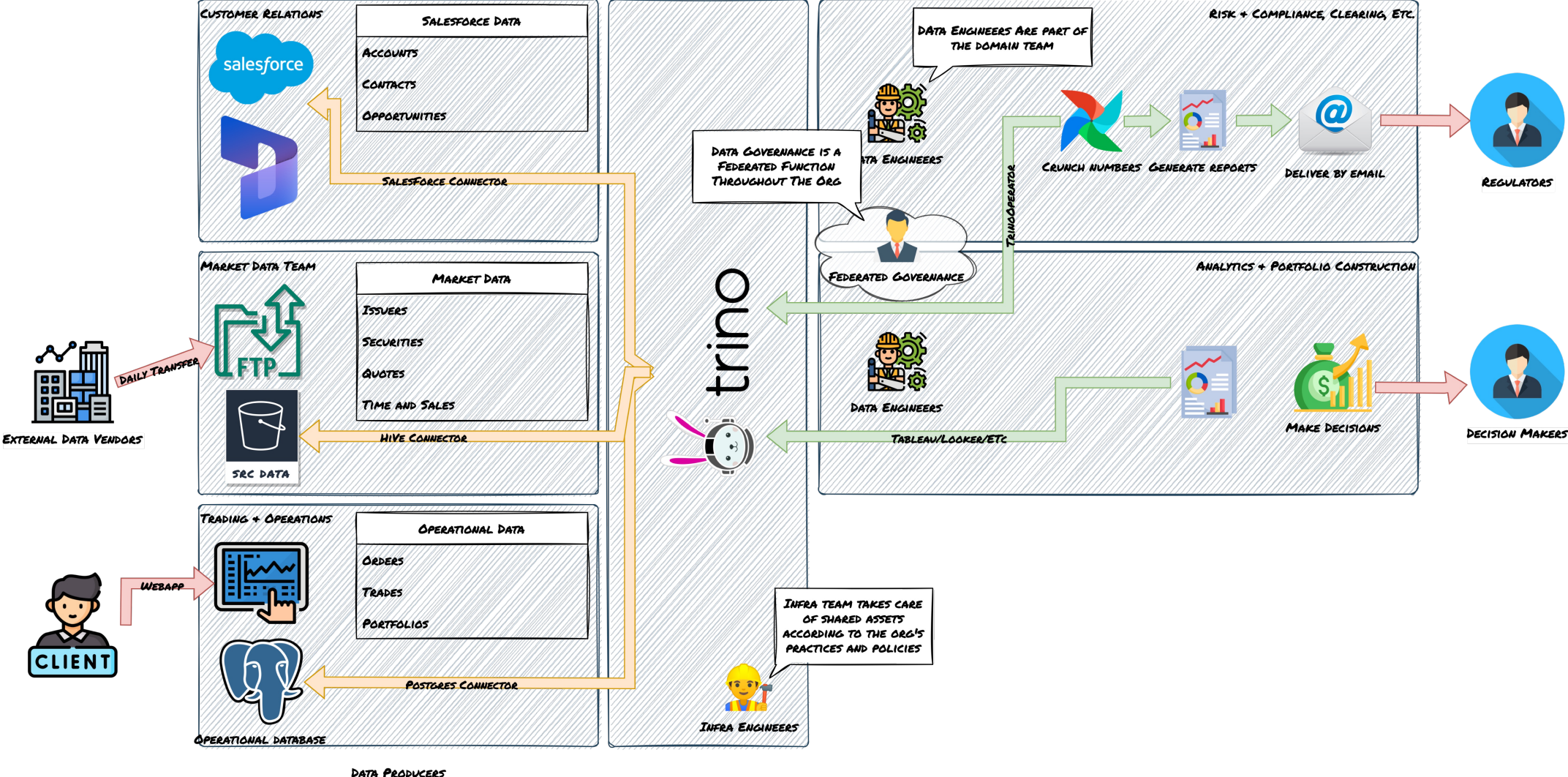


Traditional approach

- 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



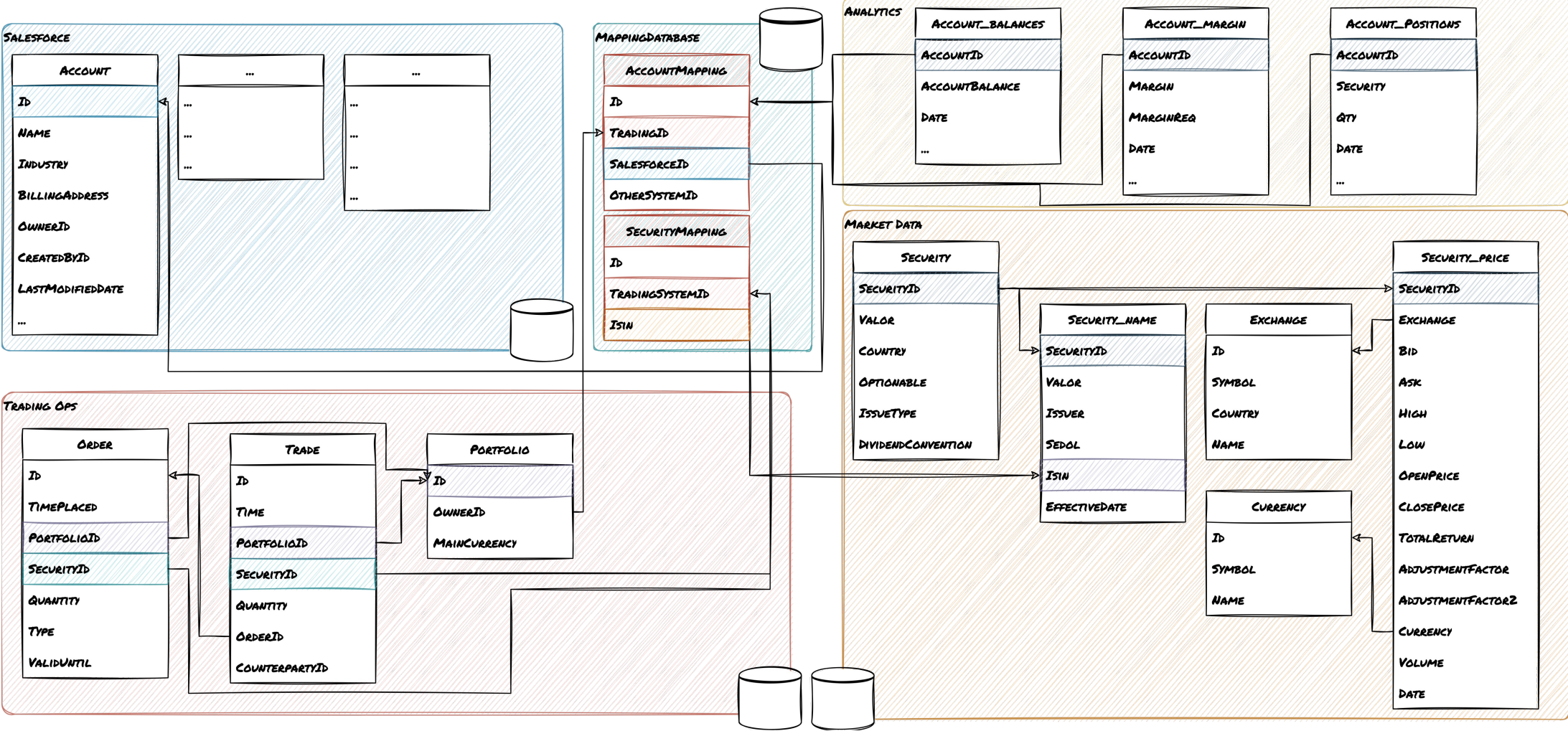
Modest cluster, ~\$12 a day

Instances (8) [Info](#) ↻ Connect Instance state ▼

Instance state = running ✕ Clear filters

<input type="checkbox"/>	Name ▼	Instance ID	Instance state ▼	Instance type ▼	Status check
<input type="checkbox"/>	amazing-mon...	i-0ab0f2ff0745eb508	✔ Running 🔍	r5b.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-0936c887bbbc1983b	✔ Running 🔍	r5dn.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-04416d1480efd1c24	✔ Running 🔍	r5n.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-0cfd6785b0ade7cd5	✔ Running 🔍	i4i.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-067edeceab0df81ea	✔ Running 🔍	r6id.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-0ac35246bb81d0741	✔ Running 🔍	r6id.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-072f0c0a290d9eb2f	✔ Running 🔍	r5.xlarge	✔ 2/2 checks passed
<input type="checkbox"/>	amazing-mon...	i-0fa2d8d1d37699a64	✔ Running 🔍	r5.xlarge	✔ 2/2 checks passed

Our federated data model



Trino is not just for analytics

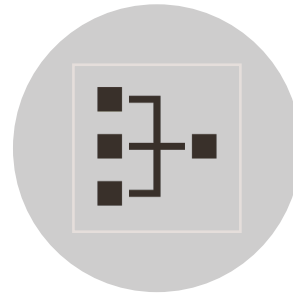
- Fast, in-memory processing engine with newly introduced fault-tolerant functionalities for queries.
- Lots of connectors built-in and flexible SPI allows users to roll their own as long as data can be represented in tabular format.
- If built-in SQL functions are not good enough, it's possible to implement 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

The screenshot shows the Apache Airflow web interface for monitoring DAGs. At the top, there are navigation links for Airflow, DAGs, Security, Browse, Admin, and Docs. The current time is 21:11 UTC. The main heading is 'DAGs'. Below this, there are filters for 'All 26', 'Active 10', and 'Paused 16'. A search bar is available for 'Search DAGs'. The main content is a table with columns: DAG, Owner, Runs, Schedule, Last Run, Recent Tasks, Actions, and Links. The table lists several example DAGs with their respective statuses and task counts.

DAG	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
<input checked="" type="checkbox"/> example_bash_operator <small>example example2</small>	airflow	2	00***	2020-10-26, 21:08:11	6	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_branch_dop_operator_v3 <small>example</small>	airflow	0	*1****		0	▶ ↺ 🗑️ ⋮	
<input type="checkbox"/> example_branch_operator <small>example example2</small>	airflow	1	@daily	2020-10-23, 14:09:17	11	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_complex <small>example example2 example3</small>	airflow	1	None	2020-10-26, 21:08:04	37	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33	2	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34	1	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_kubernetes_executor <small>example example2</small>	airflow	0	None		0	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_kubernetes_executor_config <small>example3</small>	airflow	1	None	2020-10-26, 21:07:40	5	▶ ↺ 🗑️ ⋮	
<input checked="" type="checkbox"/> example_nested_branch_dag <small>example</small>	airflow	1	@daily	2020-10-26, 21:07:37	9	▶ ↺ 🗑️ ⋮	
<input type="checkbox"/> example_passing_params_via_test_command <small>example</small>	airflow	0	*1****		0	▶ ↺ 🗑️ ⋮	

Monitor your tasks

The screenshot displays the Apache Airflow web interface for a DAG named 'example_bash_operator'. The top navigation bar includes links for 'Airflow', 'DAGs', 'Security', 'Browse', 'Admin', and 'Docs'. The current time is 10:46 PDT (-07:00) and the user is logged in as 'FB'. The DAG status is 'success' with a schedule of '0 0 ***'. Below the status bar, there are navigation tabs for 'Tree', 'Graph', 'Calendar', 'Task Duration', 'Task Tries', 'Landing Times', 'Gantt', 'Details', and 'Code'. The 'Graph' tab is selected. A filter bar shows a date range of '2021-06-02T09:27:27-C', 'Runs' set to '25', and a specific run ID 'manual__2021-06-02T16:27:26.797940+00:00'. There is an 'Update' button and a 'Find Task...' search box. Below the filter bar, there are tabs for 'BashOperator' and 'DummyOperator', and a legend for task states: 'queued', 'running', 'success', 'failed', 'up_for_retry', 'up_for_reschedule', 'upstream_failed', 'skipped', 'scheduled', and 'no_status'. The main area shows a task dependency graph with the following tasks and dependencies:

```
graph LR; runme_0[runme_0] --> also_run_this[also_run_this]; runme_1[runme_1] --> run_after_loop[run_after_loop]; runme_2[runme_2] --> run_after_loop; also_run_this --> run_this_last[run_this_last]; run_after_loop --> run_this_last; this_will_skip[this_will_skip] --> run_this_last;
```

The graph shows three parallel tasks (runme_0, runme_1, runme_2) feeding into a central task (run_after_loop). runme_0 also feeds into also_run_this, which then feeds into run_this_last. Additionally, this_will_skip feeds into run_this_last. The runme_0, also_run_this, runme_1, and run_after_loop tasks are highlighted in green, while run_this_last and this_will_skip are highlighted in pink.

Task actions

The screenshot displays the Apache Airflow web interface. At the top, the navigation bar includes 'Airflow', 'DAGs', 'Security', 'Browse', 'Admin', and 'Docs'. The current page is 'DAG: example_bash'. A modal window is open, titled 'Task Instance: run_after_loop at: 2020-10-01T00:00:00+00:00'. The modal contains several sections:

- Navigation:** Buttons for 'Task Instance Details', 'Rendered', 'Task Instances', 'View Log', and 'Filter Upstream'.
- Task Actions:** A section with buttons for 'Ignore All Deps', 'Ignore Task State', 'Ignore Task Deps', and a blue 'Run' button.
- Filtering:** A row of buttons for 'Past', 'Future', 'Upstream', 'Downstream' (selected), 'Recursive', and 'Failed', followed by a blue 'Clear' button.
- Marking:** A row of buttons for 'Past', 'Future', 'Upstream', and 'Downstream', followed by a blue 'Mark Failed' button.
- Marking:** A row of buttons for 'Past', 'Future', 'Upstream', and 'Downstream', followed by a blue 'Mark Success' button.
- Close:** A 'Close' button at the bottom right of the modal.

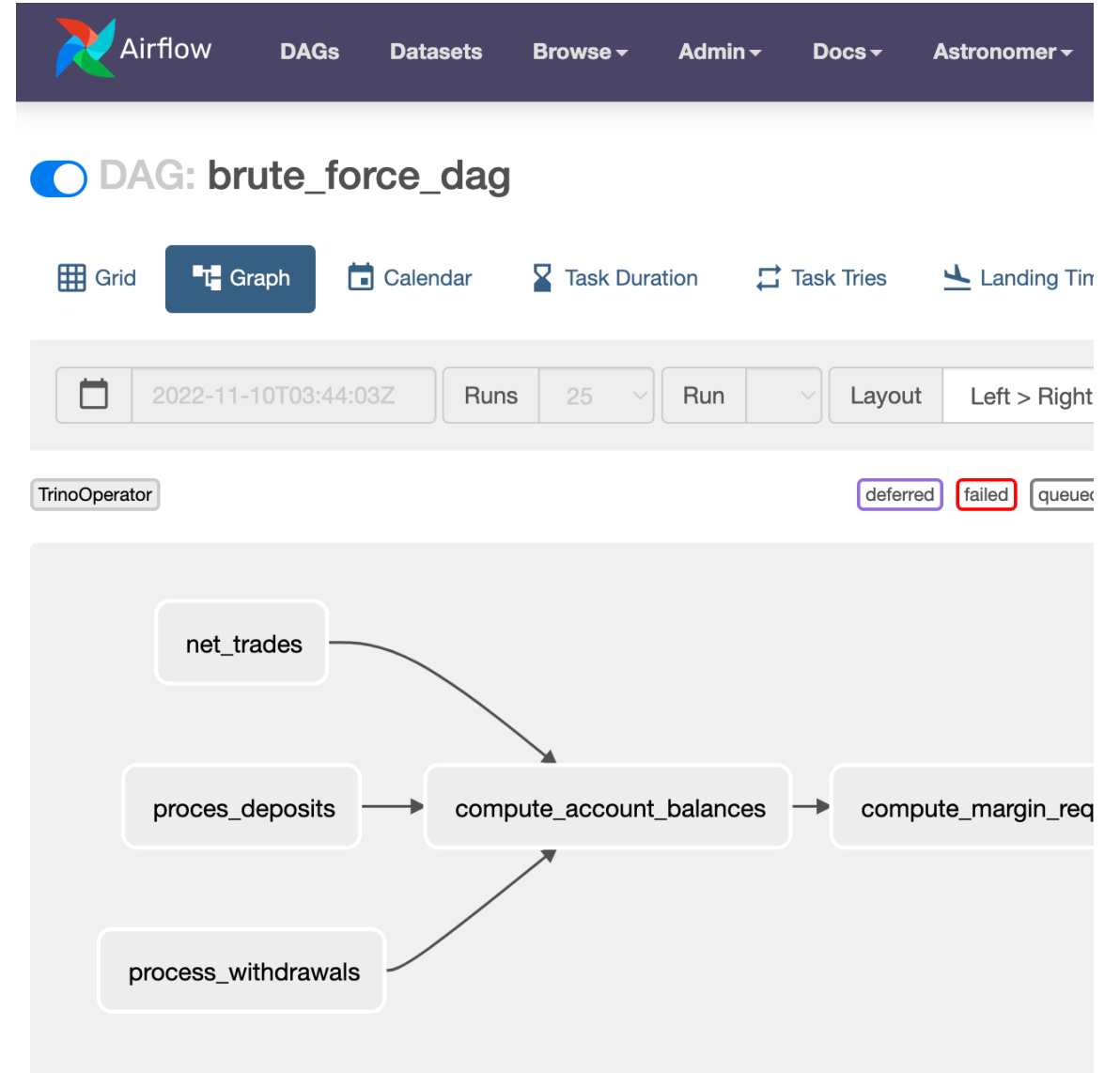
The background interface shows a DAG graph with nodes 'BashOperator' and 'DummyOperator'. A task instance is shown with a status of 'running' and a schedule of '0 0 ***'. The top right corner shows the time '21:32 EDT (-04:00)' and a user profile 'RH'.

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

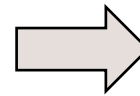
- This is the simplest approach.
- Consists of running long, expensive queries on Trino as single Airflow task.
- Task failures are handled by the built-in Airflow retry mechanism.
- Main problems with this approach are that a lot of compute resources can be wasted if a task fails, and unreliable landing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

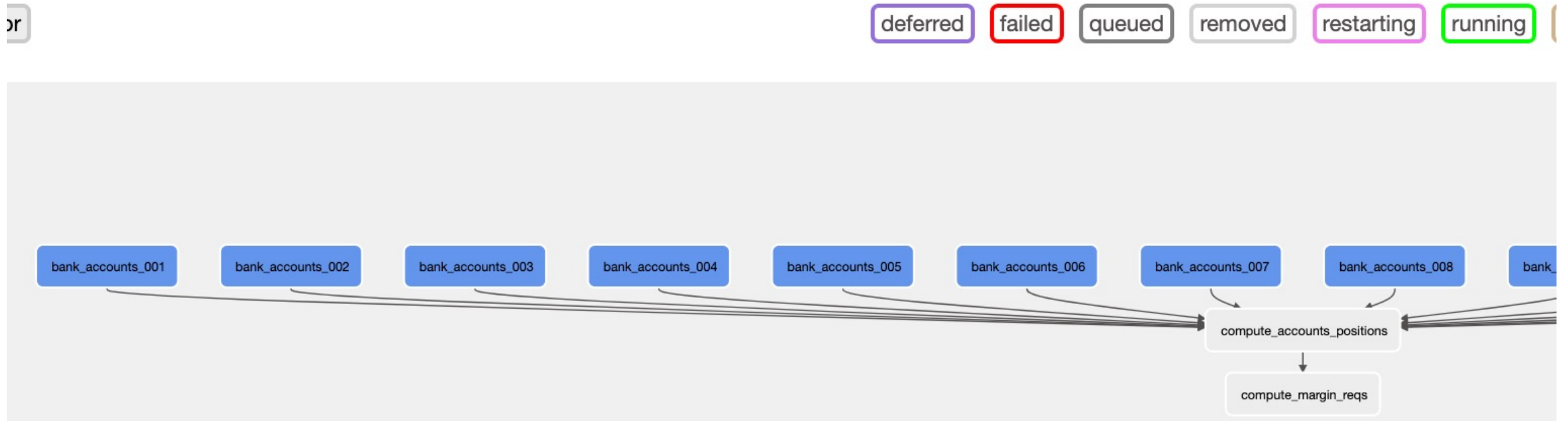


process_deposits.sql:

```
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},
```

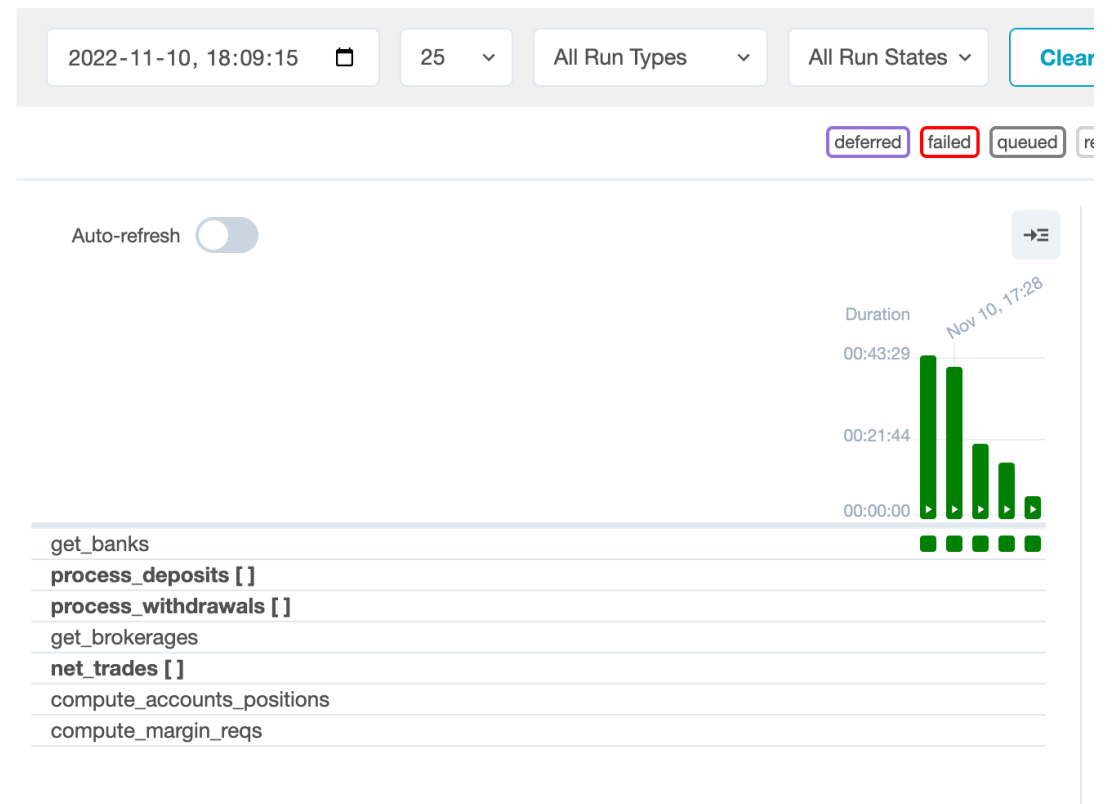
```
    )
```

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

```
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())
```

```
(
```

```
[process_deposits, process_withdrawals, net_trades]
```

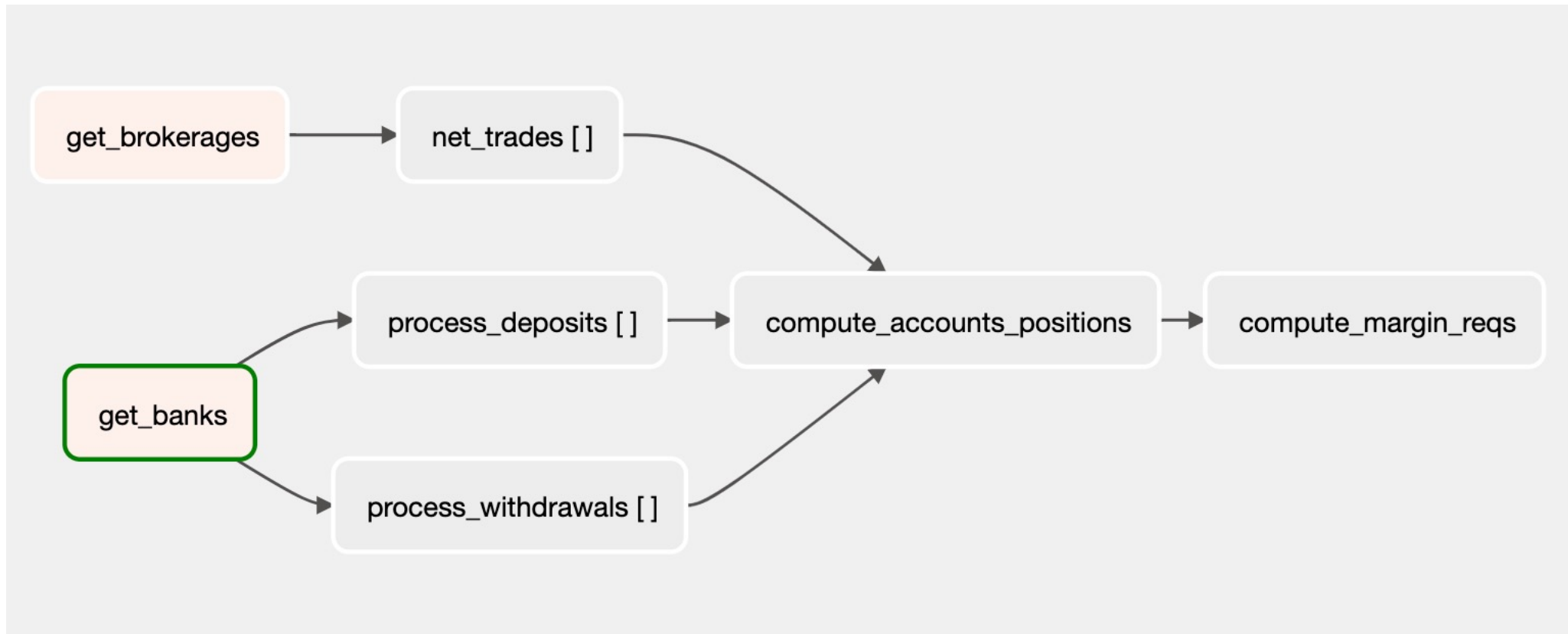
```
>> compute_account_balances
```

```
>> compute_margin_reqs
```

```
)
```

```
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

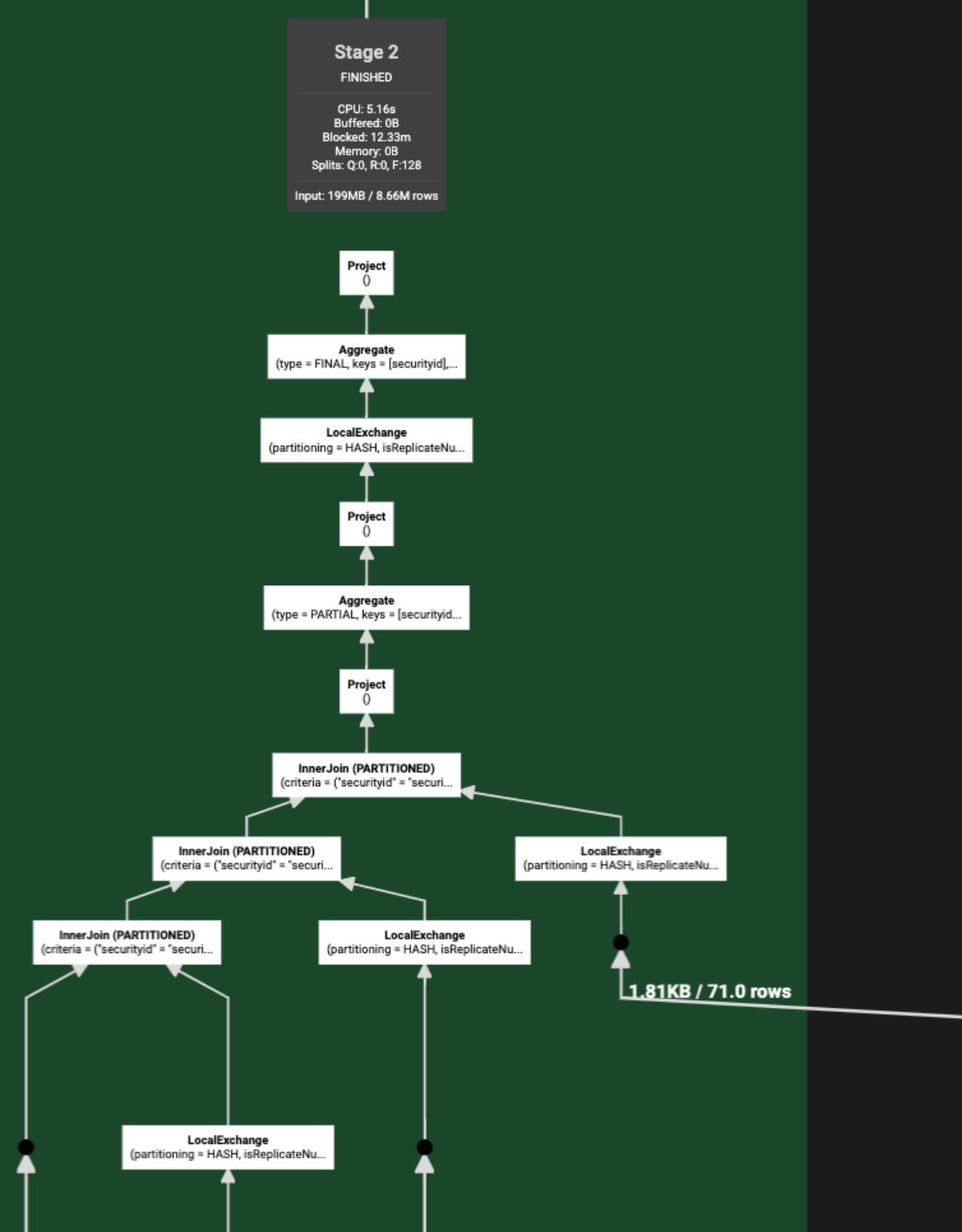


Fault-tolerant execution on Trino

- 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():  
    ...
```

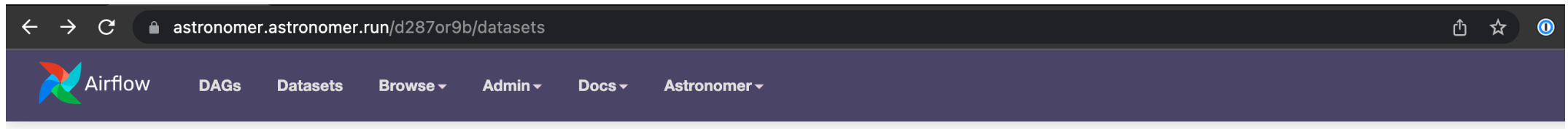
Consuming datasets

```
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

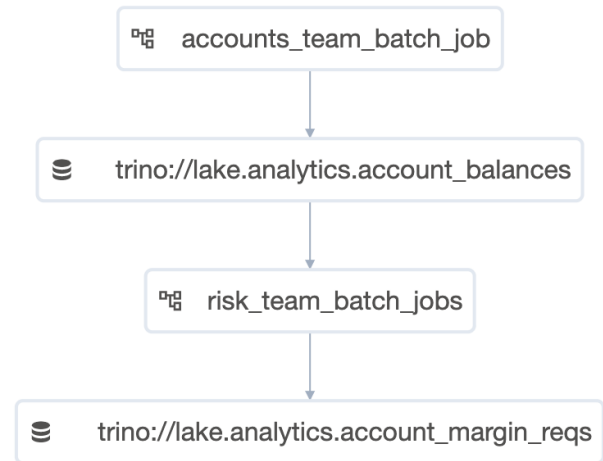
risk_team_batch_jobs me ○ ○ ○ ○ Dataset ⓘ 0 of 1 datasets updated

Datasets view



Datasets

URI ↕	LAST UPDATE ▼
trino://lake.analytics.account_balances Total Updates: 0	
trino://lake.analytics.account_margin_reqs Total Updates: 0	



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.