Democratizing Data with the

Clover Transform Framework

Christopher Hartfield

April 17, 2018

Clover

has an entirely new approach to health insurance.

Meet Clover

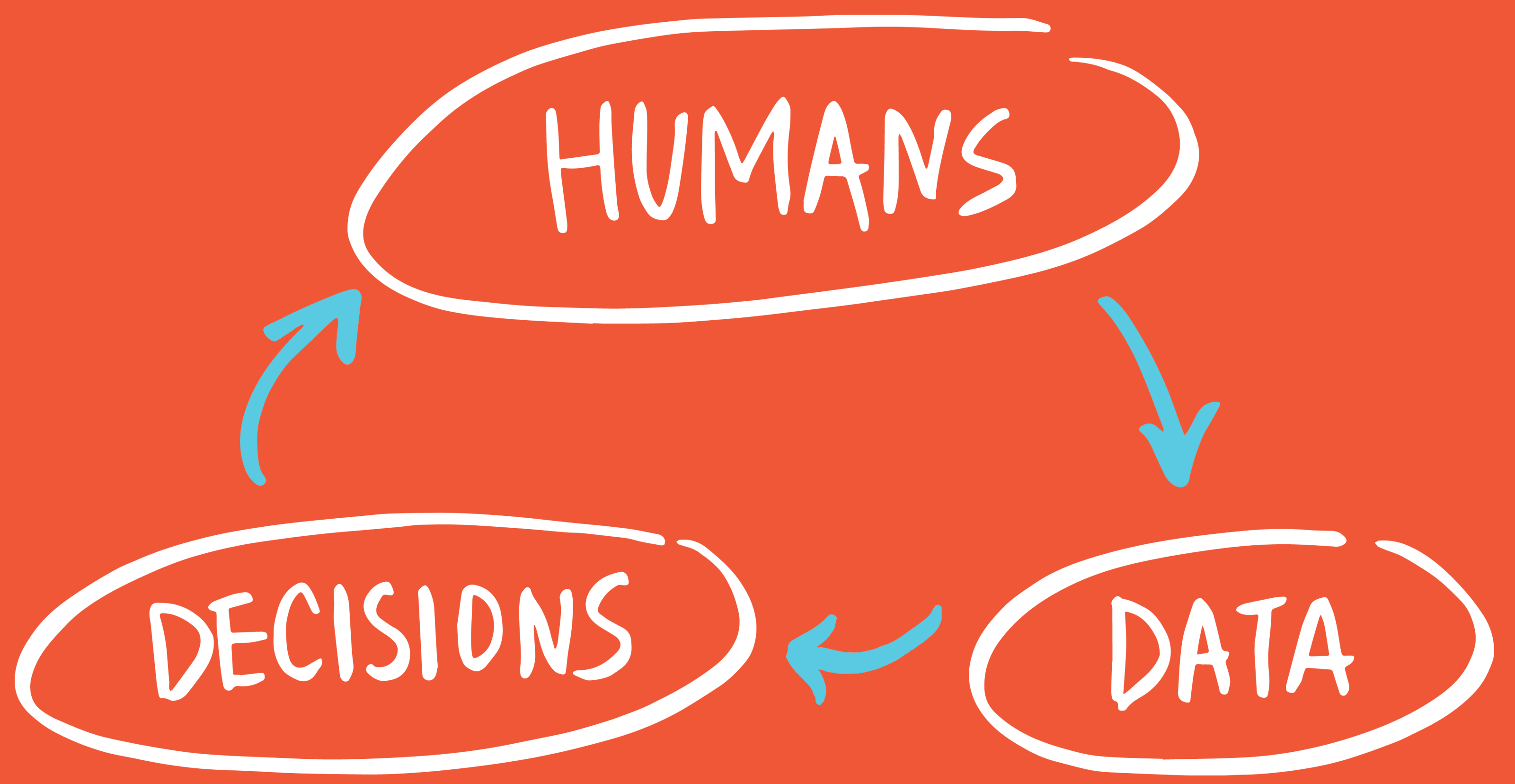
At Clover we're reinventing the health insurance model by integrating technology into every aspect of our members' healthcare.

A little about us....

- A startup Medicare Advantage Payer
- Markets in New Jersey, Pennsylvania, Georgia, and Texas.
- Headquartered in San Francisco
- Venture Backed

SEQUOIA CAPITAL | First Round | SOCIALCAPITAL | WILDCAT VENTURE PARTNERS

How Clover is different from other Medicare Advantage Companies



Clover Leverages Technology and Data to make better decisions

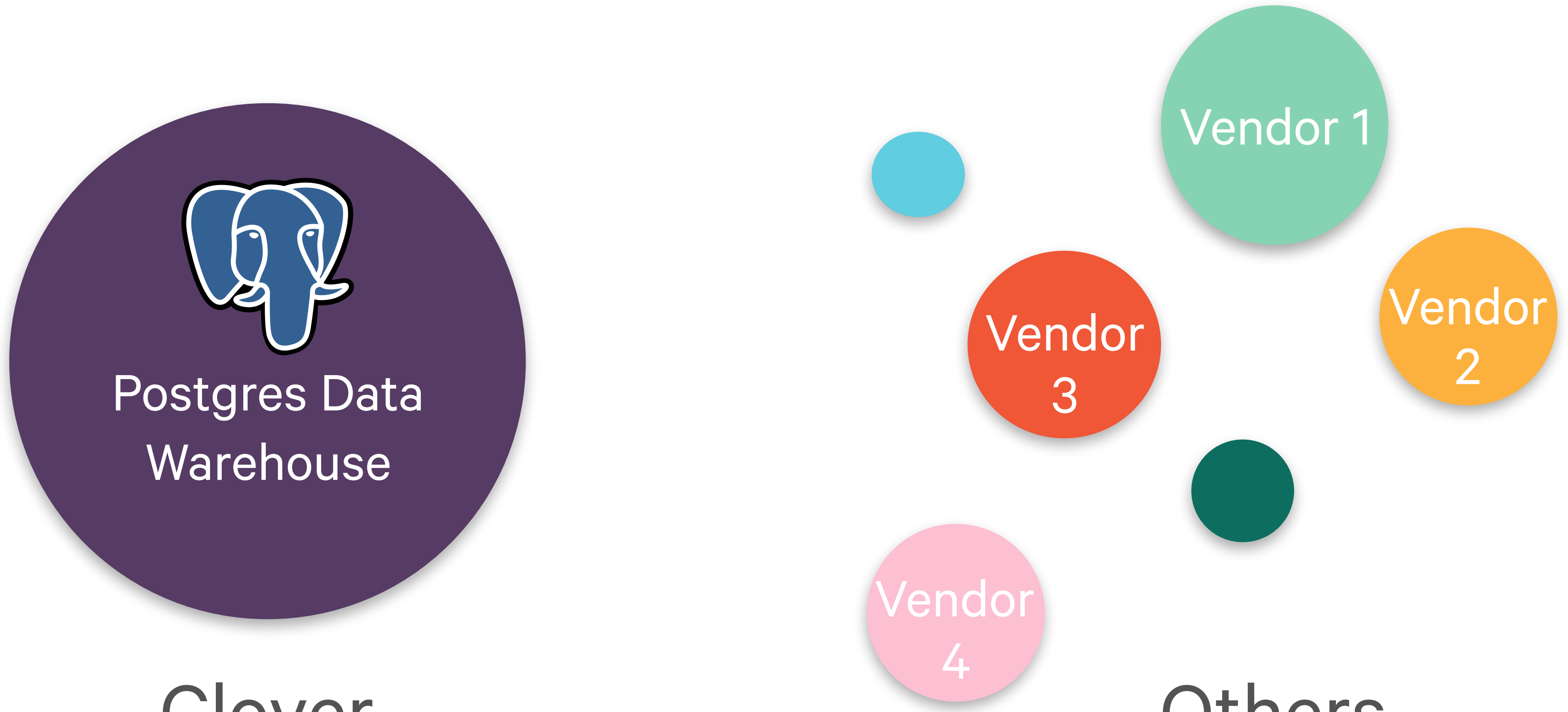
Our data and analytics platform uses continuous, real-time monitoring to create a profile of each of our members' health to help prevent hospital admissions, reduce avoidable spending, and identify and better manage chronic diseases.

Democratizing Data

Most healthcare data today is heavily silo-ed

Most health insurance companies build no software at all

- Data is isolated from one another
- Information is only connected by people, not systems



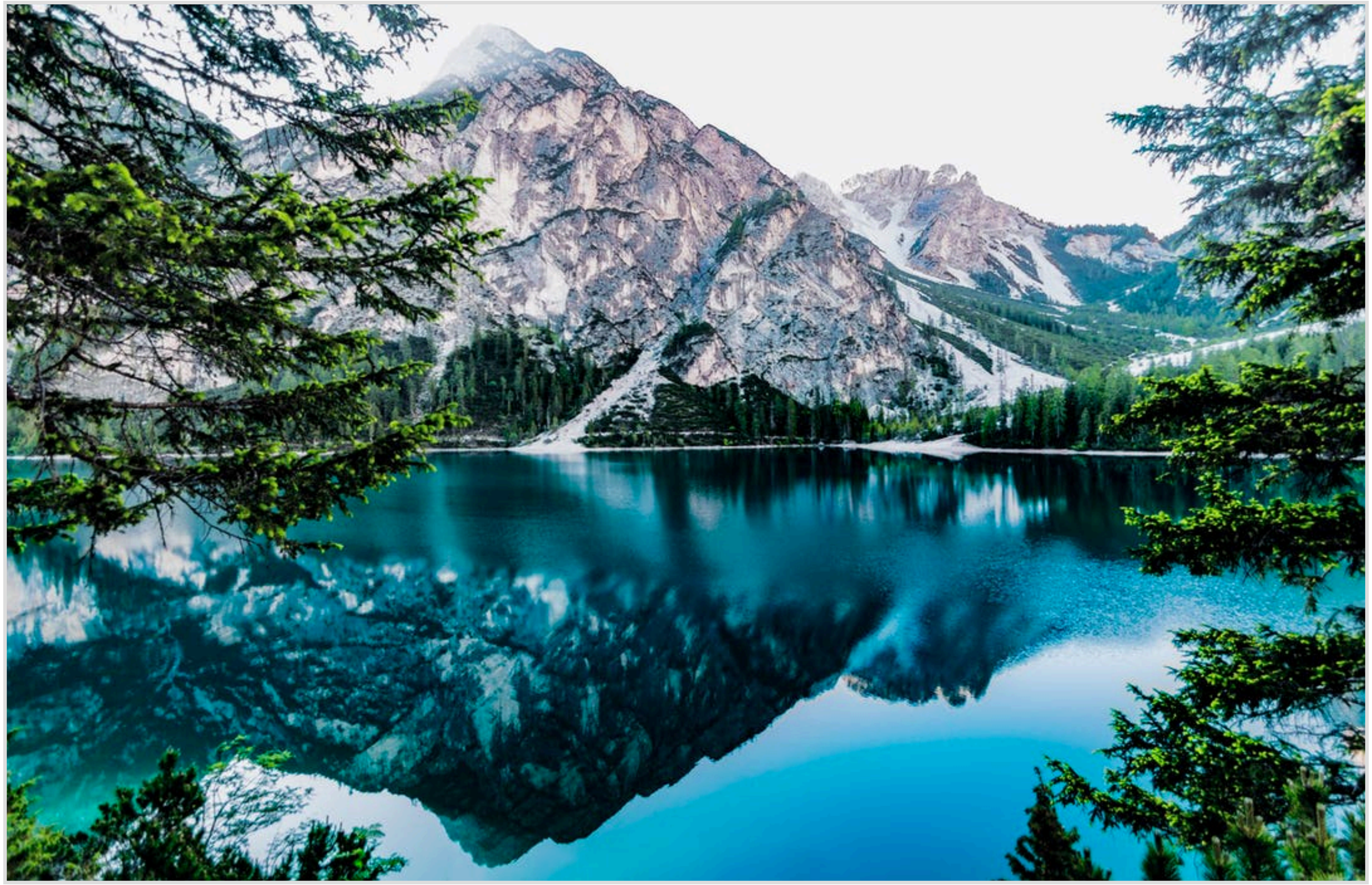
Clover

Others

A Data Lake seems like a good fit

Healthcare Data is often silo-ed. Making connections between disparate data sources is Clover's Mission.

- Many people using many different kinds of data in many different transforms.
- Centrally accessible data will make it easier for people to find data.
- Clover engineers build a lot of pipelines to bring data into our Data Warehouse for DataScience and Operations to use.



Democratizing Data

Clover is unique in that we have a large number of people who manipulate data:

Engineers

Data Scientists

Operations

Analysts

Clover actively trains lots of people how to use SQL and how to build their own transforms of data.

source: Bloomberg

Clover has more than 800 Transforms

What is a Transform?

- Manipulations of Data
- Merging, Filtering, De-dupping, etc. of pieces of data.

Clover does most of these transforms in SQL

- Typically create a new table that has the changes we've made in the SQL
- Some are in Python

```sql
CREATE TABLE claims_paid_after_death AS (
    WITH  dtrr_remove AS (
        SELECT DISTINCT
            personid,
            date_of_death_removed
        FROM stg_analytics.dttr dm
        WHERE date_of_death IS NOT NULL
    ),

    SELECT
        claims.claim_id,
        claims.personid,
        claims.servicing_provider_npi,
        dttr_remove.date_of_death,
        claims.low_service_date,
        claims.high_service_date,
        FROM trg_analytics.medical_claims claims
        JOIN dttr_remove ON d.personid = claims.personid
        WHERE claims.low_service_date > dttr_remove.date_of_death
)
```

Most of our transforms are done in SQL and create new tables as their output.

What were some of the problems we saw?

Wasn't easy for Data Scientists, Analysts, Operations, etc. to add new transforms.

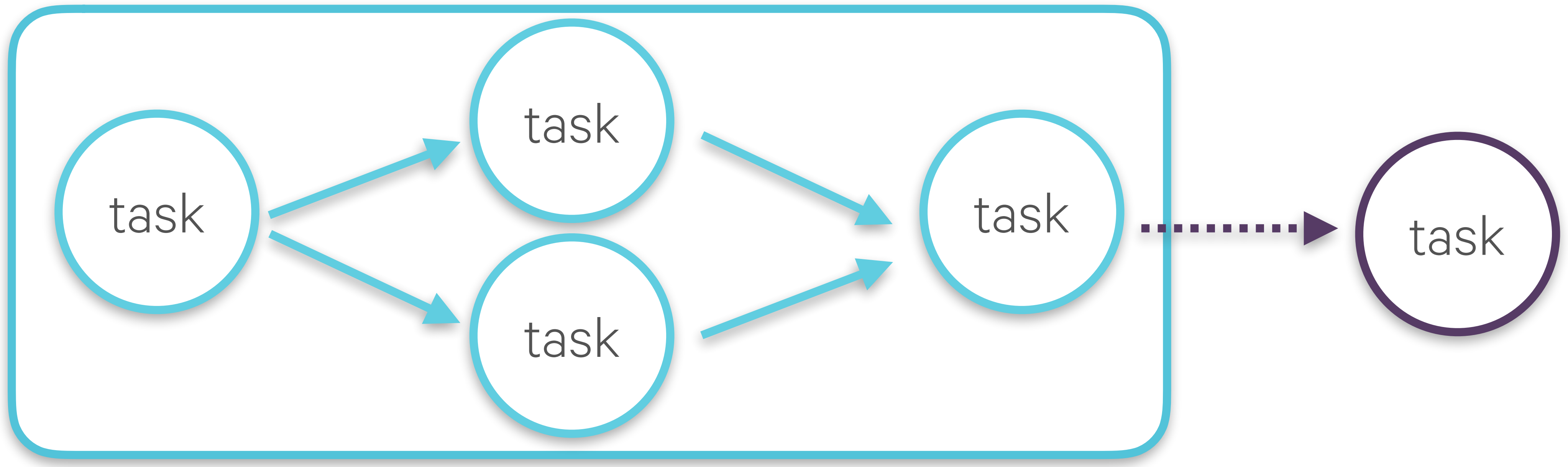
- Almost all of these were creating custom Postgres tables, but doing so in a variety of different ways.
- Some pipelines had custom monitoring, custom transaction handling, etc.
- Not really building pipelines, making a web instead.
- No best practices for testing.

Some pipelines grew to be too big!

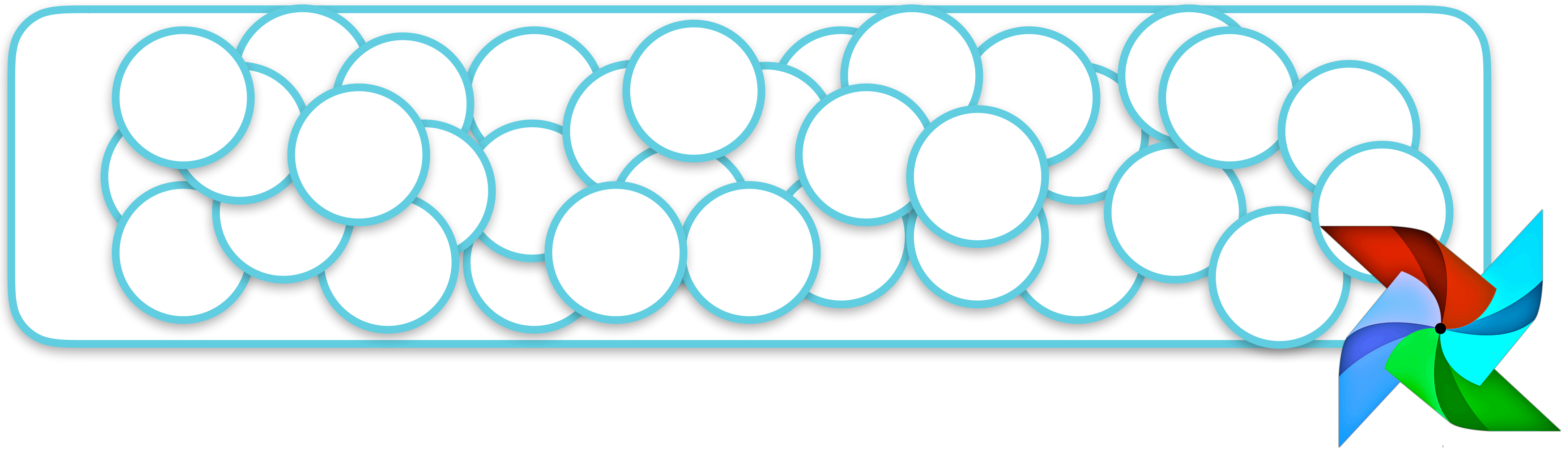
What where some of the problems we saw?

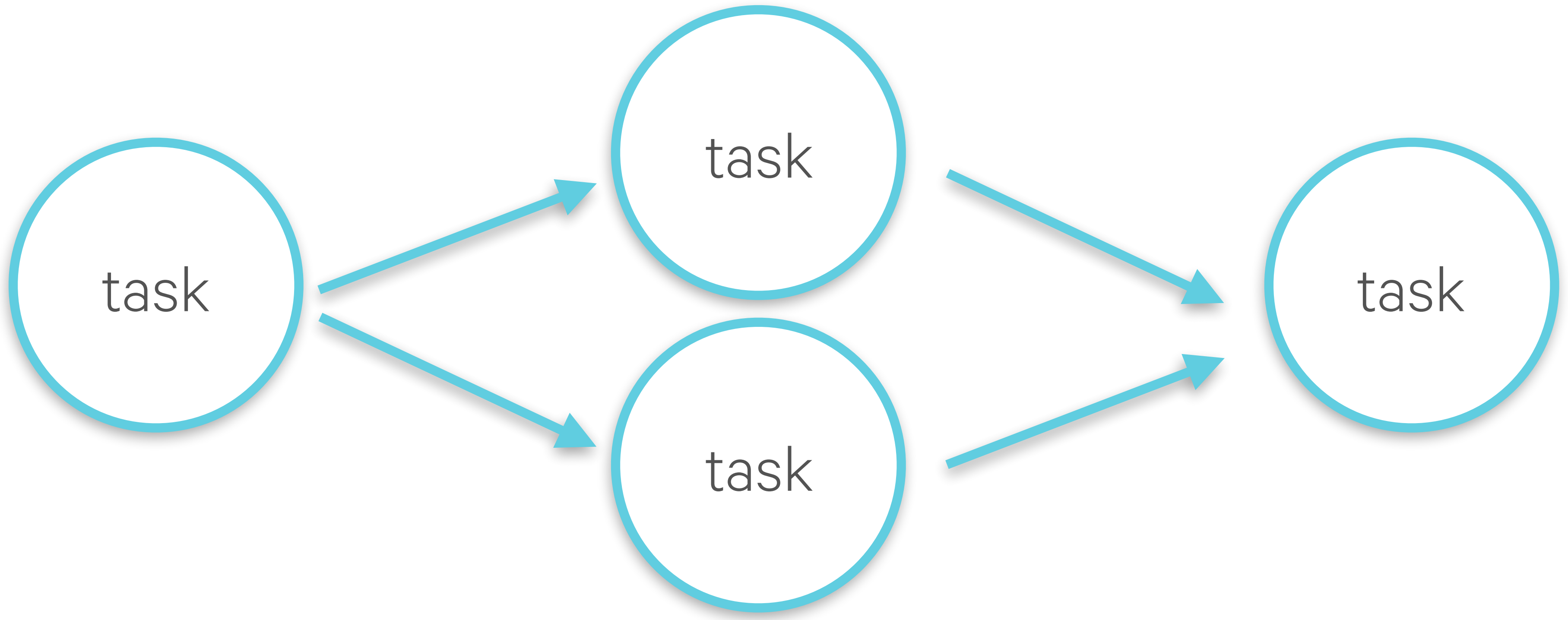
To run your tasks you had to understand Airflow and it was difficult to run the tasks locally.

Can run the full pipeline or a single task



Difficult to run a task and all it’s dependencies

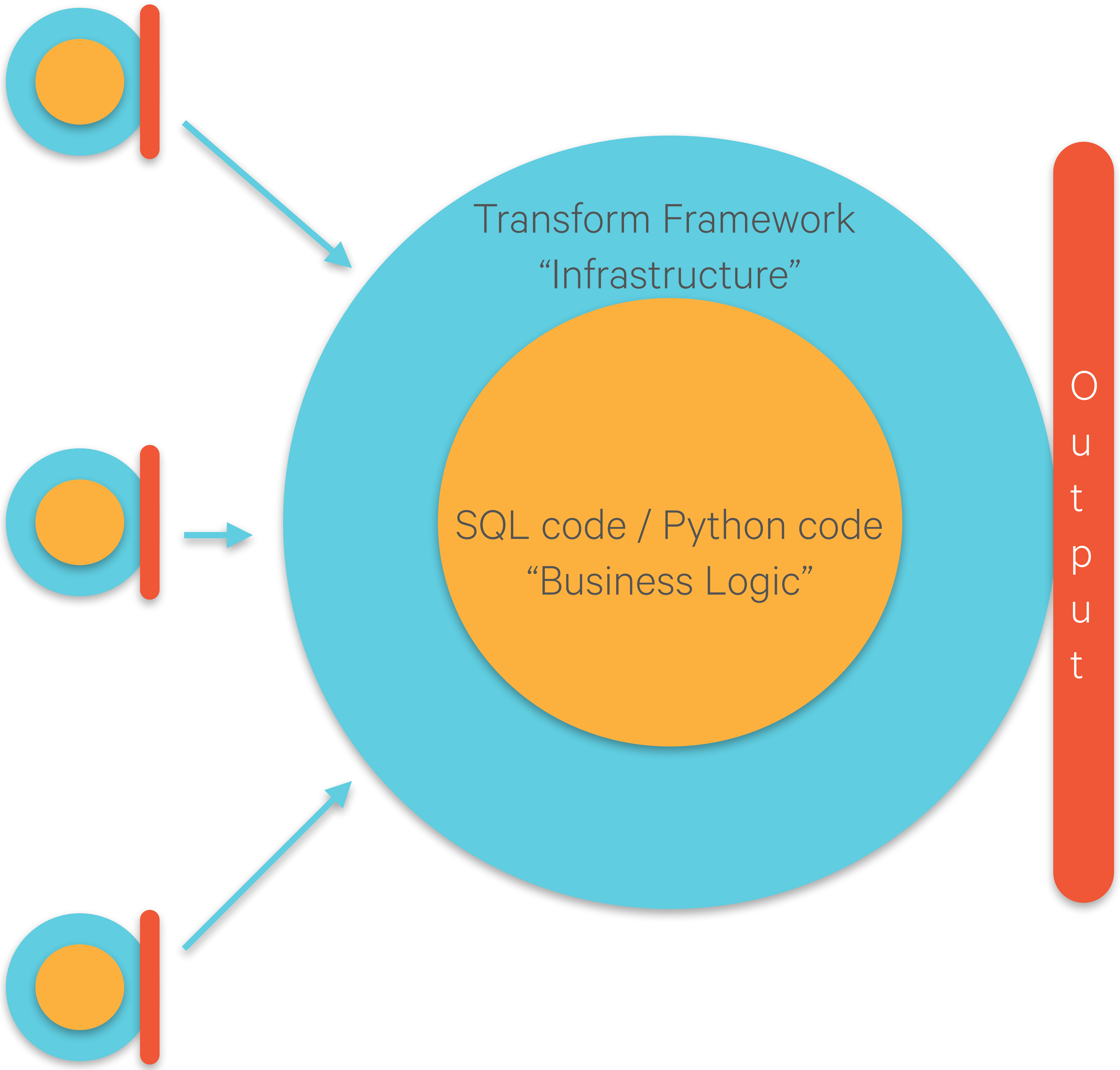




Difficult to run a ‘selection’ of the pipeline

The Clover Transform Framework

The Clover Transform Framework (CTF)



Separating business logic and infrastructure

- Data Scientists and Operations shouldn't have to build monitoring, handle database transactions, build tasks in Airflow, etc.
- Only Define the upstream dependencies.
- Define the output of your transform. Thinking in terms of data outputs instead of just running a task.
- Transform framework is a central place to add monitoring and other features.

So what does this look like to the end user?

```
/*
transform: create_table_as

owners:
  - johnny.appleseed@cloverhealth.com
  - vincent.loves.warriors@cloverhealth.com

doc: |
    This transform constructs the general membership
    table for HEDIS submissions.

inputs:
  - input_transform.sql
  - entry_point_name:external_transform.sql

output:
  name: hedis_submission.general_membership
  doc: The general membership table for HEDIS submissions
  columns:
    - name: memberid
      datatype: text
      required: true
      doc: The ID of the member
    - name: gender
      datatype: text
      required: true
      doc: "M" for male, "F" for female
*/
SELECT * FROM input_transform_table
UNION ALL
SELECT * FROM external_transform_table
```

Transforms are defined by Yaml definitions

- Abstract away creating tables, drop/swapping, index creation, etc. from the end user.
- Documentation built in.
- Define the inputs (either in the same pipeline or an external pipeline)
- No building of Airflow DAGs yourself
- Defines the output
- Owners of the transform!!!

Expanding list of transforms

Different Kinds of Transforms

- **create_table_as** - Create a table from a SELECT SQL statement.
- **upsert** - Insert or update rows from a SELECT SQL statement.
- **sql** - Run raw SQL.
- **python** - Run Python code.
- **load** - Load data into an output (like load an S3 file to the Database)
- **no_op** - Model output but don't run any transformation code.

Upsert:

```
output:
  # create_table_as requires a table output, so the table output
  # definition will go here. View the "Output Guide" section for
  # details on defining table outputs
  name: schema_name.table_name

# The columns to use when matching the new rows with existing rows in the table
upsert_match_columns:
  - column_a

# The columns to update when a match is found; the remaining columns are ignored.
# If no columns are listed, the upsert is append-only and existing rows will not be changed.
upsert_update_columns:
  - column_b

 */
 SELECT * FROM input_tables;
```

Python Transform:

```
"""
transform: python
output:
    type: table
    name: schema_name.table_name
    columns:
        ...
"""
def run(transform, run_ctx):
    # Obtain the wrapped transaction of the "db_uri" run context parameter.
    # Any "db_uri" run context is special in that it has a "wrapped_transaction"
    # attribute. This attribute is a SQLAlchemy connection object already wrapped
    # in a transaction
    transaction = run_ctx['db_uri'].wrapped_transaction
```

What this looks like under the hood

```python
def run(self):
    """Runs the select statement and creates the output table with its results

    This method expects a ``db_uri`` run context to exist for execution
    """

    db_conn = ctf.run_ctx.get('db_uri').wrapped_transaction

    # create the schema if not exists
    self.output.create_schema(db_conn)

    swap_table_name = self.output.table.full_name + '_swap'
    db_conn.execute('DROP TABLE IF EXISTS ' + swap_table_name + ' CASCADE;')

    # create data
    create_query = 'CREATE TABLE {} AS ({});'.format(swap_table_name, self.rendered_sql())
    self.execute_with_explain(db_conn, create_query)

    self._add_extras(swap_table_name, db_conn)

    # validate that the CTAS creates the same table as the
    # output table defined for this transform.
    self.output.validate_structure(table_name=swap_table_name)

    # once validation passes we'll swap the table.
    self._swap(db_conn)

    # finally we analyze
    db_conn.execute('ANALYZE ' + self.output.table.full_name)
```

What happens when the task actually gets run:

- We run explain and log the explain query before running
- Generate the full Create Table As SQL based on the SELECT query in the transform.
- Load data to the table
- Build indexes, constraints, etc.
- Analyzes the table at the end
- Take the returned row_count and log it

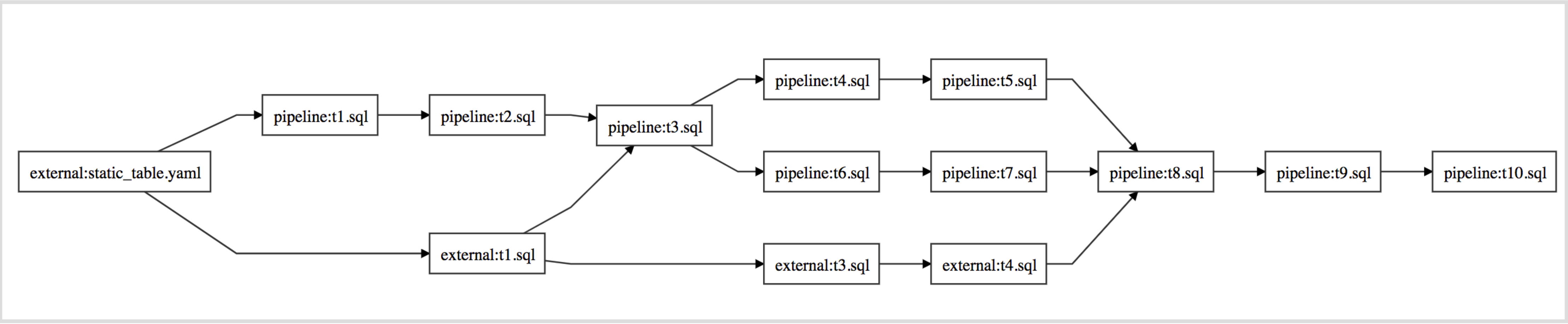
CLI Included

Create, Run, and Visualize transforms locally. Run them in production in Airflow.

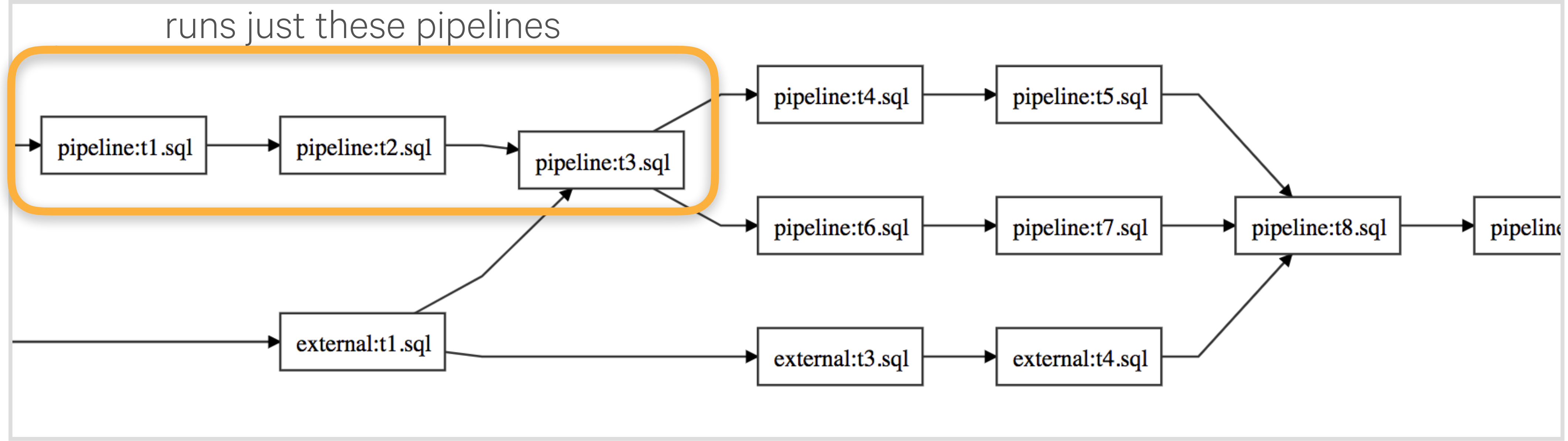
ctf start create_table_as table my_transform.sql

```
/*
transform: create_table_as
owners:
  - chris.hartfield@cloverhealth.com
doc: |
  Write some docs about the transform here
  This is an example of a multi-line YAML string
  that can be used in other "doc" fields
inputs:
  - example_input_transform.sql
  - entry_point_name:example_external_transform.sql
output:
  name: my_schema.addresses
  doc: Write some docs about the table here. Columns have a "doc" field for column-level docs
  columns:
    - name: id
      datatype: uuid
      required: true
    - name: street_1
      datatype: text
    - name: street_2
      datatype: text
    - name: city
      datatype: text
    - name: state
      datatype: text
    - name: zipcode
      datatype: text
  primary_keys:
    - id
*/
-- This is a create_table_as transform. Any SQL written here will be executed
-- within a CREATE TABLE AS statement. The output table definition above
-- will be used to construct the table in the CREATE TABLE AS statement.
--
-- Your SQL must create the table defined in the
-- transform output or else this transform will fail when executed
```

ctf ls pipeline

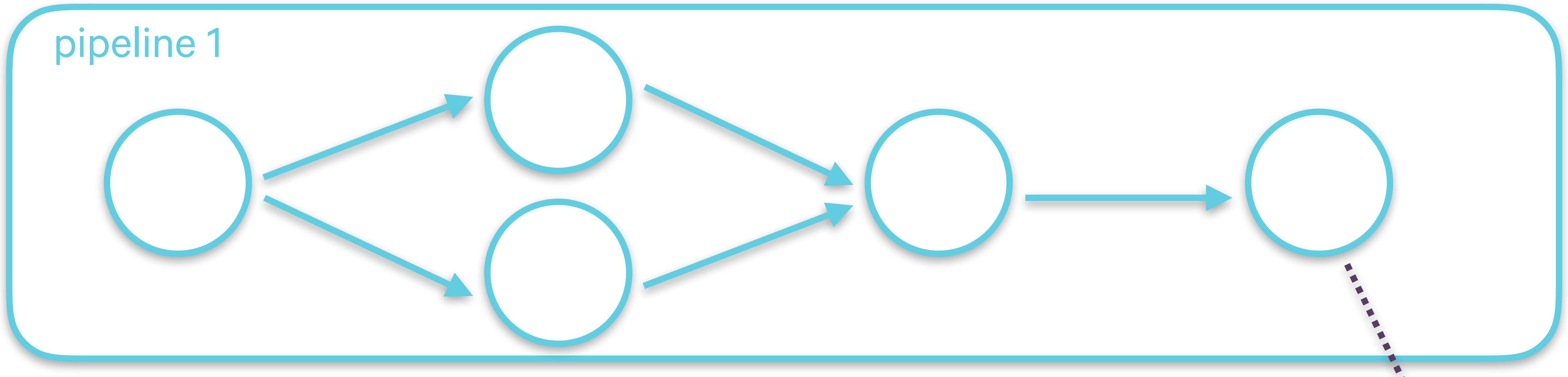


ctf run pipeline -s t1.sql -e t3.sql

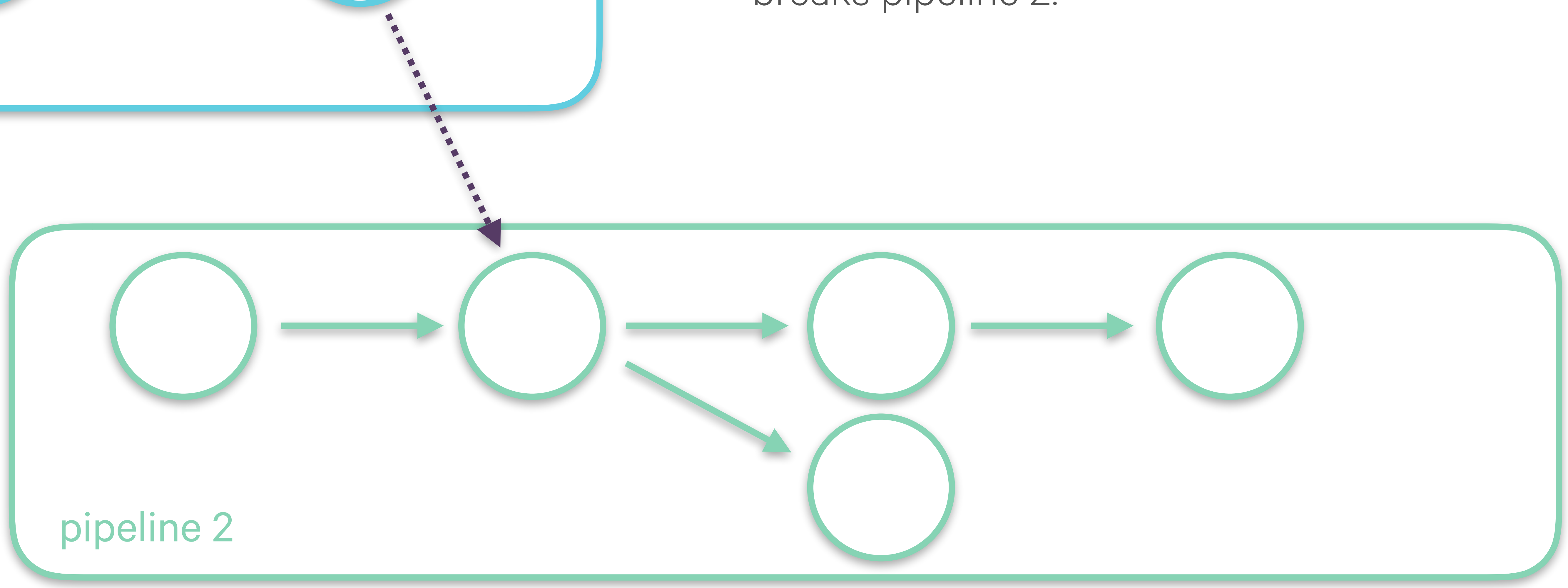


The importance of defining all your Inputs and Outputs

A transform must define all it’s inputs from both internal and external pipelines



Integration tests are in place that will catch when the output in pipeline1 changes and breaks pipeline 2.



Must define all the tables or files that you use in your transform, avoids implicit dependencies.

Can create restrictions on what tables you can actually use in downstream transforms and enforce it.

Testing Infrastructure

Defining the outputs makes testing robust

- Easily get an empty table of an upstream transform.
- Helper functions to create test data.
- One clear and obvious way to test your transforms.
- Structural tests automatically run as well.

```python
def test_t3(transacted_postgresql_db):
    """Tests the SQL of the t3 task using pgmock"""
    t3_sql = ctf.testing.get_transform('pipeline', 't3.sql').sql

    # Apply table patches to both input tables (including the external input).
    # get_pgmock_patch will build a patch for the table in the the transform based on
    # the table name and the column definitions
    t2_table_patch = ctf.testing.get_output('pipeline', 't2.sql').get_pgmock_patch([{
        'col1': 4,
        'col2': 5,
        'col3': 6
    }, {
        'col1': 40,
        'col2': 50,
        'col3': 60
    }])
    ext_table_patch = ctf.testing.get_output('external', 't1.sql').get_pgmock_patch([{
        'col1': 10,
        'col2': 20,
        'col3': 30
    }])

    # Obtain patched SQL to execute
    test_sql = pgmock.sql(t3_sql, t2_table_patch, ext_table_patch)
    results = list(transacted_postgresql_db.connection.execute(test_sql))
    assert results == [(40, 50, 60), (10, 20, 30)]
```

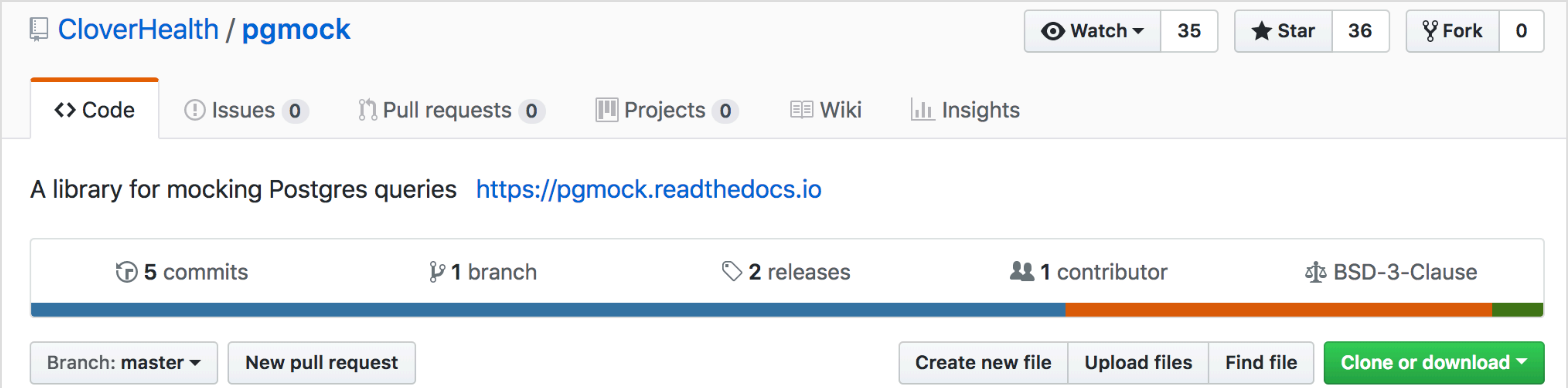
More Testing Infrastructure

pgmock

- Allows for testing individual subqueries and CTEs within SQL.
- Great for testing pieces of large sql queries.
- Open Sourced 😀

```
query = "SELECT sub.c1, sub.c2 FROM (SELECT c1, c2 FROM test_table WHERE c1 = 'hi!') sub;"

# Patch the "sub" subquery with the rows as the return value
patch = pgmock.patch(pgmock.subquery('sub'), rows=rows, cols=['c1', 'c2'])

# Apply the patch to the full query
patched = pgmock.sql(query, patch)

print(patched)
"SELECT sub.c1, sub.c2 from (VALUES ('hi!','val1'),('hello!','val2'),('hi!','val3')) AS sub(c1,c2);"
```

pgmock - https://github.com/CloverHealth/pgmock



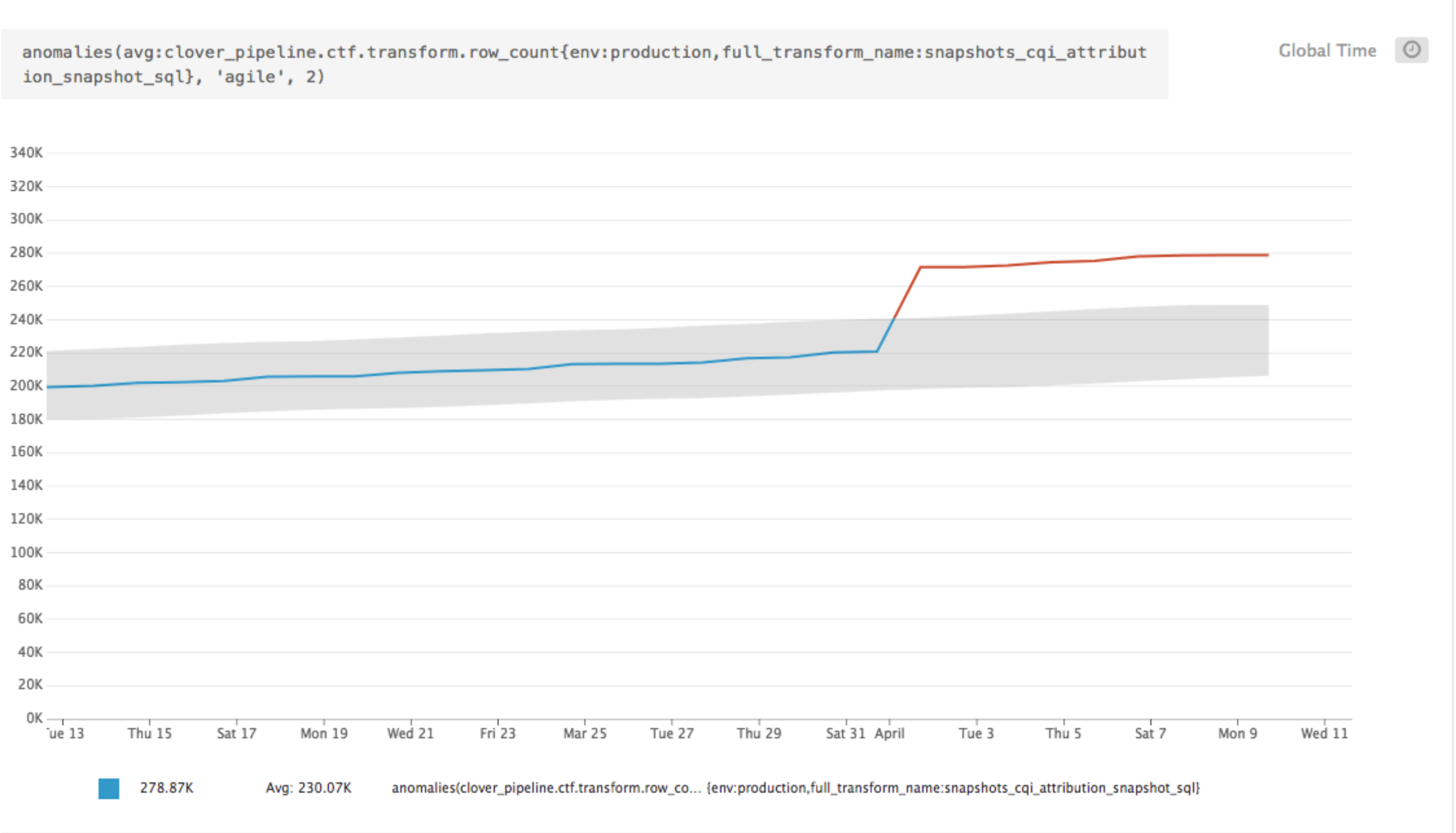
Extending the Framework

Monitoring

```
/*
transform: create_table_as

owners:
  - owner_email@addresses.com

doc: Documentation about this transform

# Validate the output table by running a validation query
validation_queries: |
  SQL STATEMENT FOR VALIDATION

inputs:
  - input_task1.sql
  - external:input_task2.sql

output:
  # create_table_as requires a table output, so the table output
  # definition will go here. View the "Output Guide" section for
  # details on defining table outputs
  name: schema_name.table_name

data_quality_dimensions:
  completeness:
    - col1
  currency:
    - col1
    - col2
*/
SELECT * FROM input_tables;
```

Monitoring can be defined in the transform yaml

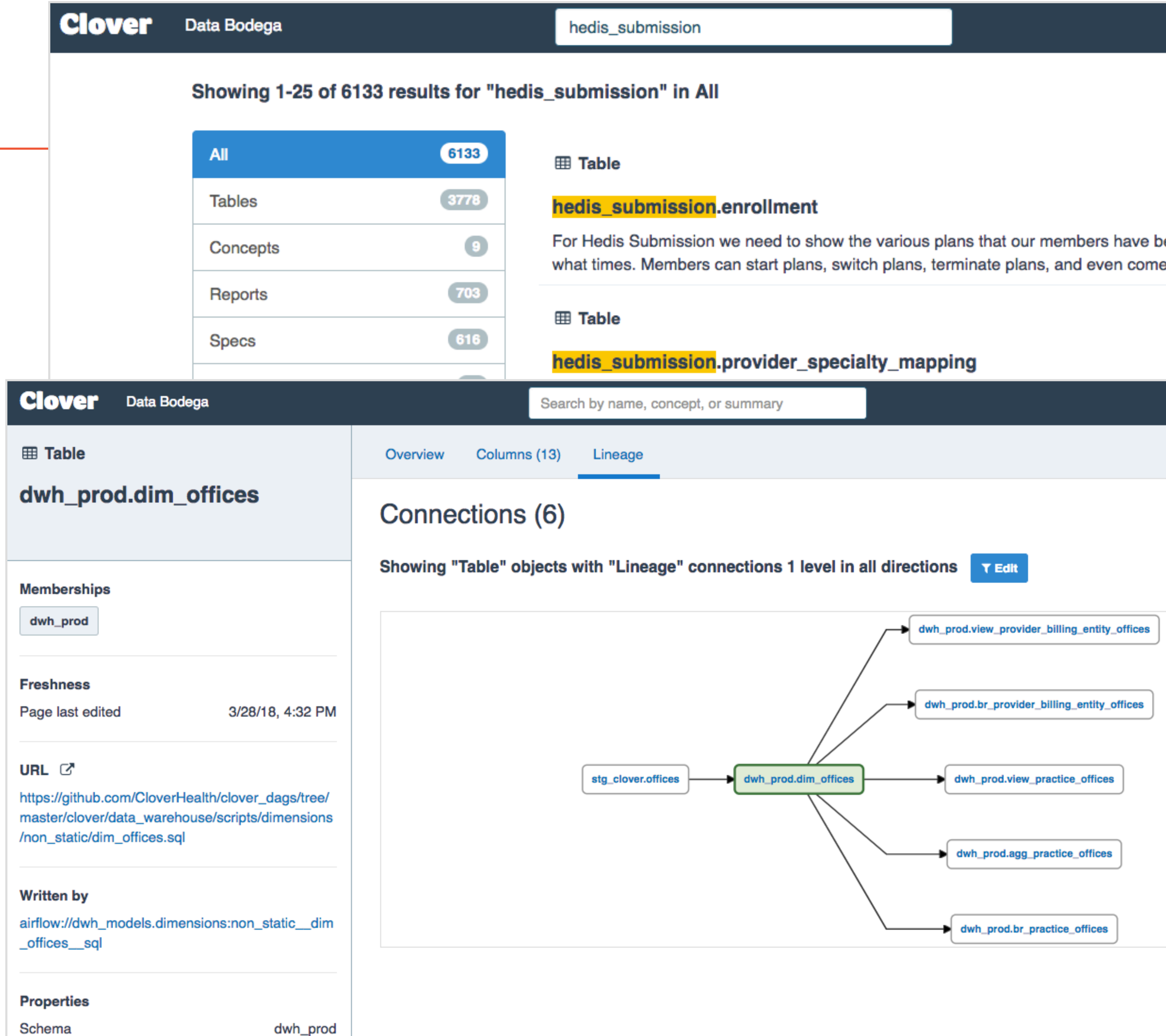
All metrics (including row counts) are sent to DataDog. Can use anomaly detection to check for data issues.



Data Bodega

With > 800 Transforms discoverability becomes a problem

- Data Bodega gives us a place to document data products, tables, and reports.
- Lineage of the data between different tables.
- Includes ModeReports so we can see how people are querying the tables created.



Machine Learning

Expanded CTF to handle our Machine Learning infrastructure

- Handles the Machine learning infrastructure in the background.
- Can split datasets into train, test, and validation allocations.
- Can run most of the scikit learn algorithms.
- All defined in yaml, no python to write.
- More accessible to a wide range of Analysts and Data Scientists.

```yaml
transform: ml_model

inputs:
  - datasets_entry_point:example_dataset.yaml

output:
  name: example_model

run_params:
  threads: 4
  dataset_splits:
    method: 'split_randomly_by_index'
    params:
      start: 2016-01-01
      end: 2018-01-01
      train_allocation: 60
      validate_allocation: 20
      test_allocation: 20
      split_by_index: 'personid'
  outcome_feature_names: [outcome_feature_name_a, outcome_feature_name_b]

algos_to_run:
  - name: model_name_a
    algo: GradientBoostedTreeClassifier
    fitting_params:
       max_depth: 3
    model_data_transforms:
      - method: remove_constant_features
      - method: select_features_k_best
        params:
          k: 10
  - name: model_name_b
    algo: LogisticRegression
    model_data_transforms:
      - method: remove_constant_features
```

Questions?

Clover

is hiring Engineers and Data Scientists!

Solve one of the country's toughest problems

Join a team that values diversity

Work in a passionate environment

Interested in joining Clover?

Come see me in Office Hours

cloverhealth.com/careers

Find anyone with a Clover badge