Enabling SQL testing in Airflow workflows using Pydantic types

3.0

Gurmeet Singh Saran, Anthropic
Kushal Thakkar, Anthropic

Agenda

1. The Problem
2. SQL Testing Framework
3. Culture Shift
4. Future

The Problem

3.0

The $80 Billion Question

September 2016: Facebook's Admission

Wrong video metrics reported to advertisers for two years!

- Average viewing time overstated by 80%
- Affected billions in advertising decisions
- Discovered after TWO YEARS in production

Source: WSJ

It's Not Just Facebook

Monday Morning 😰

"Why are yesterday's revenue numbers different today?" *-- Someone's JOIN is -- creating duplicates*

Wednesday Afternoon 🤬

"The exec dashboard is wrong. AGAIN." *-- That 'quick fix' broke -- three downstream queries*

Friday 5 PM 💀

"We just billed customers wrong. All hands on deck." *-- Missing WHERE clause -- in the billing calculation*

The Truth: Every team has their own Facebook moment brewing

The Hidden Complexity of SQL

```sql
WITH fruits AS (
  SELECT 'apple'  AS fruit, 3.55 AS price
  UNION ALL
  SELECT 'banana', 2.10
  UNION ALL
  SELECT 🔥, 4.30
)
SELECT MAX_BY(fruit, price) AS fruit
FROM fruits;
```

What is the output of this SQL Query ?

Depends if 🔥 is null or not null

apple - for clickhouse, duckdb

null - for athena, bigquery and snowflake

* Athena Breaking Changes Across Versions - https://docs.aws.amazon.com/athena/latest/ug/engine-versions-reference-0003.html#engine-versions-reference-0003-breaking-changes

The Uncomfortable Truth

We Don't Have Time

- Testing seen as overhead
- Pressure to ship fast
- Technical debt accumulation

SQL is Simple

- Underestimating complexity
- No testing framework
- Trust without verification

Most semantic SQL changes are manually tested at best, relying on production alerts to catch failures

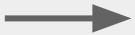
SQL Testing Framework 3.0

Design Principles

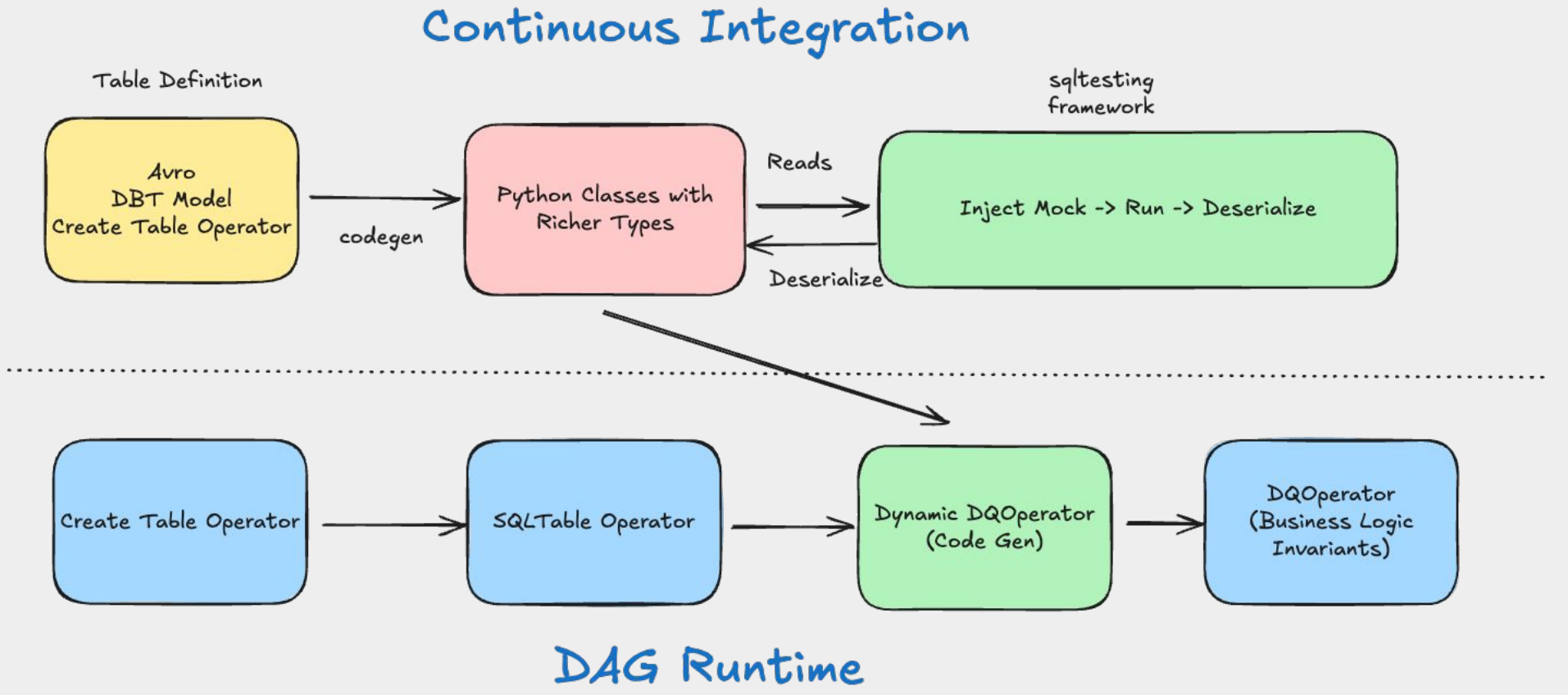
- **Zero or Minimal Footprint:** Tests should avoid creating any artifacts whenever possible.
- **Ease of Use & Extensibility:** Writing and maintaining test cases should be as simple as writing SQL.
- **Dynamic & Adaptive Testing:** Instead of relying solely on predefined test cases, our library should have the ability to automatically surface new issues as data evolves.

The Magic - CTE Injection

```python
def test_simple_user_query():
    @sql_test(
        mock_tables=[
            UsersMockTable([User(1, "Alice",'alice@fb.com')]),
            UsersMockTable([User(2, "Bob",'bob@fb.com')])
        ],
        result_class=User,
    )
    def test_user_query():
        return TestCase(
            query="SELECT * FROM users WHERE user_id = 1"
        )
    results = test_user_query()
    assert len(results) == 1
    assert results[0].name == "Alice"
    assert results[0].user_id == 1
```



```sql
WITH users AS (
    -- Injected mock data
    SELECT * FROM (
        VALUES
        (1, 'Alice', 'alice@fb.com'),
        (2, 'Bob', 'bob@fb.com')
    ) AS t(user_id, name, email)
)
-- Your original query
SELECT * FROM users WHERE user_id = 1
```


Continuous Integration
Table Definition
Avro
DBT Model
Create Table Operator
codegen
Python Classes with
Richer Types
Reads
Deserialize
sqltesting
framework
Inject Mock -> Run -> Deserialize
Create Table Operator
SQLTable Operator
Dynamic DQOperator
(Code Gen)
DQOperator
(Business Logic
Invariants)
DAG Runtime

Where does it fit?

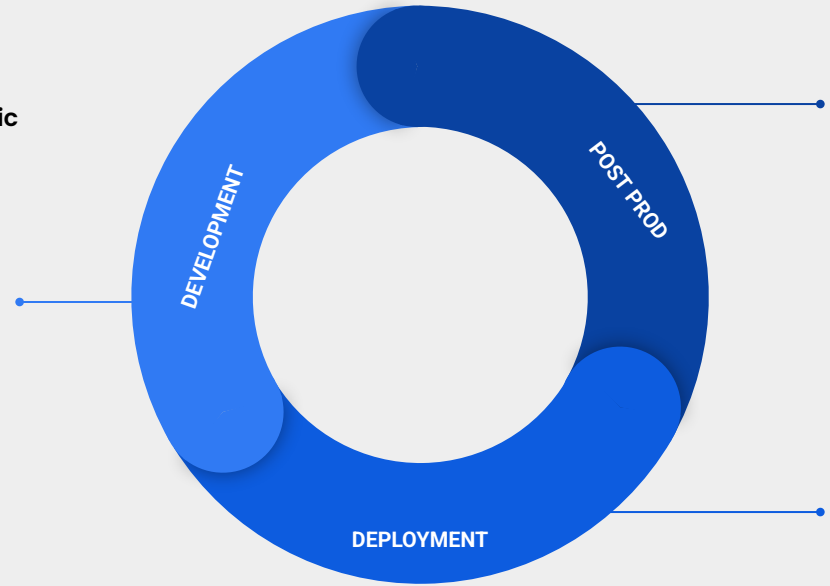
🧪 SQL Testing - Will my logic work?

✅ Test business logic

✅ Validate calculations

✅ Catch bugs early

BEFORE you deploy



📊 DQ Monitors - Is my data healthy?

✅ Monitor anomalies

✅ Track freshness

✅ Alert on issues

AFTER it's running

🔧 DQ Operator/DBT Tests - Is my model sound?

✅ Test structure

✅ Check constraints

✅ Verify relationships

AS you transform

Culture Shift 3.0

The Elephant in the Room

But Writing Tests Takes Too Much Time!

The Perceived Cost 😰

- Writing test: 30 minutes
- Maintaining test: 10 minutes/month
- Running tests: 5 minutes

“We don't have time for this!"

The Hidden Cost of NOT Testing 💸

One production bug can take **up to 120 engineer hours per incident**. Plus, **lost revenue**, **customer trust**, **team burnouts**.

Math below:

- Detection: 2-48 hours (it's Friday night)
- War room: 8 engineers × 6 hours = 48 hours
- Fix & deploy: 4 hours
- Data cleanup: 16 hours
- Post-mortem: 8 hours

But How Do We Actually Get There?

Month 1

Phase 1

Start Where It Hurts

Test only the broken queries

```
if query in ["revenue_calc", "user_metrics", "that_evil_join"]:

    write_test()  # Just these 3 queries
```

- Immediate value
- Team sees immediate benefits
- No overwhelming commitment

But How Do We Actually Get There?

Month 2

Phase 2

New Code Rule

All new queries must have test

```
if query.is_new():

    require_test()  # Going forward, not backward
```

- No technical debt increase
- Gradual coverage growth
- Developers learn by doing

But How Do We Actually Get There?

Month 3 - 6

Phase 3

The Boy Scout Rule

when you touch it, test it

```
if query.is_modified():

    add_or_update_test()  # Leave it better
```

- Organic coverage increase
- Tests stay relevant
- Knowledge spreads naturally

But How Do We Actually Get There?

Month 6 - 12

Phase 4

Full Coverage Sprint

Dedicated Effort for critical path

```
for query in critical_business_queries:
    backfill_test()  # Systematic coverage
```

- Risk-based prioritization
- Measurable progress
- Celebrate milestones

Making It Stick

The Transformation Journey

- **Skepticism** 😒 "This is just more process"
  - Action: Show, don't tell. Live demo of catching a real bug.
- **Curiosity** 🤔 "Okay, that actually would have saved us last month"
  - Action: Pair with skeptics on their first test.
- **Early Adoption** 😊 "I wrote a test and it caught something!"
  - Action: Celebrate publicly. Share success in stand-up.
- **Momentum** 🚀 "Can we test our ETL pipeline too?"
  - Action: Expand scope. Provide advanced training.
- **New Normal** 💪 "PR without tests? That's weird."
  - Action: It's now just how we work.

Future

3.0

Vision

- AI-powered test generation
    - Claude is really good at test case generation!
- Multi-cloud testing
- Auto generate DQ checks based on richer data types
- Query engine migration testing
- Perf evaluations - not just correctness

Open Source

Project: **sqltesting**

A powerful Python framework for unit testing SQL queries with mock data injection across BigQuery, Snowflake, Athena, Trino, Redshift, and DuckDB.

Project: **mocksmith**

Type-safe data validation with automatic mock generation for Python dataclasses and Pydantic models. Build robust data models with database-aware validation and generate realistic test data with a single decorator.

Questions?