#FABCONSQLCON2026

FABCON
Microsoft Fabric
COMMUNITY CONFERENCE

SQLCON
Microsoft SQL
COMMUNITY CONFERENCE

ATLANTA MARCH 16 - 20, 2026

Config-driven Data Engineering

in Microsoft Fabric

Pierre LaFromboise

Chief Data Officer
Covenant Technology Partners

25+ years implementing Microsoft data solutions
Organizations of all sizes · Every industry

A Familiar Journey

Dataflows + Warehouse

Power BI-familiar, low-code

Great starting point

Gaps in transformation capabilities

SQL-familiar, approachable

Rigid for bronze / silver layers

Ecosystem still maturing

pivot

Notebooks + Lakehouse

Full PySpark flexibility

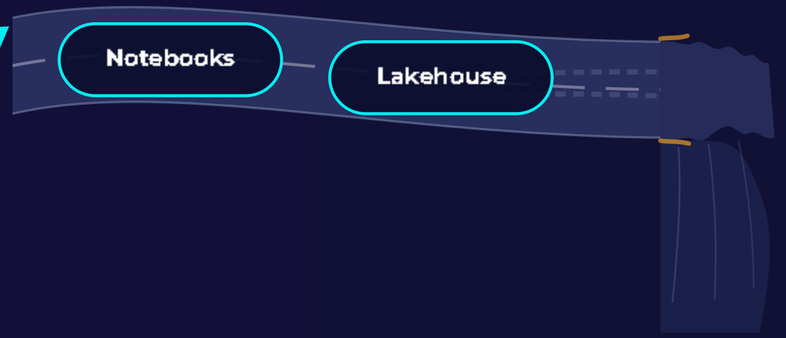
Natural next step from Dataflows

Steeper learning curve

Medallion-native

Delta Lake out of box

Discipline required to scale



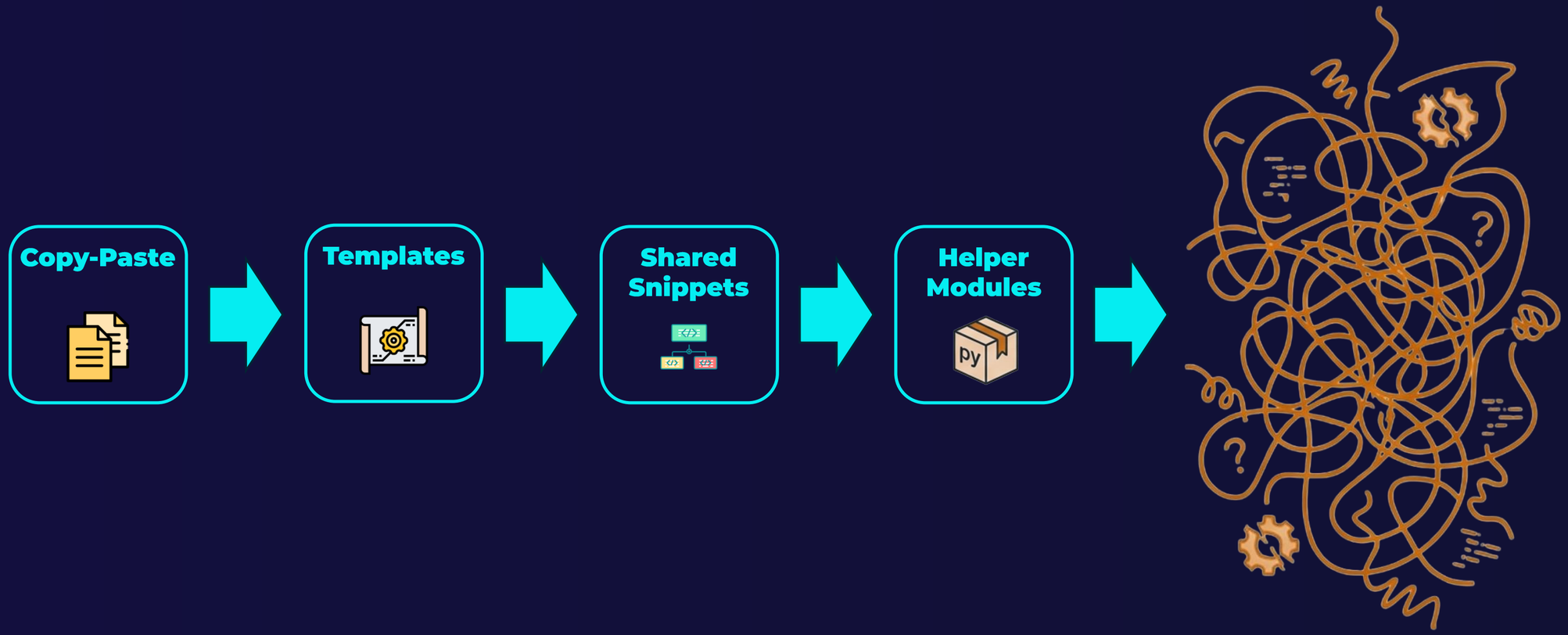
It Starts Well

The first notebook works.

The second notebook works.

The thirtieth notebook is... *complicated.*

The Abstraction Treadmill



Five Problems, One Root Cause

1. No Repeatable Structure
2. Copy-Paste Proliferation
3. Brittle Orchestration
4. Opaque Execution
5. Environment Drift

Notebooks have no contract.

There is no enforceable agreement between what was intended and what was built.

The Structural Problem

PySpark is pro-code by design.

Any governance layer is optional.

A developer can always just write a notebook cell.

You cannot enforce a standard you cannot intercept.

This isn’t a people problem. It’s an architecture problem.

What if the pipeline IS the config?

Declare what you want.
Let the engine handle how.

weevr

A configuration-driven execution framework for PySpark in Microsoft Fabric.

Open source · Apache 2.0 Licensed · Production ready

ardent-data.github.io/weevr

The Core Promise

Same config.

Same inputs.

Same outputs.

Every time.

The Configuration IS the Contract

« The config is the specification

« The config is the standard.

« The config is the audit trail.

« The config is the truth.

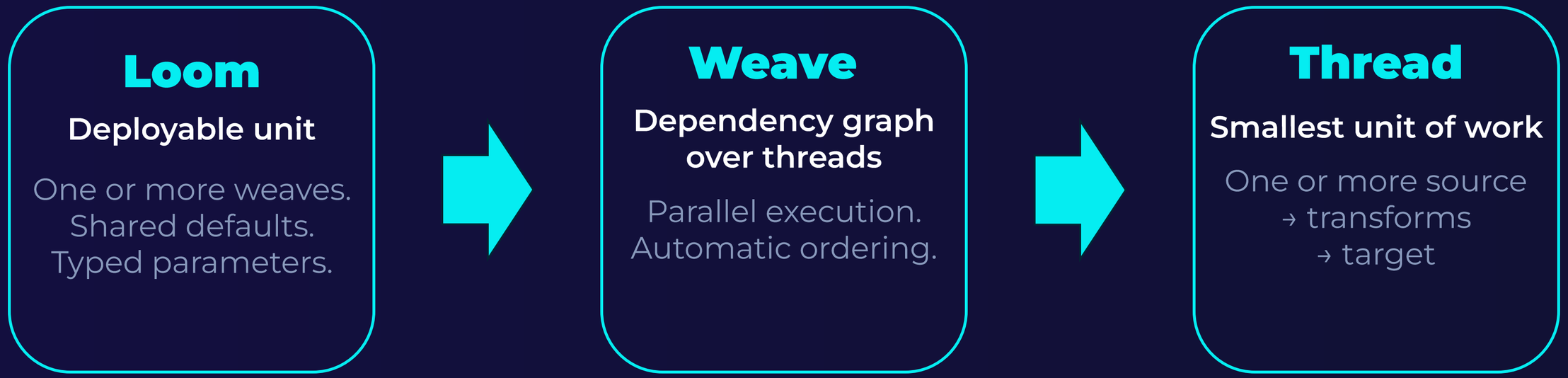
Not a comment in a notebook.

Not a wiki page.

Not a convention.

The config itself.

The Object Model



Configuration flows down. Most specific wins.

Architectural Foundation

Spark-native

No new execution engine. weevr runs on the Spark session you already have.

No Code Generation

Configuration is interpreted at runtime. Nothing is generated, compiled, or stored between runs.

Fabric & Delta Aligned

Built for OneLake, lakehouses, and Delta tables. No abstraction leaks. No workarounds. No surprises.

Deterministic Execution

Same config plus same inputs equals same outputs. The engine is stateless. The behavior is guaranteed.

What weevr Is Not

X Not a replacement for Spark

Spark is the execution engine. weevr is the configuration layer that sits on top of it.

X Not a scheduler

Orchestration remains external — Fabric Pipelines, Airflow, whatever your team already uses.

X Not a code generator

Configuration is interpreted at runtime. No PySpark is generated, stored, or compiled from YAML.

X Not opinionated about data modeling

Kimball, Data Vault, wide tables — your modeling patterns are supported, not mandated.

X Not trying to solve every Fabric problem

Deliberately scoped; PySpark, Fabric, Delta Lake. That's the lane. It's a wide lane, but it's one lane.

Features at a Glance

1. Declarative YAML Pipelines
2. 19 Transform Types
3. DAG Orchestration
4. Incremental Loading
5. Key Generation & Hashing
6. Idempotent Outputs
7. Structured Telemetry & Observability
8. Validations & Assertions

Each feature is a direct answer to a named problem.

Declarative YAML Pipelines

Problems addressed:

1. No Repeatable Structure
2. Copy-Paste Proliferation

Define sources, transforms, and targets in YAML.
One canonical definition per pipeline.
Reviewable, diffable, version-controlled in Git.

The config is the PR.

stg_customers.thread · 14 lines

```yaml
sources:
  customers:
    type: delta
    alias: raw.customers
steps:
  - filter:
      expr: "status = 'active'"
  - derive:
      columns:
        full_name: "concat(first_name, ' ', last_name)"
target:
  alias: silver.dim_customer
write:
  mode: overwrite
```

19 Transform Types

Problems addressed:

1. No Repeatable Structure
2. Copy-Paste Proliferation

The full range of data engineering needs, declared in config — not coded from scratch.

Every transform follows the same declarative pattern.

No custom code. No variation. No drift.

DAG Orchestration

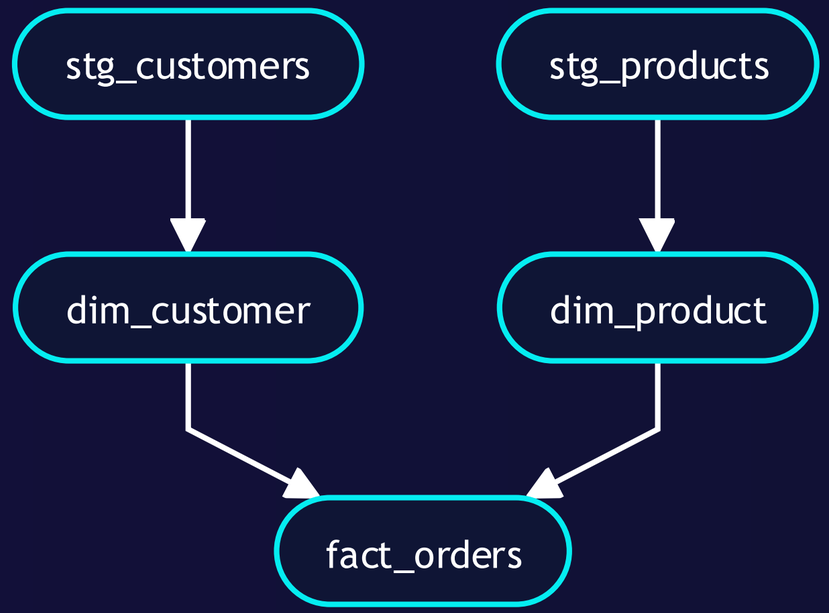
Problems addressed:

3 Brittle Orchestration

Dependencies inferred automatically from source and target relationships.

Independent threads execute in parallel.
Dependent threads wait. Always in the right order.

Execution order is declared, not improvised.



Incremental Loading

Read only what's changed.
Not everything. Every time.

Watermark tracking, CDC support, and incremental state management — declared in config, handled by the engine.

You define the strategy.
weevr manages the state.

Key Generation & Hashing

Surrogate Key Generation

Hash-based surrogate keys from business key columns.
Eight hash algorithms: xxhash64, sha256, md5, and more.
Null inputs replaced with deterministic sentinels.
Keys are always non-null. Always consistent.

Change Detection Hashing

Row-level change detection without CDC.
Hash a set of columns — if the hash changes, the row changed.
Pairs naturally with merge write mode for SCD patterns.

Idempotent Outputs

Run it once. Run it ten times.
The result is the same.

Overwrite

Naturally idempotent. Target is fully replaced. Same data in, same data out.

Append

Pair with incremental load to prevent duplicate rows. Same window in, same rows out.

Merge

Match keys ensure the same upsert result every time. Same source in, same target out.

Determinism isn't just a property of the config.
It's a guarantee of the output.

Structured Telemetry & Observability

Problems addressed:

4 Opaque Execution

OTel-compatible execution spans.
Structured JSON logging at every milestone.
Row counts, timing, and status per thread.
Full trace hierarchy — loom → weave → thread.

Route to Azure Monitor, Elasticsearch, Splunk — any JSON-capable log sink.
No custom parsing required.

When something goes wrong, you know exactly what happened, where, and what the data looked like.

telemetry.json · 12 lines

```json
{
  "timestamp": "2026-03-15T14:30:22Z",
  "level": "INFO",
  "thread": "fact_transactions",
  "weave": "facts",
  "status": "OK",
  "rows_read": 1847,
  "rows_written": 1204,
  "rows_quarantined": 3,
  "duration_ms": 847,
  "trace_id": "a1b2c3d4e5f67890"
}
```

Validations & Assertions

Pre-Write Validations

Rules evaluated against DataFrame before a single row is written.
Failing rows are quarantined — not dropped, not written, not silently lost.

Severity levels: info · warn · error · fatal

Post-Write Assertions

Checks evaluated against the target after every successful write.
Row count minimums, null checks, uniqueness constraints, custom expression.

The contract is verified, not assumed.

Same Pipeline. Two Ways.

The difference isn't the length.
It's what you must know to write it correctly.

fact_transactions.thread · 37 lines

```
config_version: "1.0"
name: fact_transactions
source:
  type: delta
  alias: raw_transactions
  path: abfss://raw@lake.dfs.core.windows.net/...
steps:
  - filter:
      condition: "amount > 0"
  - derive:
      columns:
        - transaction_date: "to_date('transaction_ts')"
  - select:
      columns:
        - transaction_id
        - customer_id
        - product_id
        - amount
        - currency
        - transaction_ts
        - transaction_date
  - cast:
      columns:
        amount: "Decimal(15,2)"
target:
  alias: silver_fact_transactions
```

fact_transactions.py · 96 lines

```
from delta.tables import DeltaTable
from pyspark.sql import SparkSession
from pyspark.sql.types import DecimalType

# ── Configuration ──────────────────────────────────
TARGET_TABLE  = "raw_fact_transactions"
SOURCE_PATH   = "abfss://raw@lake.dfs.core.windows.net/transactions.ts"
WATERMARK_KEY = "weevr_watermark_transaction_ts"
WATERMARK_COL = "transaction_date"
MATCH_KEY     = "transaction_id"

# ── Step 1: Read current watermark from Delta ──────
def get_watermark(spark, table_name, key):
    props = (
        spark.sql(f"SHOW TBLPROPERTIES {table_name}")
        .filter(col("key") == key)
        .collect()
    )
    return props[0]["value"] if props else None
    except Exception:
        return None

# ── Step 2: Check if target table exists ───────────
def target_exists(spark, table_name):
    try:
        DeltaTable.forName(spark, table_name)
```

The YAML declares intent. The Python implements infrastructure.
None of the extra code has anything to do with your data.

Seeing is believing

1 A basic pipeline from config

2 Validations and quarantine

3a Merge with soft delete and surrogate keys

3b Incremental watermark loading

4 A full pipeline weave — 5 threads, one DAG

5 A production loom — inheritance, params, conditions

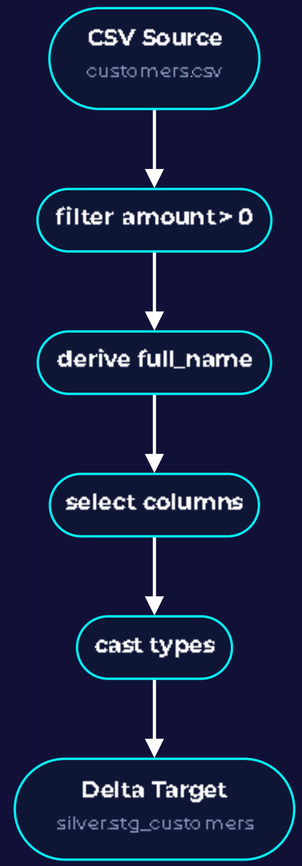
Demo 1: The Basic Thread

Problems addressed:

1. No Repeatable Structure
2. Copy-Paste Proliferation

```
ctx.run('stg_customers.thread')
```

The YAML file is the pipeline.
Nothing else required.



Demo 2: Validations & Quarantine

Problems addressed:

4 Opaque Execution

ctx.run('stg_customers_validated.thread')

Pre-write validation rules.
Failing rows quarantined — not dropped, not written.
Post-write assertions on the target.

Quality is part of the contract.

stg_customers_validated.thread · 14 lines

```yaml
validations:
  - name: email_not_null
    rule: "email IS NOT NULL"
    severity: error
  - name: valid_status
    rule: "status IN ('active','inactive')"
    severity: warn

assertions:
  - type: row_count
    min: 1
    severity: error
  - type: unique
    columns: [customer_id]
```

Demo 3a: Merge with Soft Delete

Problems addressed:

1. No Repeatable Structure

ctx.run(dim_product.thread')

Merge write mode — update, insert, soft delete.xxhash64 surrogate key generation.Null-safe by default.

Same config. Different inputs. Correct output.

RUN 1 · products.csv · initial load

product_id	name	price	sk_product
101	Widget	9.99	a3f2…8c1d
102	Gadget	24.99	b7e1…4a2f
103	Doohickey	4.99	c9d4…7b3e

products_v2.csv · same config

RUN 2 · dim_product · after merge

product_id	name	price	is_deleted	
101	Widget	12.99	false	+ update
102	Gadget	24.99	false	— same
103	Doohickey	4.99	true	× delete
104	Gizmo	19.99	false	+ new

price updated · soft deleted (row retained) · new insert

sk_product generated via xxhash64 on each run · null-safe by default

Demo 3b: Incremental Watermark

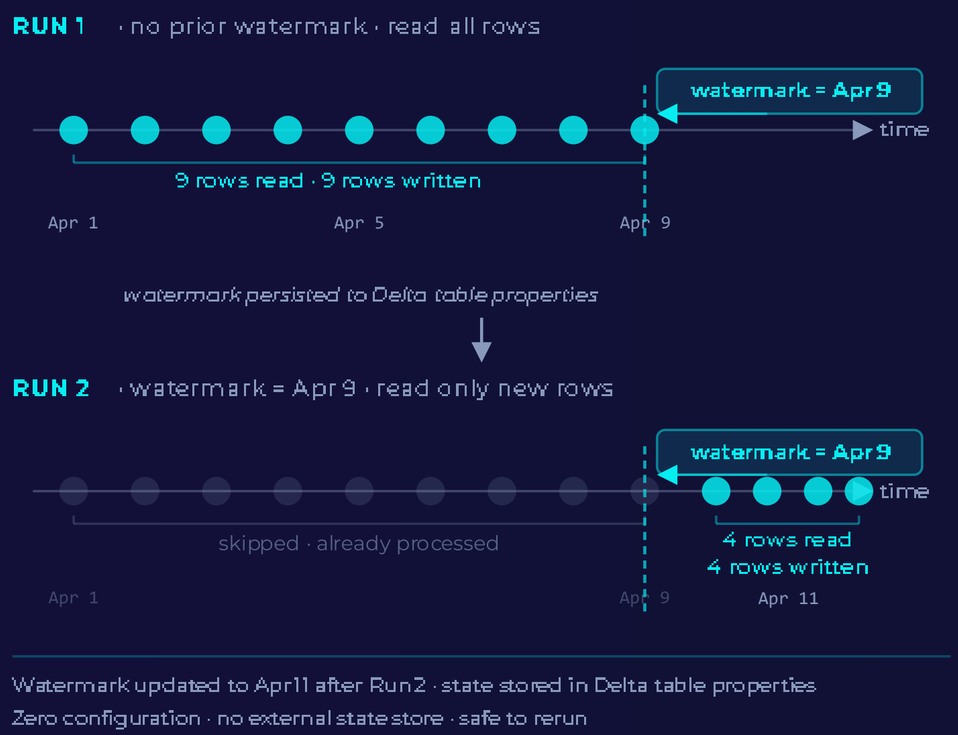
Problems addressed:

3 Brittle Orchestration

ctx.run(fact_transactions.thread')

Watermark-based incremental loading.
State persisted automatically — zero configuration.
Merge on transaction_id.

Run 1: full initial load
Run 2: only new transactions processed



Demo 4: Full Pipeline Weave

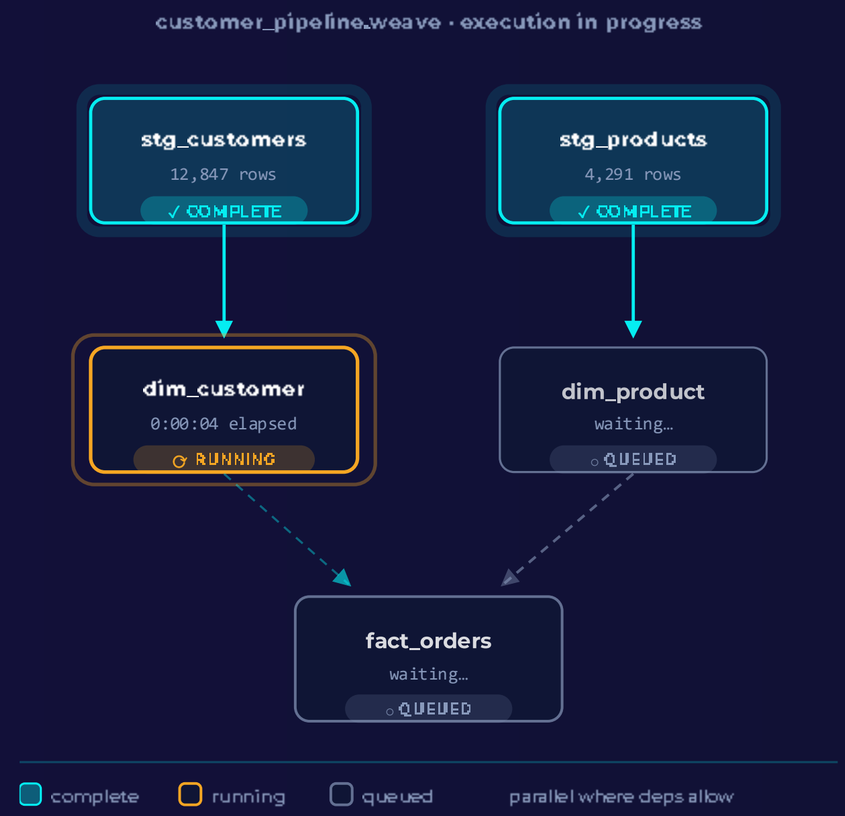
Problems addressed:

3. Brittle Orchestration
4. Opaque Execution

ctx.run('customer_pipeline.weave')

5 threads · Automatic DAG · Parallel execution
Narrow lookups · Hooks · Quality gates

The weave is the dependency contract.



Demo 5: A Production Loom

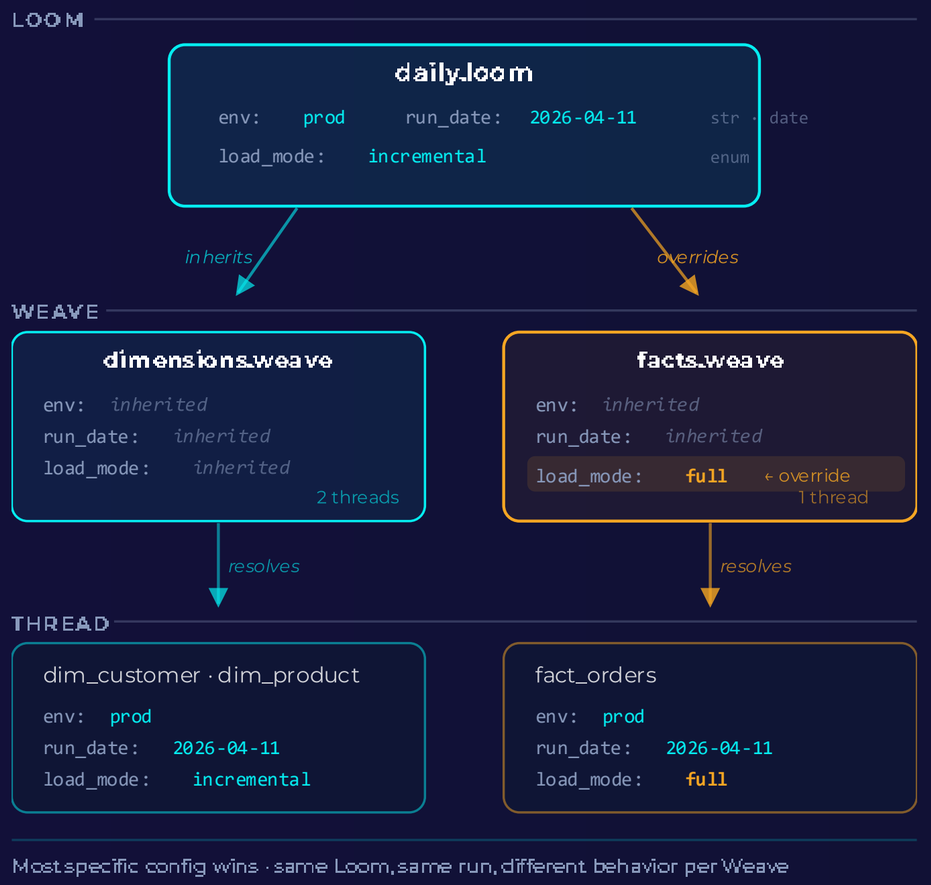
Problems addressed:

5 Environment Drift

ctx.run('daily.loom')

Two weaves · Config inheritance · Typed parameters
Conditional execution · Environment-agnostic

One config. Any environment.
The contract doesn't change.
Only the parameters do.



Declarative Pattern is the Answer

PHASE 1

Imperative Code

- Notebooks everywhere
- Copy-paste proliferation
- Environment drift
- Brittle orchestration
- Opaque execution

```
df = spark.read.csv(...)
df = df.filter(...)
df = df.withColumn(...)
# copy of copy_v3_FINAL
```

REALITY CHECK

Technical debt compounds quietly

Onboarding slows

Debugging becomes archaeology

⚠

PHASE 3 · weevr

Declarative Config

Problems solved

✓ One config. One contract.

✓ Repeatable by design

✓ DAG-aware execution

✓ Auditable & testable

✓ Environment-agnostic

```
thread:
name:    fact_orders
source:
type:    delta
transforms:
- type:    filter
- type:    aggregate
target:
mode:    merge
load:    incremental

# The config IS the contract
# Same inputs → same outputs
# Every time.
```

SQL has dbt.

PySpark on Fabric deserves the same.

dbt didn’t invent SQL transformation. It gave the community a standard way to do it. weevr is that bet for config-driven PySpark.

Whether it wins depends on whether practitioners like you decide it's worth building together.

Join the Community

① **Try it**

```
pip install weevr
```

20-minute quickstart
Full docs at
ardent-data.github.io/weevr

Apache 2.0 · Production ready
Fabric Runtime 1.3

② **Star it**

```
ardent-data/weevr
```

Every star signals
the ecosystem is ready

Watch for releases
Open issues · Read the roadmap

③ **Shape it**

GitHub Discussions
Share your thread patterns

Request transform types
Contribute to docs
Open a PR

Scan for docs.

Pierre LaFromboise
CDO · Covenant Technology Partners
linkedin.com/in/pierrelafromboise

I'll be around for Q&A. If you try weevr and hit a wall, open a discussion.
If it works — tell someone. That's how this grows.

Sound off.
The mic is all yours.
Influence the product roadmap.

Join the Fabric User Panel

Share your feedback directly with our Fabric product group and researchers.

https://aka.ms/JoinFabricUserPanel

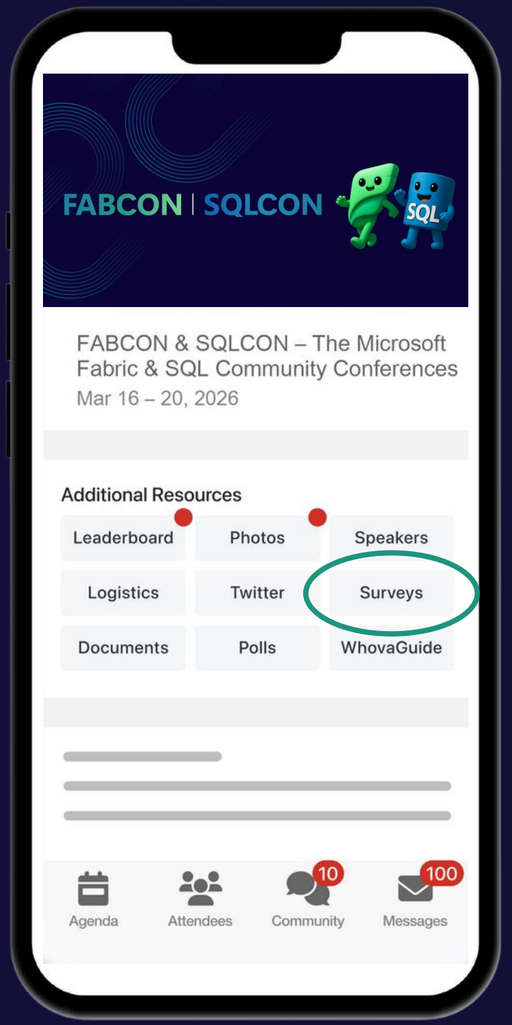
Join the SQL User Panel

Influence our SQL roadmap and ensure it meets your real-life needs

https://aka.ms/JoinSQLUserPanel

How was the session?

Complete Session Surveys in Whova for your chance to WIN PRIZES!



Get Two Fabric Certifications for FREE

Attendees of FABCON can take the Fabric Analytics Engineer or Fabric Data Engineer exam for free. Be part of the 2 fastest growing role-based certifications in Microsoft history.

Request your voucher by March 23, 2026.

https://aka.ms/fabcon/cert100