ACCELERATE **CHICAGO**

Craft software with confidence in 2026

SOFTWARE ENGINEERING CONFERENCE

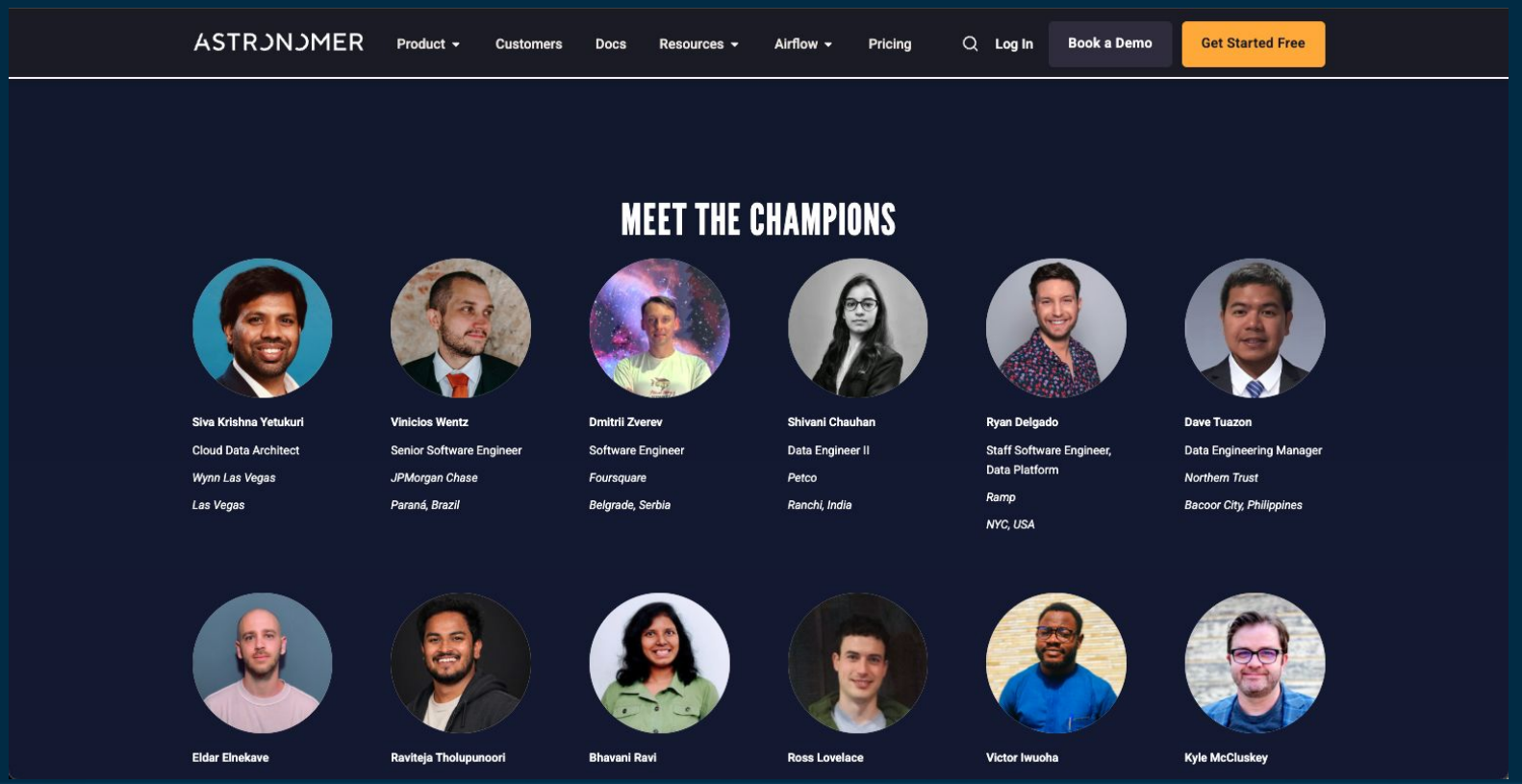
ACCELERATE CHICAGO

KYLE MCCLUSKEY

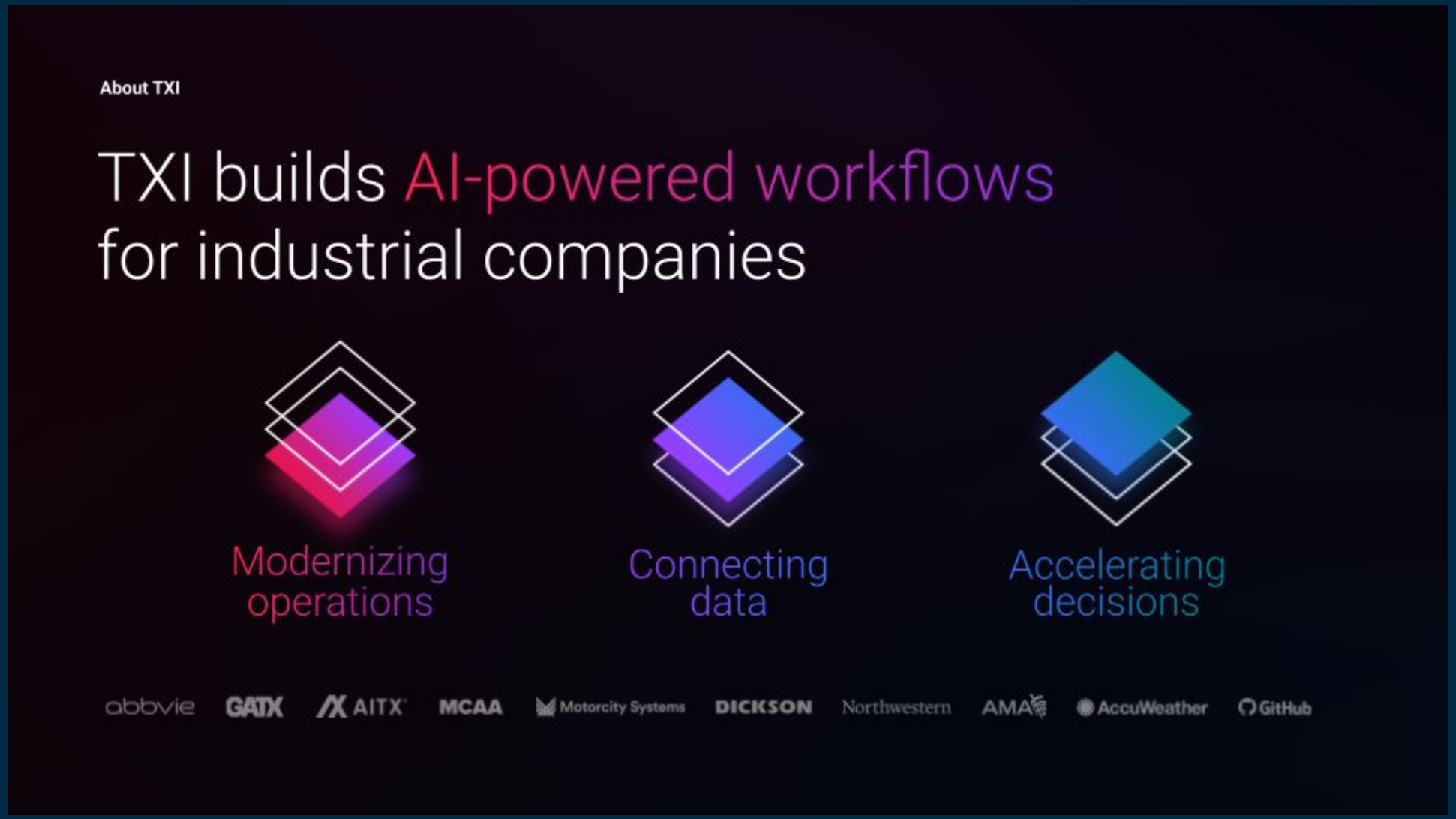
Spec-Driven Development for Airflow DAGs



About me


ASTRONOMER
Product
Customers
Docs
Resources
Airflow
Pricing
Log In
Book a Demo
Get Started Free
MEET THE CHAMPIONS
Siva Krishna Yetukuri
Cloud Data Architect
Wynn Las Vegas
Las Vegas
Vinicios Wentz
Senior Software Engineer
JPMorgan Chase
Paraná, Brazil
Dmitrii Zverev
Software Engineer
Foursquare
Belgrade, Serbia
Shivani Chauhan
Data Engineer II
Petco
Ranchi, India
Ryan Delgado
Staff Software Engineer, Data Platform
Ramp
NYC, USA
Dave Tuazon
Data Engineering Manager
Northern Trust
Bacoor City, Philippines
Eldar Elnekave
Raviteja Tholupunoori
Bhavani Ravi
Ross Lovelace
Victor Iwuoha
Kyle McCluskey

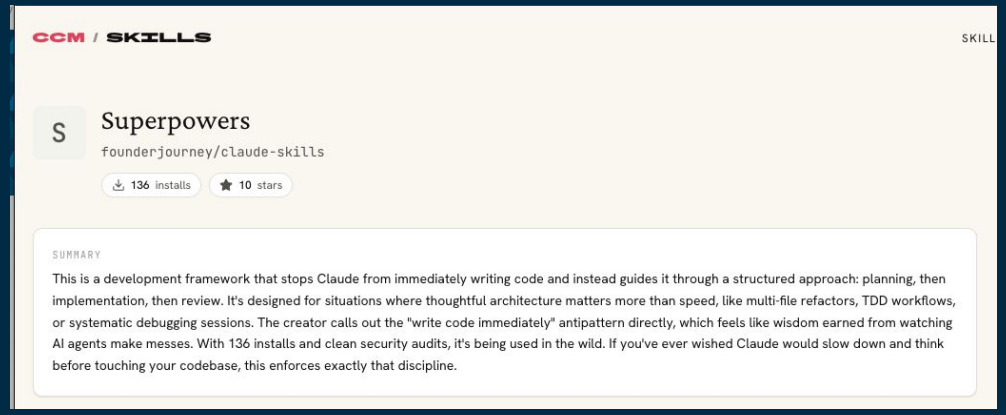
About TXI

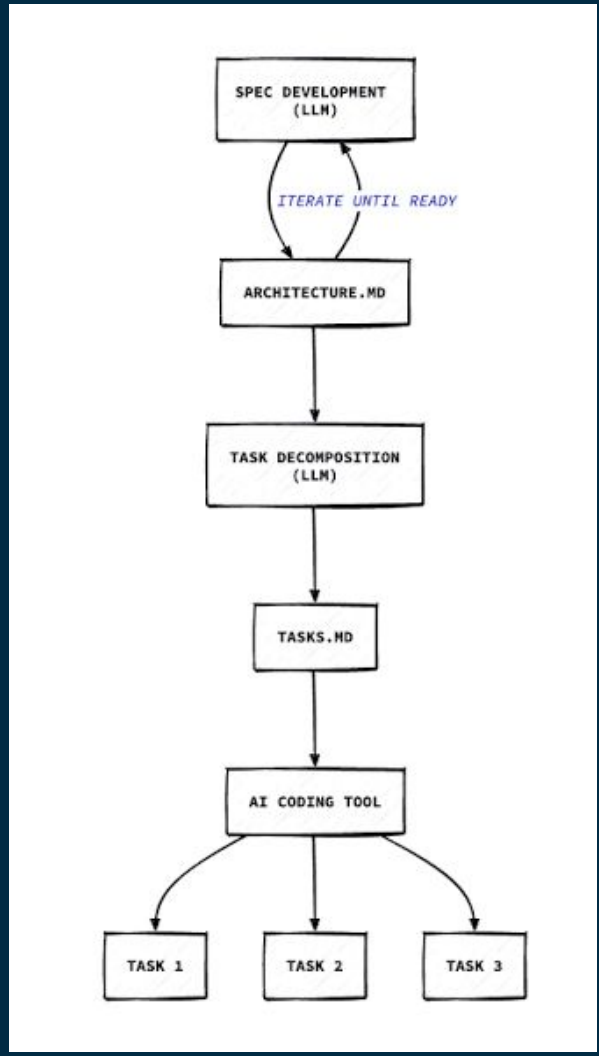

About TXI
TXI builds AI-powered workflows for industrial companies
Modernizing operations
Connecting data
Accelerating decisions
abbvie
GATX
AITX
MCAA
Motorcity Systems
DICKSON
Northwestern
AMA
AccuWeather
GitHub

Spec-Driven Development

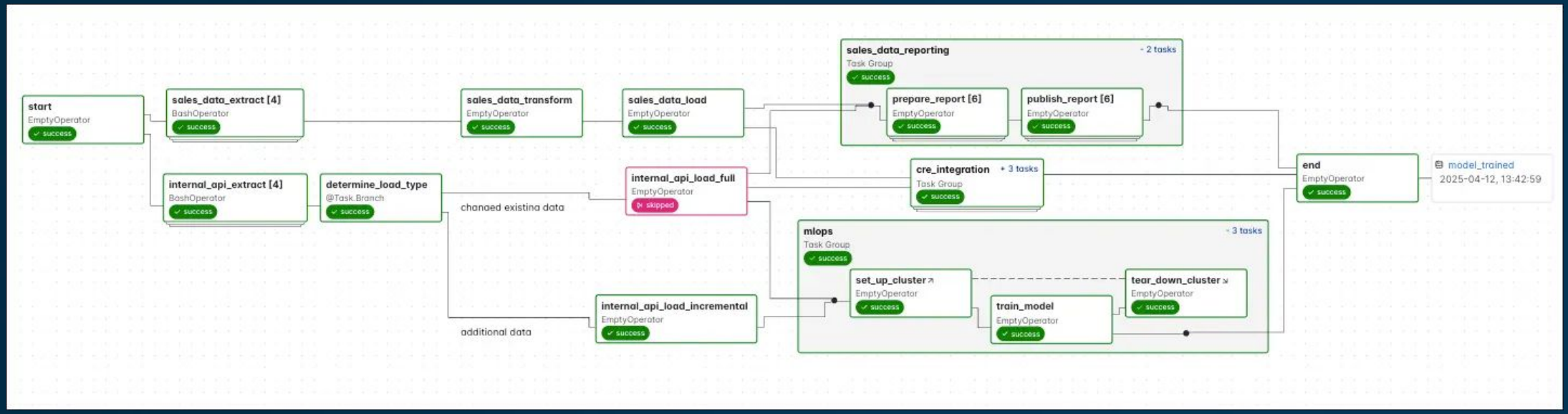
SDD @ TXI

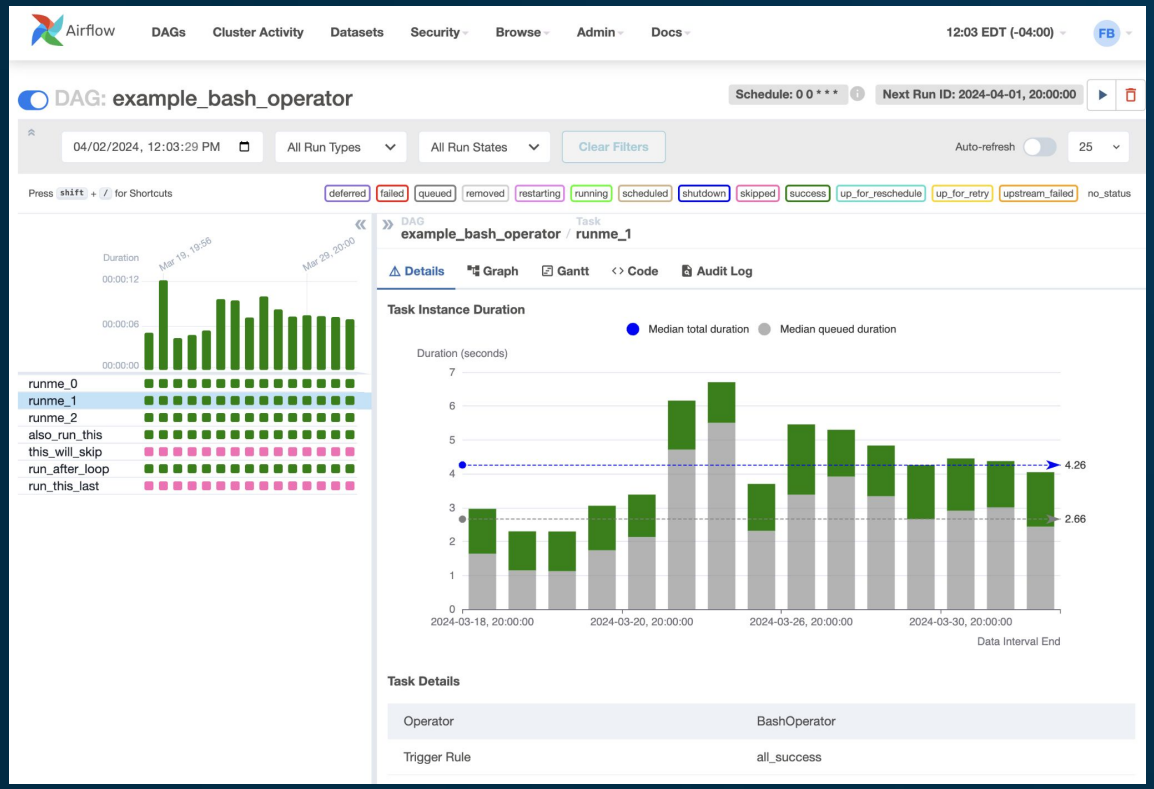
- rigorous response to “vibe-coding”
- OpenSpec, SpecKit, Superpowers, Ralph Wiggum Loop
- refine requirements with coding assistant
- generate tests first from spec
- dispatch specific agents to implement and review





Why Airflow?



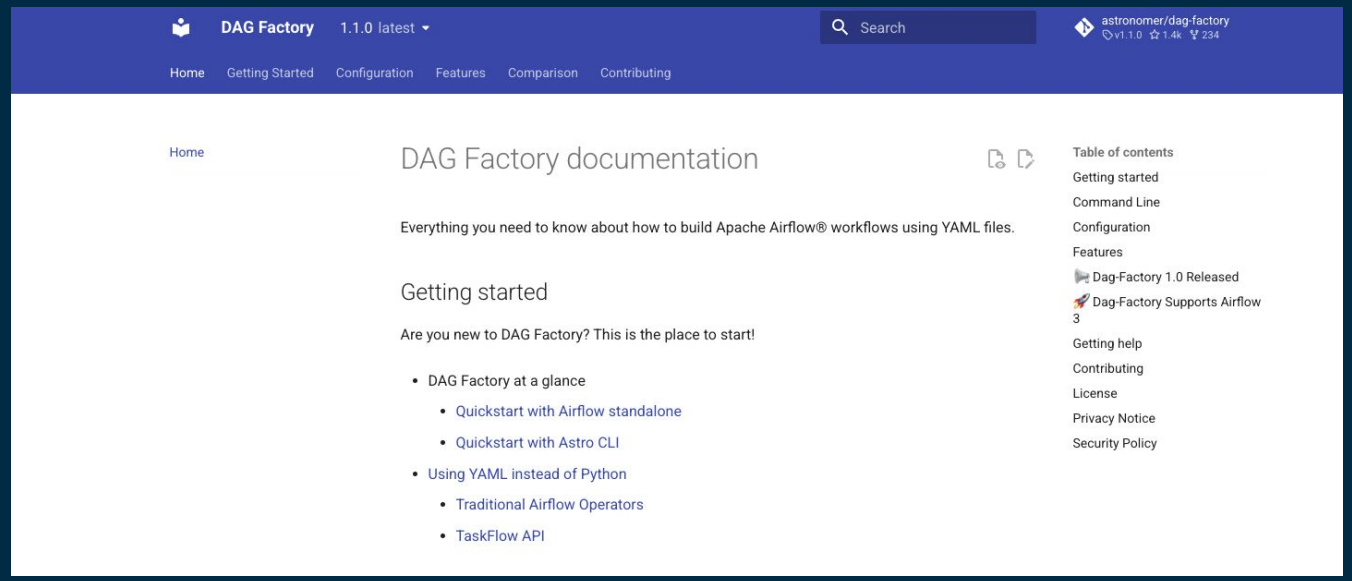


- Compositional Pipelines
- Huge community
- Conditional DAGs
- “Cron” on Steroids
- ETL/ELT Data Pipelines
- ML/AI Workflows

Data Engineering Workflows

Data Engineering w/ SDD

- Very Stable end of the Spec-Stability Spectrum
- Prescriptive declarations for source and target
- Deterministic
- Astronomer DAG Factory



aitx-brc — ✳ Create Airflow DAG Factory YAML specification — claude — 155×56

❯ I am working on a prototype of our full Airflow Ingestion Pipeline that leverages the Astronomer DAG Factory open-source tool. I want to develop a specification for what entities the pipeline needs to ingest, in what order and turn that specification into a DAGFactory YAML specification.

● I'll start by invoking the brainstorming skill, since we're designing a specification for new pipeline work — exploring requirements before jumping to YAML is exactly the right approach here.

● **Skill**(superpowers:brainstorming)
  └ Successfully loaded skill
● **Skill**(superpowers:brainstorming)
  └ Successfully loaded skill

● I'm going to follow the brainstorming process. Let me start by exploring the project context — particularly the existing pipeline structure, since the CLAUDE.md mentions a pipeline ingestion flow and there's a db_pipeline working directory.

  Listed 1 directory (ctrl+o to expand)

● **Bash**(echo "=== recent pipeline commits ===" && git log --oneline -15 -- brc/pipeline/ 2>/dev/null && echo "" && echo "=== any airflow/dagfactory references ===" && g…)
  └ === recent pipeline commits ===
    8d68c5be Refresh BRC seed data from 0617 FMS exports
    72f90bc1 refresh shop location contacts
    … +38 lines (ctrl+o to expand)

● There's substantial existing context here — Airflow DAGs already exist, plus an ADR for orchestration. Let me read the key files that capture the current ingestion order and Airflow approach.

  Read 2 files (ctrl+o to expand)

● Let me look at the existing Airflow DAG setup and the data-pipeline architecture doc to understand what's already there.

  Searched for 1 pattern, read 1 file (ctrl+o to expand)

● I've got a clear picture of the current state. Before I ask questions, here's what I found that's directly relevant:

  - **The ingestion order already exists in code.** run_all_ingestion.py runs ~42 scripts sequentially, and brc_ingestion_dag.py already encodes the full dependency graph as a hand-written Airflow DAG with **7 tiers** (cleanup → 16 independent reference tables → derived reference → entities → operational → events → details, plus a responsibility-matrix branch).
  - That existing DAG uses **fine-grained TaskFlow task groups** (extract → transform → validate → load) per entity, wired with Python >> operators.
  - **DAGFactory** (the Astronomer tool you want to use) is declarative: YAML defines tasks that reference Python callables/operators and dependencies. It supports task_groups and dependencies blocks.

  So this is really about **re-expressing an already-known ingestion graph as a DAGFactory YAML spec**, plus writing the entity/ordering spec that drives it. That reframes my questions. Let me ask them one at a time.

  First, the most consequential one — scope:

☐ Scope

For this prototype, how much of the ingestion graph should the DAGFactory YAML cover?

❯ 1. Full graph (all ~42 entities)
     Replicate the complete dependency graph from brc_ingestion_dag.py / run_all_ingestion.py — every reference, entity, operational, event, detail, and matrix table. Most faithful, larger YAML.
  2. Representative vertical slice
     A subset that exercises every tier and dependency pattern (one reference → entity → operational → event → detail chain + the matrix branch). Proves

● User answered Claude's questions:
  └ · How should the DAGFactory version relate to the existing hand-written brc_ingestion_dag.py? → Coexist as a separate prototype

● Coexist — clean, low risk. One thing I noticed while reading: run_all_ingestion.py actually covers **more** entities than the Python DAG does (e.g. cost-bucket tiers, lessee_contacts, header/detail extras, shop-network rates, shopping_statuses/reasons, brc_cost_types, and a final compute_validation_flags). Since you want the full graph, I'll treat run_all_ingestion.py's ordered SCRIPTS list as the authoritative entity set and derive the complete tiering from it.

  Last question — the tail end of the pipeline:

← ☐ Tail steps ✔ Submit →

Should the DAGFactory DAG include the post-load steps that run_all_ingestion.py runs but the current Python DAG omits?

❯ 1. [ ] compute_validation_flags
  Final task that computes validation flags across loaded BRC data. Runs last in run_all_ingestion.py. Include as a terminal tier after details/extras load.
  2. [ ] Post-ingestion validation report
  The validate_post_ingestion row-count/retention check (currently a Python function, not a task). Add as a final gate task that fails the DAG on empty/low-retention tables.
  3. [ ] shop_network rate tables
  ingest_shop_network_labor_rates / time_standards / qualification_rates — a parallel branch off shop_networks/shop_locations. Include in the graph.
  4. [ ] Type something
     Submit

---

  5. Chat about this

Enter to select · ↑/↓ to navigate · Esc to cancel

● This is the key technical input. Classification is now unambiguous:

- **Single main() tasks** (no full E/T/V/L surface): responsibility_types, cost_buckets, cost_bucket_user_teams, railcar_categories, shop_networks, shopping_statuses, brc_statuses, responsibility_matrix_v2 (VL only — DAG already runs it as main()), and compute_validation_flags.
- **E/T/V/L task groups** (full ETVL): the remaining ~30 scripts, with two needing has_transform=false (brc_qualifiers, railcar_class_codes) and one flagged edge case (railcar_lessee_history shows ETV — no load_to_target — I'll verify during implementation).

Now let me propose the design. There's one genuinely open technical decision — **how DAGFactory passes data between the E/T/V/L phases**, since YAML can't do what Python >> + return values do. Let me present that as approaches with a recommendation, then the full design.

Proposed approaches: inter-phase data passing in DAGFactory

The problem: the Python DAG's phases return staging-file paths and pass them as args (do_transform(extracted, ...)). Pure YAML has no return-value plumbing.

Approach A — Deterministic staging paths + explicit YAML dependencies (recommended).
The existing helper _staging_path(table, phase) already derives the path purely from (table, phase). So each phase callable can *recompute* its input path from its op_kwargs (module_name, table_name, flags) — it never needs the upstream return value to find its data. The only thing the return value provided was **ordering**, and in DAGFactory that comes from the explicit dependencies: block. No XCom templating at all. The phase logic lives in a small dag_factory_callables.py (thin wrappers mirroring do_extract/do_transform/…), and the YAML stays fully declarative.

Approach B — XCom path passing via Jinja in YAML. Each task pulls upstream output with "{{ ti.xcom_pull(...) }}" in op_kwargs. Faithful to the Python DAG's return-passing, but brittle, verbose, and error-prone in YAML.

I recommend **A** — it's the natural DAGFactory idiom and the deterministic-path helper already exists.

A second small choice: to avoid an O(16×N) explosion of tier-to-tier edges in YAML, I propose inserting lightweight **tier-gate EmptyOperator tasks** (tier1_complete, tier2_complete, …). Each group depends on the prior gate and feeds the next. This preserves the Python DAG's "all of tier N before any of tier N+1" semantics with O(N+M) readable edges instead of a cross-product.

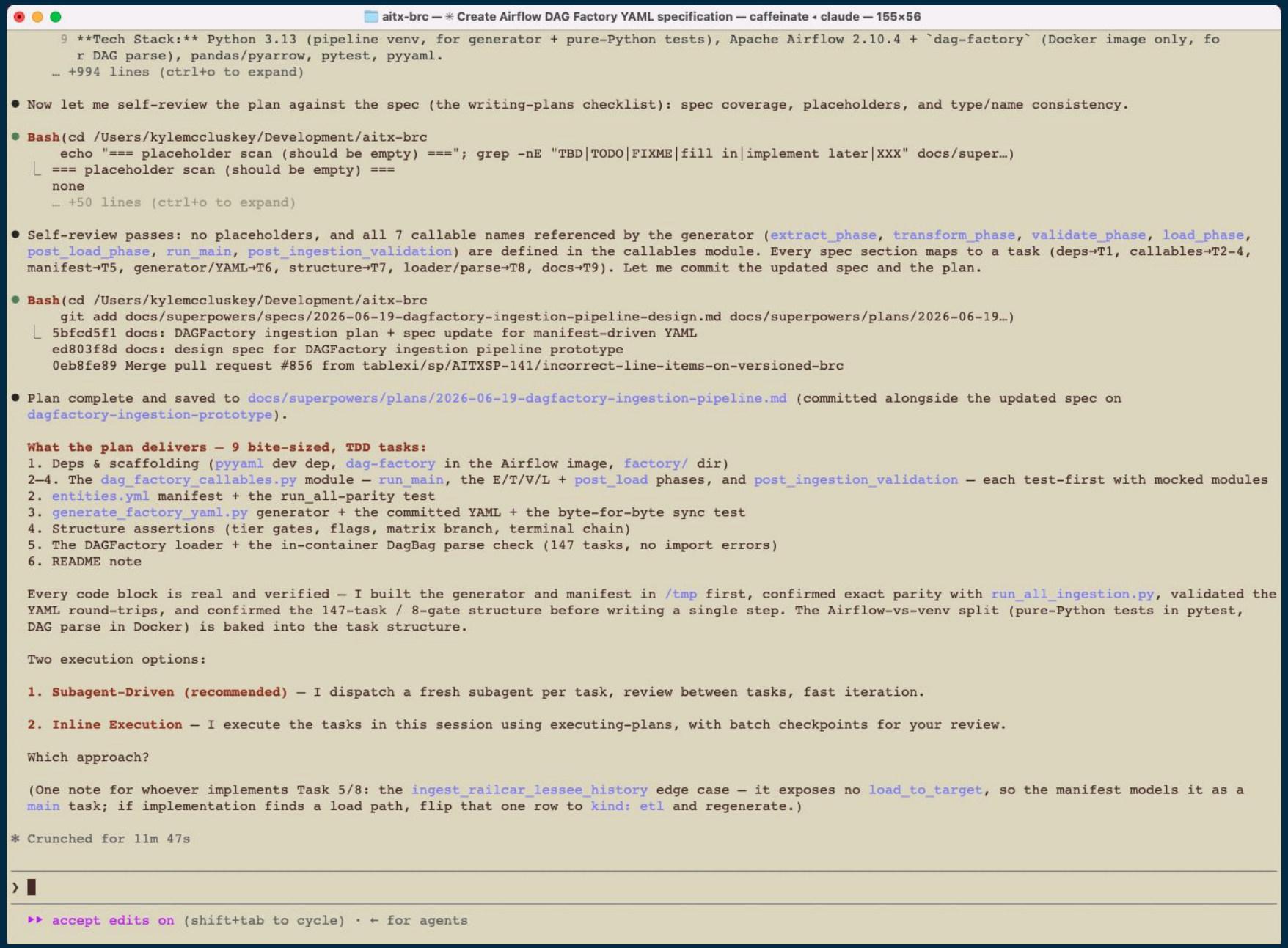
Does Approach A + tier gates sound right before I lay out the full tier structure?

✻ Cooked for 3m 16s

❯ Yes I like approach A with the Tier Gates as that conforms to the DAGFactory idiom and avoids the tier-to-tier edges problem

---

▸▸ accept edits on (shift+tab to cycle)


aitx-brc — ✳ Create Airflow DAG Factory YAML specification — caffeinate ◂ claude — 155×56
9 **Tech Stack:** Python 3.13 (pipeline venv, for generator + pure-Python tests), Apache Airflow 2.10.4 + `dag-factory` (Docker image only, fo
r DAG parse), pandas/pyarrow, pytest, pyyaml.
… +994 lines (ctrl+o to expand)
● Now let me self-review the plan against the spec (the writing-plans checklist): spec coverage, placeholders, and type/name consistency.
● Bash(cd /Users/kylemccluskey/Development/aitx-brc
echo "=== placeholder scan (should be empty) ==="; grep -nE "TBD|TODO|FIXME|fill in|implement later|XXX" docs/super…)
└ === placeholder scan (should be empty) ===
none
… +50 lines (ctrl+o to expand)
● Self-review passes: no placeholders, and all 7 callable names referenced by the generator (extract_phase, transform_phase, validate_phase, load_phase, post_load_phase, run_main, post_ingestion_validation) are defined in the callables module. Every spec section maps to a task (deps→T1, callables→T2-4, manifest→T5, generator/YAML→T6, structure→T7, loader/parse→T8, docs→T9). Let me commit the updated spec and the plan.
● Bash(cd /Users/kylemccluskey/Development/aitx-brc
git add docs/superpowers/specs/2026-06-19-dagfactory-ingestion-pipeline-design.md docs/superpowers/plans/2026-06-19…)
└ 5bfcd5f1 docs: DAGFactory ingestion plan + spec update for manifest-driven YAML
ed803f8d docs: design spec for DAGFactory ingestion pipeline prototype
0eb8fe89 Merge pull request #856 from tablexi/sp/AITXSP-141/incorrect-line-items-on-versioned-brc
● Plan complete and saved to docs/superpowers/plans/2026-06-19-dagfactory-ingestion-pipeline.md (committed alongside the updated spec on dagfactory-ingestion-prototype).
What the plan delivers — 9 bite-sized, TDD tasks:
1. Deps & scaffolding (pyyaml dev dep, dag-factory in the Airflow image, factory/ dir)
2–4. The dag_factory_callables.py module — run_main, the E/T/V/L + post_load phases, and post_ingestion_validation — each test-first with mocked modules
2. entities.yml manifest + the run_all-parity test
3. generate_factory_yaml.py generator + the committed YAML + the byte-for-byte sync test
4. Structure assertions (tier gates, flags, matrix branch, terminal chain)
5. The DAGFactory loader + the in-container DagBag parse check (147 tasks, no import errors)
6. README note
Every code block is real and verified — I built the generator and manifest in /tmp first, confirmed exact parity with run_all_ingestion.py, validated the YAML round-trips, and confirmed the 147-task / 8-gate structure before writing a single step. The Airflow-vs-venv split (pure-Python tests in pytest, DAG parse in Docker) is baked into the task structure.
Two execution options:
1. Subagent-Driven (recommended) — I dispatch a fresh subagent per task, review between tasks, fast iteration.
2. Inline Execution — I execute the tasks in this session using executing-plans, with batch checkpoints for your review.
Which approach?
(One note for whoever implements Task 5/8: the ingest_railcar_lessee_history edge case — it exposes no load_to_target, so the manifest models it as a main task; if implementation finds a load path, flip that one row to kind: etl and regenerate.)
* Crunched for 11m 47s
›
▸▸ accept edits on (shift+tab to cycle) · ← for agents

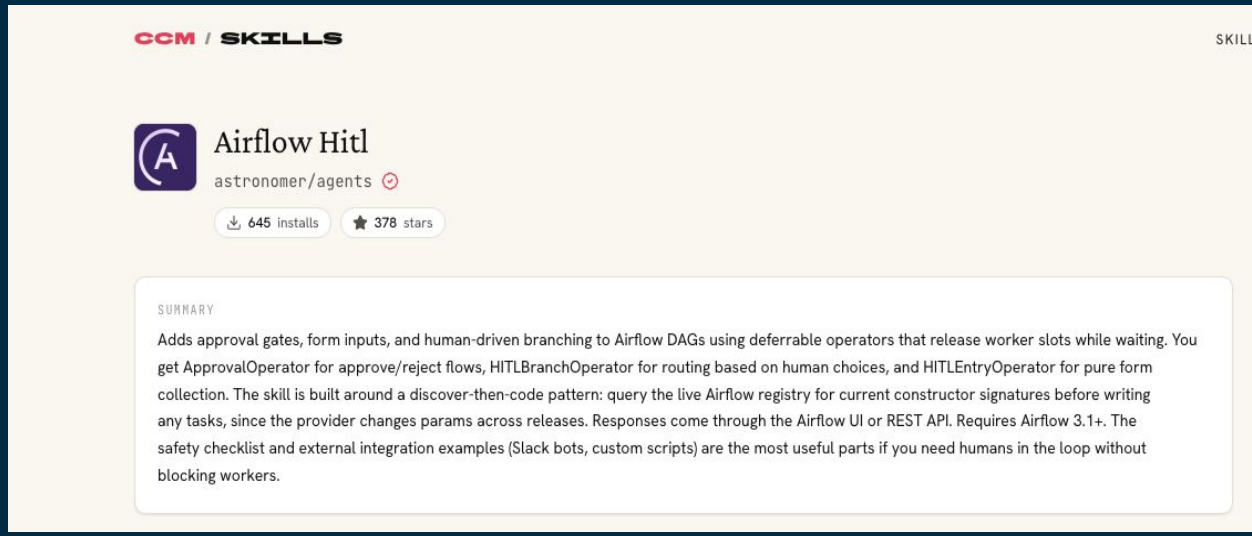
Key Takeaways for DE

- understanding your data is the hard part
- applicable to multiple DE technologies (DBT, Great Expectations)
- most time spent refining requirements and implementation plan
- process is still rigorous TDD
- Code reviews are essential, but different
  - Not “Check that the thing was done right.”
  - Rather “Check that the right thing was done.”

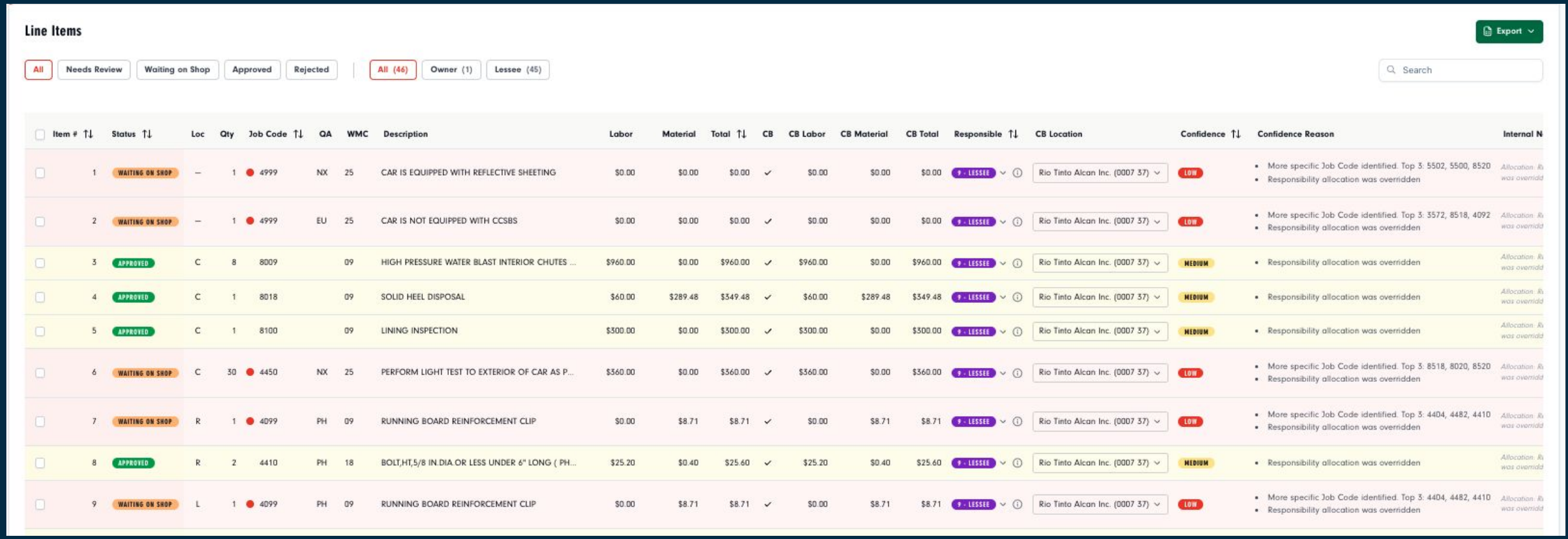
Machine Learning Workflows

Machine Learning with SDD

- “Not Very” end of the Spec-Stability Spectrum
- Qualitative descriptions of user behavior
- Non-Deterministic
- Astronomer Human-In-The-Loop Skill



AI Models generate suggestions of varying confidence



Line Items

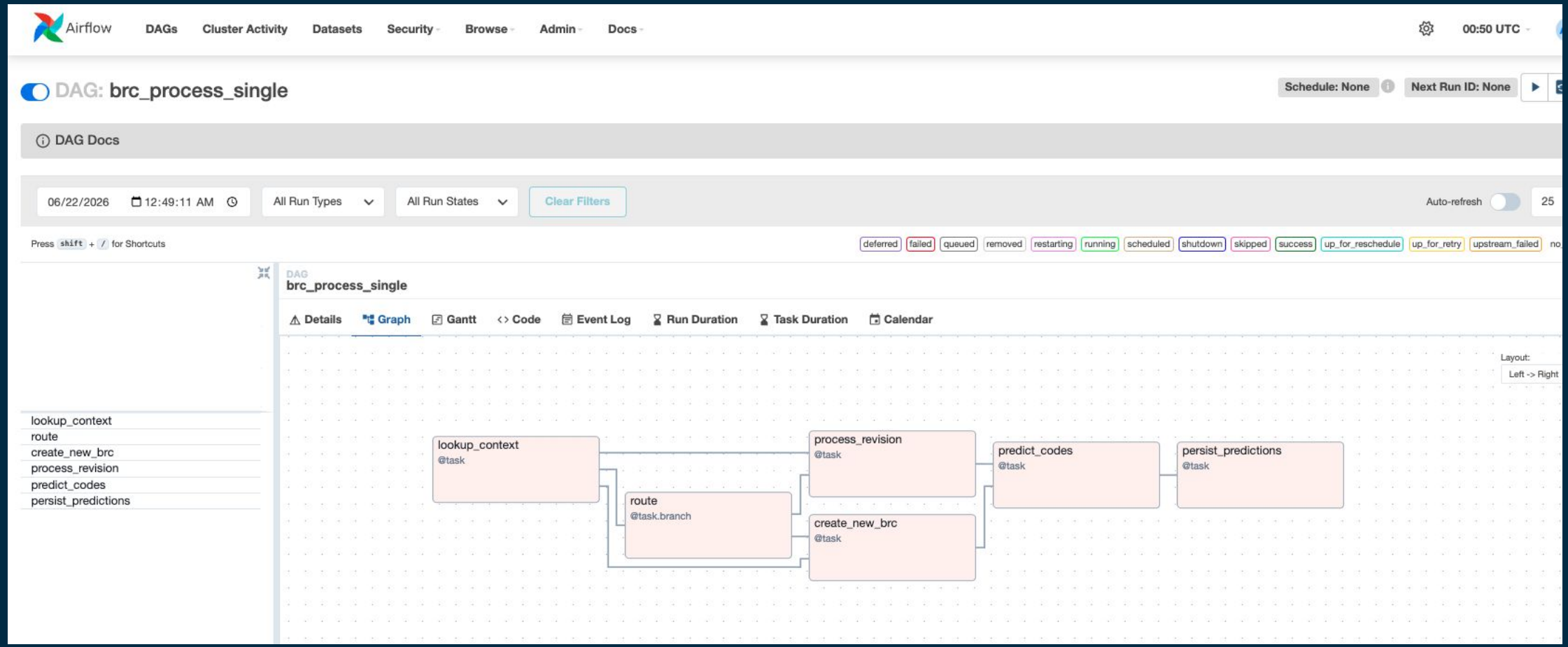
Export

All | Needs Review | Waiting on Shop | Approved | Rejected | All (46) | Owner (1) | Lessee (45)

Search

Item #	Status	Loc	Qty	Job Code	QA	WMC	Description	Labor	Material	Total	CB	CB Labor	CB Material	CB Total	Responsible	CB Location	Confidence	Confidence Reason	Internal N
1	WAITING ON SHOP	–	1	4999	NX	25	CAR IS EQUIPPED WITH REFLECTIVE SHEETING	$0.00	$0.00	$0.00	✓	$0.00	$0.00	$0.00	LESSEE	Rio Tinto Alcan Inc. (0007 37)	LOW	• More specific Job Code identified. Top 3: 5502, 5500, 8520 • Responsibility allocation was overridden	
2	WAITING ON SHOP	–	1	4999	EU	25	CAR IS NOT EQUIPPED WITH CCSBS	$0.00	$0.00	$0.00	✓	$0.00	$0.00	$0.00	LESSEE	Rio Tinto Alcan Inc. (0007 37)	LOW	• More specific Job Code identified. Top 3: 3572, 8518, 4092 • Responsibility allocation was overridden	
3	APPROVED	C	8	8009		09	HIGH PRESSURE WATER BLAST INTERIOR CHUTES ...	$960.00	$0.00	$960.00	✓	$960.00	$0.00	$960.00	LESSEE	Rio Tinto Alcan Inc. (0007 37)	MEDIUM	• Responsibility allocation was overridden	
4	APPROVED	C	1	8018		09	SOLID HEEL DISPOSAL	$60.00	$289.48	$349.48	✓	$60.00	$289.48	$349.48	LESSEE	Rio Tinto Alcan Inc. (0007 37)	MEDIUM	• Responsibility allocation was overridden	
5	APPROVED	C	1	8100		09	LINING INSPECTION	$300.00	$0.00	$300.00	✓	$300.00	$0.00	$300.00	LESSEE	Rio Tinto Alcan Inc. (0007 37)	MEDIUM	• Responsibility allocation was overridden	
6	WAITING ON SHOP	C	30	4450	NX	25	PERFORM LIGHT TEST TO EXTERIOR OF CAR AS P...	$360.00	$0.00	$360.00	✓	$360.00	$0.00	$360.00	LESSEE	Rio Tinto Alcan Inc. (0007 37)	LOW	• More specific Job Code identified. Top 3: 8518, 8020, 8520 • Responsibility allocation was overridden	
7	WAITING ON SHOP	R	1	4099	PH	09	RUNNING BOARD REINFORCEMENT CLIP	$0.00	$8.71	$8.71	✓	$0.00	$8.71	$8.71	LESSEE	Rio Tinto Alcan Inc. (0007 37)	LOW	• More specific Job Code identified. Top 3: 4404, 4482, 4410 • Responsibility allocation was overridden	
8	APPROVED	R	2	4410	PH	18	BOLT,HT,5/8 IN.DIA.OR LESS UNDER 6" LONG (PH...	$25.20	$0.40	$25.60	✓	$25.20	$0.40	$25.60	LESSEE	Rio Tinto Alcan Inc. (0007 37)	MEDIUM	• Responsibility allocation was overridden	
9	WAITING ON SHOP	L	1	4099	PH	09	RUNNING BOARD REINFORCEMENT CLIP	$0.00	$8.71	$8.71	✓	$0.00	$8.71	$8.71	LESSEE	Rio Tinto Alcan Inc. (0007 37)	LOW	• More specific Job Code identified. Top 3: 4404, 4482, 4410 • Responsibility allocation was overridden	

DAG Persist them for async review



Bring a Human Into The Loop

> I have an additional use case I would like to prototype. I want to explore using the Astronomer HITL skill to extend the existing brc_process_single DAG. After the predict_codes task predicts its codes, I would like to prompt the user to choose one before they get persisted

● This is a new feature to prototype, so I'll start with the brainstorming skill before any implementation — same disciplined flow as last time.

● **Skill**(superpowers:brainstorming)
  └ Successfully loaded skill

● I'll explore the relevant context first: the existing brc_process_single DAG (especially predict_codes and the persistence step), and what "Astronomer HITL" actually means against this repo's Airflow version.

  Read 2 files (ctrl+o to expand)

● I've read the existing brc_process_single DAG. The flow is lookup_context → route → [create_new_brc | process_revision] → predict_codes → persist_predictions, where predict_codes returns top-K rows {brc_detail_id, prediction_type, rank, label, probability} and persist_predictions writes **all ranks** to brc.brc_detail_predictions.

  One thing I need to verify before anything else: **Astronomer/Airflow HITL is an Airflow 3.x feature**, but this repo's image is pinned to apache/airflow:2.10.4. That's a potential hard blocker, so let me confirm the actual version requirement from Astronomer's docs rather than assume.

● I have enough to shape the design. Here it is in four sections — react to any of them.

Design: HITL job-code selection in brc_process_single

1. Environment (isolated Airflow 3.1)

A new self-contained brc/pipeline/airflow-hitl/ with its own docker-compose.yml on **apache/airflow:3.1.x**, joining the existing aitx-network. It **reuses** the shared dev **SQL Server** and the **model prediction service** (only Airflow itself is new) — so predict_codes and the brc_details update hit the same DB/model as today. Crucially, HITL operators are **deferrable**, so this compose must include a **Triggerer** component (the 2.10 compose doesn't), alongside the Airflow 3 api-server/scheduler/dag-processor. The 2.10.4 setup and all its DAGs are untouched.

2. DAG flow & the HITL gate

A ported copy brc_process_single_hitl keeping the real shape, with the gate inserted between predict and persist:

```
lookup_context → route → [create_new_brc | process_revision]
   → predict_codes  (unchanged: top-K job_code + why_made per detail)
   → build_choice_inputs   (reshape: one {brc_detail_id, options:[top-K job codes], narrative} per detail that got a job_code prediction)
   → choose_job_code        (HITLOperator, dynamic-mapped .expand() — one form per detail line; options = that line's predicted job codes)
   → apply_choice           (per detail: UPDATE brc.brc_details with the chosen code)
```

The human answers each mapped form in the Airflow 3 UI **"Required Actions"** inbox; the chosen option returns via XCom to apply_choice.

3. Data & persistence

- apply_choice (mapped per detail) runs UPDATE brc.brc_details SET legacy_applied_job_code = <chosen>, job_code_id = (SELECT job_code_id FROM brc.brc_job_codes WHERE job_code = <chosen>) WHERE brc_detail_id = ?.
- Details **without a narrative/prediction** get no form and are skipped (matches today's predict behavior).
- **Scope cuts (YAGNI):** why_made is **out of scope** (job_code only); the original persist_predictions (writing top-K to brc_detail_predictions) is **replaced** by apply_choice — the chosen code applied to the line is the prototype's outcome.
- **No timeout** for the prototype (the form waits for a human). I'll note HITLOperator's defaults/timeout knob as a documented future toggle (e.g., auto-pick rank-1).

4. Porting, testing, risks

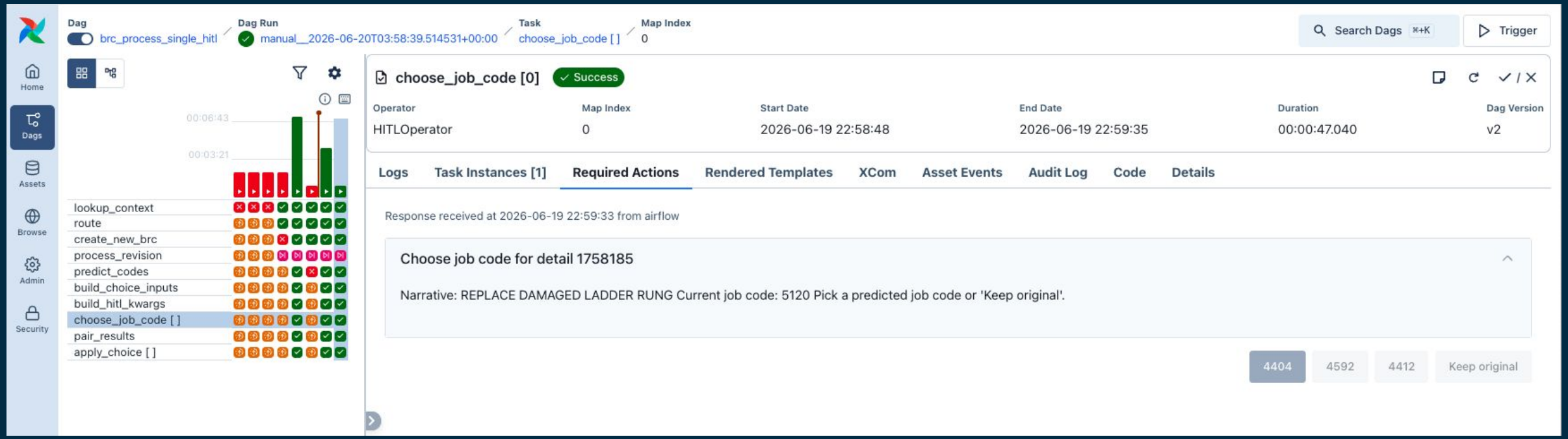
- **Airflow 2→3 port** is real work: imports move (e.g., from airflow.sdk import dag, task; basic operators + HITL come from apache-airflow-providers-standard), and I'll verify @task.branch, dynamic mapping, dag_run.conf access, and the exact HITLOperator return-dict shape during planning (against the 3.1 API docs).
- **Testing:** pure-Python unit tests for the reshape (build_choice_inputs) and the apply_choice SQL logic (mockable, run in the pipeline venv); plus an **in-container** check that the Airflow 3.1 env starts, the DAG imports with no errors, and the HITL task defers and renders a form (the manual end-to-end being: trigger → answer forms in UI → verify brc_details updated).
- **Risks:** the Airflow 3.1 compose (4 components + triggerer) is the heaviest new piece; the full DAG needs the model service + DB reachable to run end-to-end; exact HITL API/return shape to be pinned in planning.

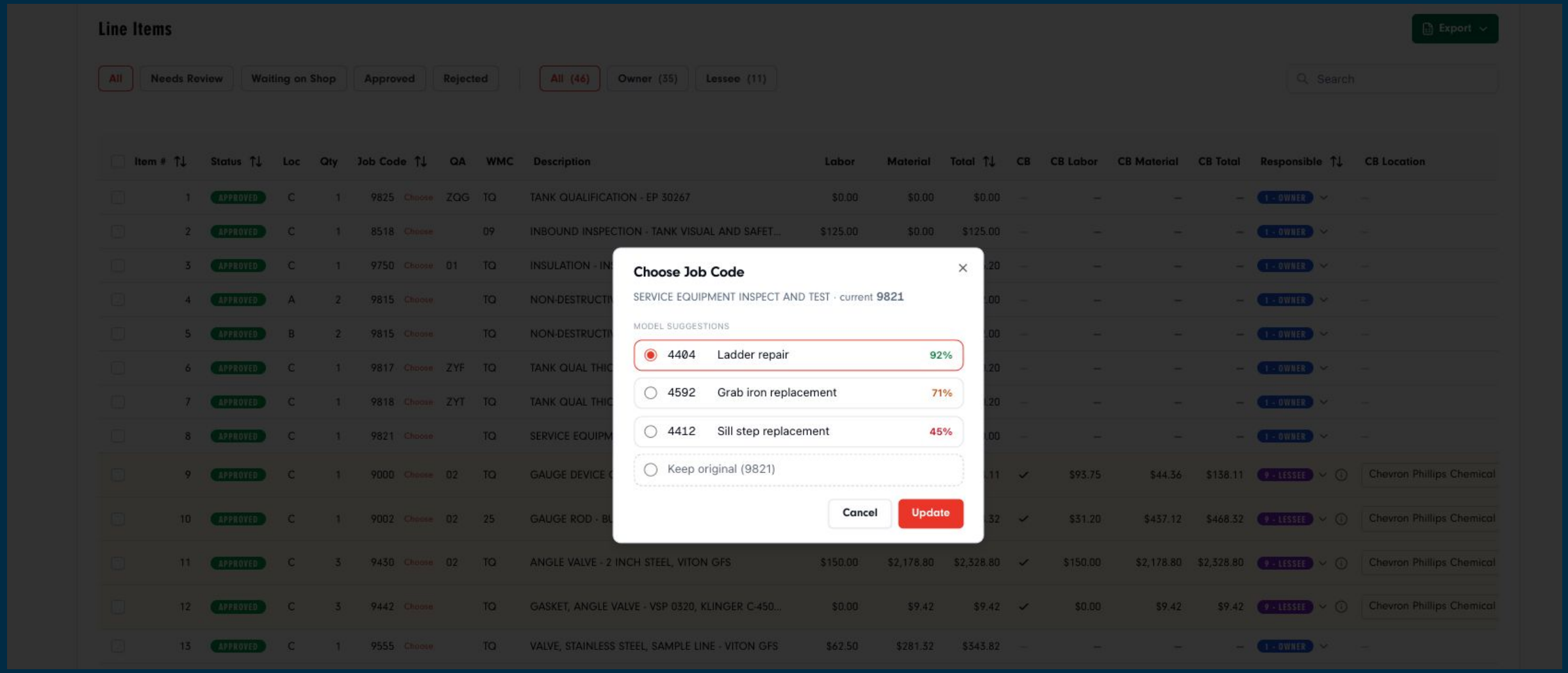
Does this look right — particularly replacing persist_predictions with apply_choice, dropping why_made, and the isolated-but-DB-sharing environment? Once you're happy I'll write the spec.

✻ Worked for 5m 0s

❯ yes, this looks good

● Approved. Following the same flow as before — I'll create a dedicated branch (off main, isolated from the DAGFactory branch), write the spec, self-review, then hand it to you for review.


Dag
brc_process_single_hitl
Dag Run
manual__2026-06-20T03:58:39.514531+00:00
Task
choose_job_code [ ]
Map Index
0
Search Dags ⌘+K
Trigger
Home
Dags
Assets
Browse
Admin
Security
00:06:43
00:03:21
lookup_context
route
create_new_brc
process_revision
predict_codes
build_choice_inputs
build_hitl_kwargs
choose_job_code [ ]
pair_results
apply_choice [ ]
choose_job_code [0]
Success
Operator
HITLOperator
Map Index
0
Start Date
2026-06-19 22:58:48
End Date
2026-06-19 22:59:35
Duration
00:00:47.040
Dag Version
v2
Logs
Task Instances [1]
Required Actions
Rendered Templates
XCom
Asset Events
Audit Log
Code
Details
Response received at 2026-06-19 22:59:33 from airflow
Choose job code for detail 1758185
Narrative: REPLACE DAMAGED LADDER RUNG Current job code: 5120 Pick a predicted job code or 'Keep original'.
4404
4592
4412
Keep original


Line Items
Export
All
Needs Review
Waiting on Shop
Approved
Rejected
All (46)
Owner (35)
Lessee (11)
Search
Item #
Status
Loc
Qty
Job Code
QA
WMC
Description
Labor
Material
Total
CB
CB Labor
CB Material
CB Total
Responsible
CB Location
Choose Job Code
SERVICE EQUIPMENT INSPECT AND TEST · current 9821
MODEL SUGGESTIONS
4404 Ladder repair 92%
4592 Grab iron replacement 71%
4412 Sill step replacement 45%
Keep original (9821)
Cancel
Update

Key Takeaways for ML

- understanding your user is the hard part
- applicable to multiple ML technologies (classical ML, LLM)
- most time spent refining requirements and implementation plan
- parts of the process that can’t be TDD are delegated to a human
- “reviews” occur in real-time based on rules or thresholds

QUESTIONS?

ASK ME IN THE APP

Download on the App Store

GET IT ON Google Play

PLEASE RATE MY SESSION