

# DBT Tooling Research

Opportunity assessment for an LLM-leveraged DBT tool

Compiled from research agents

2026-04-26

- [Overview](#)
- [Appendix A — Master Index and Synthesis](#)
  - [DBT Tooling Research — Master Index](#)
- [Appendix B — Initial Opportunity Report](#)
  - [DBT Tooling Opportunity Report](#)
- [Appendix C — Pain Points Deep Dive](#)
  - [dbt Pain — Deep Dive into the Lived Experience](#)
- [Appendix D — AI/LLM Tools Landscape](#)
  - [The dbt AI/LLM Tooling Ecosystem — Deep Dive \(April 2026\)](#)
- [Appendix E — Tool Design Sketches](#)
  - [dbt LLM Tool Design Sketches](#)
- [Appendix F — Claude + DBT Technical Surface](#)
  - [dbt + Claude: Technical Surface for 2026](#)

## Overview

This document collects the full output of a research effort to evaluate whether to build an LLM-leveraged DBT tool. Five research artifacts are included as labeled appendices:

- **Appendix A** — Master index and synthesis (recommended starting point)
- **Appendix B** — Initial tooling opportunity report
- **Appendix C** — Pain points deep dive (voice of the user, ~5,400 words)
- **Appendix D** — AI/LLM dbt tools landscape (~4,225 words)
- **Appendix E** — Tool design sketches for top three opportunities (~7,800 words)
- **Appendix F** — Claude + dbt technical surface reference (~5,300 words)

Total: ~28,000 words of research compiled 2026-04-25 to 2026-04-26.

# Appendix A — Master Index and Synthesis

## DBT Tooling Research — Master Index

Compiled 2026-04-25 to inform a decision on whether to build an LLM-leveraged DBT tool that synergizes with clauditor (this repo's LLM-graded skill evaluation framework).

### The five research artifacts

1. [dbt-tooling-opportunity-report.md](#) — initial scan: top 10 pain points, full tooling landscape table, seven LLM-shaped opportunities ranked, strategic notes, sources.
2. [dbt-pain-deep-dive.md](#) (~5,400 words) — voice-of-the-user research with direct quotes, failure stories, indirect signals across writing/testing/validating/maintaining. 40+ sourced quotes from Pedram Navid, Tristan Handy, Max Halford, dbt Discourse, GitHub issues.
3. [dbt-ai-tools-deep-dive.md](#) (~4,225 words) — every AI/LLM tool in dbt as of April 2026: comparative matrix, incumbent map across 12 jobs-to-be-done, quality-of-output assessment, architectural patterns, the eval gap, strategic openings. Live GitHub data (stars, last-push, open issues).
4. [dbt-tool-design-sketches.md](#) (~7,800 words) — full design treatment of three top opportunities (PR Companion, YAML Forge, ProcMigrate) with named-persona user journeys, mock UX, technical architecture, hard problems, MVP scope, path-to-100-users, 12-month evolution, why-not. Cross-cutting recommendation matrix.
5. [dbt-claude-technical-surface.md](#) (~5,300 words) — implementer reference: dbt artifact schemas (manifest v9-v20), dbt MCP server tool inventory, Anthropic SDK / Agent SDK / Claude Code subprocess surfaces, warehouse integration patterns, CI/CD shapes with action.yml snippets, output integration, OSS templates to learn from, token economics (\$0.11-\$0.80/PR).

Total: ~28,000 words of research across five files.

### Headline conclusion

**Build a YAML/tests/docs generator with quality eval baked in** (“YAML Forge”-shaped). The decisive factors:

- **Synergy with clauditor (10/10).** The Phase-3 rubric grader is literally `quality_grader.grade_quality()` applied to a new artifact class. Every clauditor improvement compounds; every Forge dollar funds clauditor R&D.
- **The eval gap is the only durable moat.** Every vendor ships generators (dbt Copilot, Paradime, Altimate, codegen). Almost nobody systematically grades their own output on dbt artifacts. The single OSS

dbt-native eval framework (paradime-io/dbt-llm-evals, 25%) targets application output, not artifact quality.

- **The “drop always-pass tests” pruning premise is a story** that no incumbent tells. Stories drive adoption.
- **Solo-dev feasibility (7/10)**. The LLM work IS the core value, unlike PR Companion (mostly integration plumbing) or ProcMigrate (agent loop + dialects + parity = years of work).

Honest tradeoffs: - Forge has the lowest pain severity (6/10) — chronic, not acute. PR Companion (7/10) and ProcMigrate (10/10) have more urgent pain but worse defensibility / capital requirements for a solo dev. - ProcMigrate has the biggest dollar TAM but is sales-led, capital-intensive, brutally competed by Datafold’s existing Migration Agent. Wrong shape. - PR Companion is the recommended **year-2 expansion** after Forge proves the clauditor-grading-as-product pattern.

Final scoring (out of 60): YAML Forge **46** > PR Companion 43 > ProcMigrate 38.

## Top three pain points to anchor on

From the deep-dive (with quotes traceable in dbt-pain-deep-dive.md):

1. **schema.yml drudgery** — the most-cited single complaint. Hand-writing column descriptions and tests, keeping them in sync with SQL. dbt-codegen and dbt Copilot help but neither evaluates output quality.
2. **Test coverage is noise, not signal** — ~1% column unit-test coverage typical. Hundreds of not\_null/unique tests that catch nothing. Unit tests (1.8) have YAML-fixture friction that killed adoption. The “green CI, broken dashboards” failure mode is the canonical horror story.
3. **PR review is vibes** — reviewers can’t see what data changed or what breaks downstream without manual checkout. Slim CI tests SQL syntax, not data. Datafold solves it for \$30k-\$75k/yr.

YAML Forge attacks #1 and #2 directly. PR Companion would attack #3 in year 2.

## Strategic context (April 2026)

- **dbt Labs + Fivetran merger (Oct 2025)** — community fear of Cloud-only features driving Core-native interest. Tailwind for OSS-first tooling.
- **dbt Fusion engine (Rust, May 2025 beta)** — manifest v20 is additive over v12; readers continue working. Cross-engine artifact mixing breaks Recce-style diffs.
- **dbt MCP Server (2025)** — 544, 39 issues, last push 2026-04-24. Most interesting tools (Discovery, Admin, Semantic Layer) are dbt Cloud-only. For OSS-first tools, depending on dbt-mcp over direct artifact reads has questionable ROI.
- **dbt-labs/dbt-agent-skills (Feb 2026)** — 430 in under 3 months. The Skills shape is winning as a substrate.

- **Ultimate dbt Power User stability complaints** dominate over AI-quality complaints — incumbents are losing on basic reliability, not on intelligence.

## Recommended next moves

If the user wants to validate this direction:

1. **Spike:** Build a 1-day prototype that loads `manifest.json` for a real dbt project, picks one model, asks Claude to draft `schema.yml`, samples warehouse data via `dbt show`, runs each candidate test, prunes always-pass. Measure: are the surviving tests actually useful? (This is the load-bearing assumption.)
2. **Validate the eval rubric:** Define a clauditor-style rubric for “good `schema.yml`” — coverage, accuracy, terminology consistency, no-noise-tests. Grade 5 hand-written examples and 5 LLM examples. Does the rubric discriminate? (This is whether clauditor’s grading methodology survives in this domain.)
3. **Distribution test:** Post the prototype to dbt Slack `#tools-and-utilities` and `r/dataengineering` with a “what would make this useful?” framing. Volume of replies = demand signal.

If signals are positive, the YAML Forge MVP is shippable in 2-3 days per the design doc.

## File map

```
docs/temp/
├── dbt-research-index.md           ← you are here
├── dbt-tooling-opportunity-report.md ← executive scan (start
here)
├── dbt-pain-deep-dive.md           ← user voice
├── dbt-ai-tools-deep-dive.md       ← competitive landscape
├── dbt-tool-design-sketches.md     ← three full designs +
recommendation
└── dbt-claude-technical-surface.md ← implementer reference
```

Read order for a builder evaluating viability: opportunity-report → pain-deep-dive → ai-tools-deep-dive → tool-design-sketches → technical-surface.

Read order for a builder ready to start: tool-design-sketches (Forge section) → technical-surface → opportunity-report Section 4.

# Appendix B — Initial Opportunity Report

## DBT Tooling Opportunity Report

A research synthesis for builders considering an LLM-leveraged DBT tool. Current as of April 2026. The dbt landscape is in flux: dbt Labs merged with Fivetran in October 2025, the Rust-based Fusion engine went public beta in May 2025, and the dbt MCP server shipped in 2025 — meaning the surface for AI-augmented tooling is wide open and not yet locked down by incumbents.

### 1. Top 10 Pain Points (ranked by frequency × severity)

1. **Schema.yml + documentation drudgery** — the single most-cited recurring complaint. Authors must hand-write column lists, descriptions, and tests in YAML, keep them in lockstep with model SQL, and re-do the work whenever the SELECT changes. Existing palliatives (dbt-codegen's `generate_model_yaml`, dbt Copilot's "Generate Documentation") help but are partial: codegen still requires manual descriptions, Copilot is gated behind dbt Cloud Enterprise.
2. **Test coverage is hard, slow, and low-signal** — community consensus is that ~1% of columns get unit-test coverage; teams either write hundreds of `not_null/unique` tests that catch nothing, or skip tests entirely. dbt's native unit tests have well-documented limits: incremental-merge logic can't be tested, only "exists" checks (no negative assertions), high YAML setup cost, and tests rot as logic evolves.
3. **PR review is "vibes-based"** — reviewers can't tell what data changed, what downstream breaks, or whether a column rename is safe without manually checking out the branch and running queries. The "semi-confident LGTM" is the canonical failure mode.
4. **Warehouse cost from bad SQL patterns** — top 10% of models burn 65–75% of compute. Cartesian joins, full refreshes on huge tables, redundant scans, oversized window functions, X-Large warehouses where X-Small would do. Engineers don't know which model is the culprit until the bill comes.
5. **Onboarding friction for analysts** — non-engineers must learn SQL + Jinja + Git + PR workflow + CI + `virtualenvs` + dbt CLI to ship a model. Jinja in particular is "harder to read than the worst SQL I've ever seen" per community quotes.
6. **Refactoring legacy SQL / stored procs to dbt** — labor-intensive, error-prone, and the existing tools (Datafold Migration Agent, Altimate's migration skill) are commercial and Snowflake/BQ-centric. Smaller teams hand-port.
7. **Stale model / dead code accumulation** — projects pass 1k models and nobody knows which are referenced. dbt has `deprecation_date` (1.6+) but no automatic detection of orphans, hard-coded refs that

- bypass `ref()`, or models with no downstream consumers. Models persist in the warehouse after deletion from the project.
8. **Breaking-change detection is shallow** — Slim CI rebuilds modified models, but Advanced CI's true row-diff is dbt-Cloud-only. "If I rename `customer_id` to `cust_id`, what breaks?" still requires manual lineage tracing for Core users.
  9. **Incremental model bugs** — schema-drift on remove-column requires `--full-refresh` (silent failure otherwise), `--full-refresh` itself has known broken paths on Redshift/Fabric/Databricks, and there's no way to unit-test the merge step.
  10. **Project-level conventions drift** — naming, folder structure, `ref` vs. source usage, documentation-required-on-public-models. `dbt_project_evaluator`, `dbt-bouncer`, `dbt-checkpoint`, `dbt-coverage`, `SQLFluff` each cover a slice; teams have to wire 3-5 tools together and even then custom rules require Python plugin development.

Other recurring concerns worth noting but lower-tier: dbt Labs commercial direction post-Fivetran (community fear of Cloud-only features), Semantic Layer / MetricFlow adoption is mostly an organizational not technical problem, lineage gaps when Python models or non-dbt tools sit in the DAG.

## 2. Tooling Landscape

Tool	Category	LLM-aware?	Gap it leaves open
dbt Cloud / dbt Studio IDE	First-party SaaS IDE + orchestrator	dbt Copilot (Enterprise tier only)	Locked to Cloud; Core users get nothing; black-box test/doc generation with no eval
dbt Core CLI	First-party local	No	Pure execution; no quality gates, no AI
dbt Fusion Engine	First-party Rust engine (beta May 2025)	LSP enables AI integration	New; feature-parity still in progress; not a UX product
dbt MCP Server	First-party MCP for AI agents	Yes (semantic-layer + discovery API)	Read-only context; doesn't write/refactor code or score quality
dbt-codegen	Codegen package	No	Still requires hand-written descriptions/tests
dbt_project_evaluator	Best-practices linter	No	Static rules only; no AI fix

Tool	Category	LLM-aware?	Gap it leaves open
dbt-bouncer	Convention enforcement	No	suggestions; weak on unused-models detection
dbt-checkpoint	Pre-commit hooks	No	Custom rules require Python; no auto-fix
dbt-coverage	Doc/test coverage stats	No	Only runs at commit; can't run in dbt Cloud
SQLFluff	SQL linter/formatter	No	Reports gaps but doesn't fill them
dbt-utils / dbt-expectations	Test packages	No	Style only; no semantic understanding
Elementary Data	Observability + anomaly detection	ML anomaly detection (not LLM)	Authors still must pick which tests apply
Datafold Cloud	Data diff + column lineage + CI	Migration Agent uses LLMs	Catches issues post-hoc, not at PR time
Recce	PR review / impact validation	No (rule-based diffs)	Commercial; Snowflake/BQ/Redshift only; expensive
Paradime + DinoAI	AI-native managed dbt IDE	Yes (warehouse-aware LLM assistant)	Shows diffs; doesn't explain why or what to do about it
Altimate dbt Power User	VSCode extension + MCP server	Yes (datapilot, MCP for Cursor/Copilot)	Closed SaaS; competes with dbt Cloud directly
dbt-llm-evals (Paradime OSS)	Warehouse-native LLM eval	Yes — judges LLM outputs in-warehouse	Plugin-shaped; depends on user's IDE; quality of generated artifacts is uneven per user reports
dbt-llm-agent (pragunbhutani)	OSS LLM agent for dbt	Yes	Evaluates LLM output inside dbt, not authoring quality
			Early-stage; analysis-focused, not authoring

Tool	Category	LLM-aware?	Gap it leaves open
select.dev	Snowflake cost intel for dbt	No	Cost-only; vendor-locked; doesn't fix the SQL
Seemore Data	Autonomous data-engineer agent	Yes (LLM agent)	Closed beta; broad scope dilutes dbt-specific UX
Monte Carlo	Data observability	LLM agents (Monitoring + Troubleshooting)	Enterprise SaaS; not authoring-focused
Atlan / DataHub / Castor / Select Star	Catalogs + lineage	Limited LLM features	Read-only; outside the dev loop
Hex PR Review	Notebook-based PR review	Some LLM features	Tied to Hex notebooks

**Key observation:** AI-aware authoring tools cluster at two extremes — closed-source SaaS IDEs (dbt Cloud Copilot, Paradime, Altimate Cloud) and early-stage OSS experiments (dbt-llm-agent). The quality-evaluation layer that the cycle should produce — “did the LLM’s generated test/doc/refactor actually improve the project?” — is essentially empty except for dbt-llm-evals (which judges arbitrary LLM outputs, not dbt-authoring quality).

### 3. LLM-Shaped Opportunities (ranked by underservedness × pain)

#### 1. dbt-aware “PR review companion” that explains data + code impact in plain language

Recce shows you the diff; Datafold shows you column lineage; nothing closes the loop with **“here’s what changed, here’s why it’s risky, here’s the test you forgot, here’s the doc that’s now wrong.”** An LLM with the manifest, catalog, and a column-level diff can produce a single PR comment that combines: (a) summary of logic changes, (b) downstream models/dashboards/reverse-ETL syncs at risk, (c) suggested missing tests, (d) doc/yaml drift detected, (e) cost-class change estimate. Open-source-able, runs in CI, doesn’t require dbt Cloud. Incumbents: Recce (no LLM), Datafold (no per-PR narrative), dbt Cloud Advanced CI (Cloud-only, no narrative).

#### 2. Schema.yml + tests + docs generator with quality eval baked in

The “generate yaml from model” features exist (codegen, Copilot, Paradime) but reviewers consistently report that AI-written tests are noise. The differentiator: a tool that drafts → samples warehouse data via dbt MCP → runs candidate tests against real data → drops tests that always pass or are uninformative → emits only signal-bearing tests with a one-line “why” comment. Plus rubric-graded doc generation (clarity, completeness,



terminology consistency). This is the natural place for clauditor-style skill evaluation methodology applied to a domain. Incumbents partial: dbt Copilot (Cloud-only, no eval), Altimate (uneven quality, no scoring).

### **3. Stored-proc / legacy-SQL → dbt refactoring agent (open source)**

Datafold Migration Agent is the leader and it's commercial enterprise-tier. There is room for an OSS LLM agent that: (a) parses procedural SQL, (b) clusters logic into staging/intermediate/mart layers, (c) emits dbt models with `ref()`s, (d) generates `audit_helper`-style row-count/value-diff tests so the migration is verifiable, (e) iterates until parity. Especially valuable because most migrations are one-time events and teams resist paying enterprise SaaS for a finite project.

### **4. Stale-model + tech-debt cleanup agent**

Projects with 1k+ models accumulate orphans, hard-coded refs, never-queried tables, and shadow lineage. A scheduled agent that cross-references manifest + warehouse query history (Snowflake `query_history`, BQ `INFORMATION_SCHEMA.JOBS`) + downstream BI tool metadata can: (a) flag candidates for `deprecation_date`, (b) detect dead-code paths an LLM can confirm by reading the model, (c) propose deletion PRs with safety justifications, (d) drop the matching warehouse relations. `dbt_project_evaluator` does ~10% of this; nothing closes the loop.

### **5. Cost-aware refactor proposer**

`SELECT` and Seemore identify expensive models; the **fix** still falls on engineers. An LLM that takes (model SQL + `EXPLAIN` plan + warehouse spend per run + sample data) and proposes a refactor PR — incrementalize, add a filter, swap CTE for table, push down predicates, downsize warehouse — with a projected cost delta and a row-count diff guardrail. Hooks into Snowflake `query_history` / `dbt-snowflake-monitoring`. The “AI suggests, human merges” loop is unproven here and underserved.

### **6. Onboarding / explainer agent grounded in your project**

Most “explain my model” features are read-only chat (dbt MCP, Paradime DinoAI). The opportunity: a guided onboarding agent that walks a new analyst through your team's project — picks 5 models matched to their domain, explains the lineage in plain English, has them edit a guarded sandbox model, runs the local tests, gives feedback. Reduces “first PR shipped” time from weeks to hours. Adjacent to Anthropic's Skills work — could be packaged as an MCP-discoverable skill that ships with a dbt project.

### **7. Project-evaluator-on-LLM-rails (custom rules without Python)**

`dbt-bouncer` and `dbt_project_evaluator` cover static checks but custom rules require Python. An LLM-backed linter that takes natural-language

rules (“every public mart model must have a description, an owner, a freshness test, and snake\_case columns ending \_at for timestamps”) and enforces them in CI — emitting auto-fix PRs for violations — would meaningfully lower the bar for team-specific governance.

## 4. Strategic Notes for the Builder

- **The MCP server is a moat-shrinker, not a moat.** dbt Labs shipped the MCP server precisely so any AI agent can read project context. Differentiation has to come from what the agent does with that context — quality of generated artifacts, eval rigor, integration depth into PR/CI workflow, OSS distribution.
- **dbt Cloud Copilot is the obvious incumbent for authoring**, but it’s gated behind Enterprise tier and only works in the Studio IDE. A Core-friendly OSS alternative that runs in any IDE (or none) has structural distribution advantages, especially post-Fivetran-merger anxiety driving teams to evaluate Core-native paths.
- **The quality-eval gap is the most defensible angle.** Multiple sources cite “AI generates 100 noisy tests in a minute” as the failure mode. A tool that evaluates what LLMs produce against a project-specific rubric — and only ships signal-bearing artifacts — has no clear competition outside Paradime’s dbt-llm-evals (which judges LLM outputs in production, not authoring loops).
- **PR-review companion + test/doc generator are the highest-ROI starting bets.** Both have demonstrated demand, both are partially served (so users know they want this), both fit naturally in a CI pipeline that doesn’t depend on a SaaS IDE.

## Sources

- [How dbt Can Help Solve 4 Common Data Engineering Pain Points \(dbt Labs\)](#)
- [Top 5 Advanced dbt Anti-Patterns That Nearly Killed Our Analytics Team \(Medium\)](#)
- [The 7 dbt Anti-Patterns Quietly Destroying Your Warehouse Budget \(Medium\)](#)
- [Unit tests \(dbt Developer Hub\)](#)
- [dbt unit testing best practices \(Datafold\)](#)
- [6 Mistakes In Dbt Unit Testing \(Monte Carlo\)](#)
- [Continuous integration jobs in dbt](#)
- [Announcing advanced CI \(dbt Labs\)](#)
- [Two Approaches to dbt Slim CI \(Snowpack Data\)](#)
- [Feature: Command to auto-generate schema.yml files \(dbt-core issue #1082\)](#)
- [SQLFluff GitHub](#)
- [Lint and format your code \(dbt Developer Hub\)](#)
- [Paradime AI for dbt](#)
- [dbt-llm-evals \(Paradime OSS\)](#)
- [Refactoring legacy SQL to dbt \(dbt Developer Hub\)](#)
- [From stored procedures to dbt: A modern migration playbook](#)
- [Datafold + dbt: Ship Better Data](#)

- [Datafold dbt integration](#)
- [Elementary Data GitHub](#)
- [dbt Power User VSCode extension](#)
- [Supercharging Cursor IDE: dbt Power User MCP server \(Altimate blog\)](#)
- [How to reduce Snowflake costs with smart architecture \(dbt Labs\)](#)
- [The Hidden Costs of dbt + Snowflake and How to Fix Them \(Medium\)](#)
- [dbt Mesh GA announcement](#)
- [Intro to dbt Mesh](#)
- [Recce GitHub](#)
- [The Ultimate PR Comment Template for data projects \(Recce\)](#)
- [How the dbt MCP Server connects AI to trusted data](#)
- [dbt-mcp GitHub](#)
- [About dbt Copilot](#)
- [A new era of data engineering: dbt Copilot is GA](#)
- [Meet the dbt Fusion Engine](#)
- [About Fusion \(dbt Developer Hub\)](#)
- [dbt-bouncer GitHub](#)
- [dbt\\_project\\_evaluator](#)
- [Introducing the dbt\\_project\\_evaluator](#)
- [Clean your warehouse of old and deprecated models \(dbt Discourse\)](#)
- [deprecation\\_date \(dbt Developer Hub\)](#)
- [Analyzing your DAG to identify unused dbt models \(select.dev\)](#)
- [SELECT dbt integration](#)
- [How to review an analytics pull request effectively \(dbt Labs\)](#)
- [Code review best practices for Analytics Engineers \(Datafold\)](#)
- [Implement dbt data quality checks with dbt-expectations \(Datadog\)](#)
- [Using AI to build a robust testing framework \(Mikkel Dengsoe\)](#)
- [LLM-powered Analytics Engineering with Snowflake Cortex \(dbt Developer Blog\)](#)
- [Bringing structured context to AI with dbt](#)
- [Optimize and troubleshoot dbt models on Databricks](#)
- [Configure incremental models](#)
- [dbt Just Sold Out \(Reliable Data Engineering, Medium\)](#)
- [dbt-llm-agent \(pragunbhutani\)](#)

## Appendix C — Pain Points Deep Dive

### dbt Pain — Deep Dive into the Lived Experience

A research dossier of direct quotes, failure stories, and indirect signals about what hurts when you actually use dbt in production. Sourced from Hacker News, Substack (Pedram Navid, Benn Stancil, Max Halford), Reddit r/dataengineering, GitHub issues on dbt-core / dbt-snowflake, dbt Discourse, vendor postmortems (Datafold, SYNQ, Elementary, Monte Carlo, Select Star), and the 2024/2025 State of Analytics Engineering surveys.

The structure follows the four activities — **Writing models, Testing, Validating, Maintaining** — then closes with cross-cutting patterns.

---

## 1. Writing Models

### Direct quotes

#### On Jinja templating as a foreign body in SQL:

“Jinja syntax is harder to read and write than the worst SQL some engineers have ever seen.” — recurring r/dataengineering complaint, summarized at [Orchestra: dbt Best Practices Reddit Insights](#)

“Macros are a jinja-powered hot mess of untestable code, yet they remain the predominant way in which any logic outside of SQL is done in dbt.” — Pedram Navid, [We need to talk about dbt](#)

“If you have worked with dbt for more than a week, you have probably encountered a frustrating debugging session where everything looks correct, but your model refuses to compile.” — Alwyn DSouza, [Don't Nest Your Curlies](#)

“Overusing Jinja, such as turning every repeated line into a macro, makes models harder to onboard new engineers to, harder to review in pull requests, and harder to debug.” — same source, summarizing the macros-as-debt failure mode

#### On the syntax-validation gap (no compile errors until you run dbt):

“The year is 2022, and I still need to run dbt before I can catch a syntax error.” — Pedram Navid, [We need to talk about dbt](#)

#### On the ref() macro as ergonomic friction:

“I hate that you have to specify dependencies by hard-coding them with ref(something). I can't just copy/paste my SQL query into my database client and execute it. I have to fiddle about and remove the curly braces.” — Max Halford, [A rant against dbt ref](#)

“When dbt determines the dependencies, builds the execution DAG, and runs the queries in the resulting order, it won't magically indicate you forgot to specify a ref.” — Max Halford, same post — describing silent dependency bugs that surface only when parallel execution races against a stale upstream.

#### On namespace collisions at scale:

“Given a dbt project of sufficient size, odds converge to 1.0 that two tables will want the same name, yet namespaces are not supported in refs.” — Pedram Navid, “We need to talk about dbt”

## On dbt Cloud IDE as a development environment:

“dbt Cloud is a really bad experience... it’s nothing more than a text editor with some syntax highlighting.” — Pedram Navid, “We need to talk about dbt”

“Loading it is exceptionally slow... I’ll end up closing the window and switching to the terminal instead.” — Pedram Navid, same post — also reports losing work when forgetting to switch branches before starting.

“IDE sessions typically take 15-30 seconds to start, with some experiencing upwards of 2 minutes or longer ... [the IDE was] far too slow to start up and respond, with Git actions, saving, renaming and creating files all being terribly slow compared to VSCode.” — summarized from [dbt Labs IDE performance updates blog](#) and dbt Discourse “Your IDE session has timed out” thread.

## On forced bifurcation between SQL and YAML:

“How often have you updated a column in your model and not bothered to edit the schema file because it was too far away?” — Pedram Navid, [dbt Reimagined](#)

## Failure stories

1. **The “spent two hours figuring out why my model wouldn’t compile” pattern.** The Towards Data Engineering Jinja guide opens with the canonical scenario: “everything looks correct, but your model refuses to compile” — nested `{% %}` inside string interpolations, whitespace control characters (`{%- -%}`) silently swallowing trailing newlines that another macro depends on, and `{{ ref('foo') }}` failing because of an invisible BOM. The debug loop is: change one curly, dbt compile, wait, read error, repeat. ([source](#))
2. **“The team renamed a column and broke 14 dashboards over the weekend.”** The DataChef “Missing Right Side of Your dbt DAG” article codifies this: “When you’re editing a dbt model, a seemingly minor change (like renaming a column or updating a WHERE clause) can quietly break dozens of downstream charts. Without a safe place to test both sides together, teams either slow down their pace of development or risk introducing errors into content for end users.” dbt’s DAG sees only what is upstream of the model — it has no visibility into Looker, Mode, Tableau, reverse-ETL syncs, or ML jobs. ([source](#))
3. **The macro time bomb at 47 callers.** The “Your dbt Macros Are Technical Debt Time Bombs” post describes: “real scenarios where macros written 18 months ago were called by 47 models but the original author was no longer available, creating significant refactoring risks.” Nobody is sure what the macro does; nobody is sure what will break if it is changed; the macro stays in place and accumulates a halo of workarounds. ([source](#))

## Indirect signals (what people complain about sideways)

- **SELECT \* aversion via dbt\_utils.star.** Teams reach for `{{ dbt_utils.star(from=ref('upstream'), except=['x']) }}` to “avoid enumerating columns” — the SQLMesh migration writeup notes that this apparent convenience comes with “the maintenance burden, the opacity of the transformations, and the challenges in debugging and optimization,” because the resolved column list is hidden until compile time. ([source](#))
  - **The “I just compile it locally and copy-paste into Snowflake to debug” workflow** is endemic — this is why Max Halford’s `ref()` complaint resonates. The fact that people maintain a shadow workflow outside dbt to actually understand what their models do is itself a complaint.
  - **“Build fast and messy” is the default at velocity-pressured shops.** “Teams often face a trade-off: build dbt models carefully while meeting all best practices, or build them fast and messy, and worry about tech debt later.” The framing of best-practice as “the slow option” is a tell. ([source](#))
  - **The proliferation of “dbt power user” VS Code extensions, dbt-coverage packages, dbt-checkpoint pre-commit hooks, dbt-osmosis, dbt-autofix.** Each one exists because the core developer experience leaves a gap a third-party feels obligated to close.
- 

## 2. Testing

### Direct quotes

#### On why teams skip tests / let them go red:

“Once teams proceed with merging despite a failing test with the reasoning ‘we all know this one test is noisy, so we just been ignoring it,’ they step (and set their team) on a slippery slope of allowing more and more tests to fail, eventually leading to tests not being taken seriously and significant data quality issues slipping through.” — Elementary, [dbt tests: How to write fewer and better data tests](#)

“With more pipelines, models, and tests across the stack, the number of alerts grew. Important alerts were sometimes drowned out by noisy ones, and engineers frequently cited ‘alert fatigue’ as a pain point.” — Elementary, [dbt observability 101](#)

#### On generic tests as low-value noise:

“Some tests are inherently noise, poorly-defined, or redundant. It’s ok to remove a test if you believe it’s not adding incremental value.” — Datafold, [7 dbt testing best practices](#)

“Just because you can write a test doesn’t mean you should.” — same source, on the cargo-culting of `not_null/unique/accepted_values` on every column of every model.

### **On unit-test adoption being near-zero despite dbt 1.8 shipping it:**

“Unit testing arrived in dbt 1.8, but nobody does it. The YAML accounting killed adoption before it started, with writing fixtures by hand, sampling data, and formatting dictionaries creating too much friction. More specifically, developers must figure out which models their test depends on, query the warehouse to get realistic sample data, format everything as YAML dictionaries with the right structure, then do it again for every edge case they want to cover and maintain all of it as models evolve.” — David Rubio, [dbt test coverage: the missing SLI in your data platform](#)

“In SQL there is still no equivalent to test coverage, and even with dbt tests, it’s hard to answer a simple question: How much of SQL logic is actually tested?” — same source — on why teams have no honest measure of whether their tests cover anything load-bearing.

### **On the false comfort of green-test runs:**

“Copy-pasting YAML without understanding what you’re testing is how you end up with a green CI pipeline and broken dashboards.” — Adrienne Vermorel, [Unit Testing dbt Models: Real-World Examples and Patterns](#)

“Incremental models are the most common source of subtle bugs in dbt projects because they have two completely different code paths — one that runs on a full refresh, another that runs during normal incremental loads — and teams often test only one of them. A typical failure mode looks like: you develop your incremental model and it works perfectly in dev, deploy to production and run a full refresh with everything looking great, but three months later someone notices duplicates appearing in downstream reports because the incremental logic had a bug that only manifested after months of daily runs.” — same source.

### **Failure stories**

1. **The Monday-morning ARR doubling.** Select\* From’s “The Cost of Silent dbt Failures” opens with: “A null value in a join key caused a ‘massive fan-out,’ doubling the ‘Total ARR’ figure overnight. The discovery process took until 9:15 AM when the Head of Sales noticed the anomaly. Resolution required three hours of manual debugging through dbt models and SQL code. A simple 5-line YAML test could have caught this on Friday.” ([source](#))
2. **Two and a half weeks of lost data because nobody added a freshness test.** From a TowardsDataScience post on source freshness:



“I’ve dealt with issues in the past where ingestion tools said data was being ingested into the warehouse, only to find out that it wasn’t properly working for over two weeks. This resulted in two and a half weeks of lost data! If I had added freshness tests at the source earlier, this would have never happened.” dbt’s source freshness was always available — the team simply didn’t wire it in, because it isn’t on by default. ([source](#))

3. **Half-populated incremental table going green for months.** “When a source system adds a new column, the incremental model keeps running every hour and everything shows green with dashboards looking fine, but the table becomes split — half complete with the new data and half incomplete without it. The model doesn’t fail and keeps adding data while ignoring the new column, so data checks might never catch this since nothing technically broke — you just stopped capturing important business information.” ([source](#))
4. **Unique-key duplicates that survive unique tests.** The dbt-core issue tracker has a long-standing class of bugs (issues [#7873](#), [#7597](#), [#5691](#)) where incremental models with NULL values inside the unique key silently append duplicates. The downstream unique test then either passes (because the test runs after the dedup the developer assumed was happening) or fails opaquely days later when the NULL pattern shows up in production data. The dbt Discourse thread “incremental model + unique constraint still allows duplicates” runs into this from the user side. ([discourse](#))

## Indirect signals

- **severity: warn** is a feature whose existence signals “tests fail too often to block CI.” A team that sets severity: warn on a unique test has effectively decided the test is wrong but hasn’t yet decided what right looks like.
  - **Elementary, Monte Carlo, SYNQ, re\_data, Datafold all sell “data observability” — i.e. the layer that catches what dbt tests miss.** The market exists because the in-tree test surface is acknowledged insufficient. Mikkel Dengsøe (SYNQ) frames this directly: “A dbt model failing may be an early warning sign of a trivial issue or indicate that your entire pipeline is down, and context is needed to make this call.” ([source](#))
  - **The 2024 State of Analytics Engineering** found “poor data quality emerged as a predominant issue for 57% of professionals, an increase from 41% in 2022.” Two-plus years into widespread dbt adoption, data-quality complaints are getting worse, not better. ([source](#))
  - **Test running cost as a hidden tax.** “dbt tests (especially uniqueness, not\_null, and relationships) generate SQL queries that scan full tables, and running tests on billion-row fact tables daily can burn significant credits.” Teams disable expensive tests for cost, leaving big tables least-covered. ([source](#))
-



### 3. Validating (CI / Breaking-Change Detection)

#### Direct quotes

##### On Slim CI's structural blind spot:

"With Slim Diff configuration, downstream models will be prevented from running unless they have been designated as exceptions with the `slim_diff: diff_when_downstream` dbt meta tag. This suggests that care must be taken to identify which downstream models truly need to be tested, as the default behavior may not catch all potential breaking changes that could escape to production." — Datafold, [Slim CI: Cost-Effective Solution](#)

##### On dbt CI testing syntax not behavior:

"A column rename or semantic logic change can still break a dashboard even if the exposure compiles." — DataChef, [The Missing Right Side of Your dbt DAG](#)

##### On Datafold pricing as the "real CI" tax:

"Datafold's Cloud tier starts at \$799/month when billed annually. For larger deployments, small to mid-sized teams (5-15 data sources, <10TB data) typically pay annual contract values ranging from \$30,000-\$75,000 for cloud-hosted deployments." — [Vendr Datafold listing](#)

"Datafold has a commercial subscription requirement, which posed challenges for adopting and using the tool effectively." — G2 reviewer summary, [Datafold Reviews](#)

##### On dbt Cloud Advanced CI as the bundled-but-priced version:

"Over the last 6 months, dbt Cloud pricing has changed drastically, with a 100-700% increase from Dec-2022, followed by new 'Shiny New Pricing' that will cut even deeper holes into already stretched analytics budgets." — Paradime, [What's the new dbt Cloud price increase about? — Part 2](#)

##### On the structural CI gap dbt cannot close on its own:

"If you maintain a dbt project long enough, you end up with a problem: you can see what's upstream of a model, but it is surprisingly hard to answer what is downstream in the real world. Which dashboards, reports, ML jobs, or extracts depend on this thing, and who owns them?" — DataChef, "Missing Right Side"

#### Failure stories

1. **The exposure compiles, the dashboard breaks anyway.** The Omni community tells the canonical version: a dbt model edit passes dbt

build, all unit tests pass, all data tests pass, the PR merges, the next morning the Looker dashboard renders zeros because a column semantic changed (e.g., revenue switched from cents to dollars; the column name didn't change). dbt's contract enforcement caught nothing because the schema didn't change. ([source](#))

2. **The “we don't actually test data in CI, we test SQL syntax” admission.** Slim CI's value prop — “only build modified models and downstream of modified models” — is itself a workaround for the fact that running the full DAG in CI is too slow / expensive. The trade-off is that you're now CI-testing a fraction of the project against a production manifest of unchanged parents — which means the PR runs against production data the developer hasn't seen, and the feedback loop is “did the SQL parse and execute?” rather than “did the data come out right?”. Datafold's Slim Diff and dbt's Advanced CI exist as paid layers on top to add row-level diffing because Slim CI alone gives you syntactic comfort and semantic blindness.
3. **The first-run incremental policy bug.** “If there is an error in your incremental policy, the first run will still be successful and dbt will not throw an error until the second run. This can be especially troublesome for models run overnight once per day — the engineer will manually test their model during business hours thinking it works, then the stakeholder gets a failure message after hours saying the data is now stale.” ([source](#))
4. **Breaking-DDL detection is missing.** dbt-fusion issue [#1532](#) (“Feature Proposal: Safe Incremental Rebuild with Automatic Breaking DDL Detection”): “When a developer makes structural changes to an incremental model — reordering columns, inserting a column in the middle, renaming a column, or changing incompatible data types — dbt has no automatic recovery path and the run either errors at runtime or produces silent data corruption.” This is open in 2026 — i.e., the foundational tool still doesn't detect a class of breaking change that ships to prod weekly.

## Indirect signals

- **The fact that “Slim CI vs. Advanced CI vs. Datafold vs. re\_data vs. Synq vs. SaaS-of-the-week” is even a conversation** is a tell that no one tool gives operators what they need from a CI run. Teams stack three or four because each catches a different blind spot (syntax, schema, row-diff, semantic, freshness).
- **dbt Mesh model contracts are themselves a symptom.** The product literature says contracts are about “graceful evolution”; the practical use is “stop my downstream consumers from yelling at me when I change a column.” The fact that contracts are an opt-in advanced feature rather than the default means most projects ship without them, and the breaking-change detection story degrades to “tribal knowledge + Slack apology.”
- **The “no announcement, just a blog post” rollout of consumption-based pricing** (per Pedram Navid) signals that even the vendor treats

CI cost as something to be quietly billed for, rather than a first-class capability.

---

## 4. Maintaining

### Direct quotes

#### On stale model graveyards:

“dbt models persist in your production database even after they’re deleted from your project, adding clutter to the warehouse and potentially slowing down operations like database or schema cloning.” — summarized in the [dbt Discourse “Clean your warehouse of old and deprecated models”](#) thread.

#### On the macro time-bomb again, this time from a maintenance angle:

“Macro complexity grows exponentially with team size and project age, where what seems clever at 50 models becomes unmanageable at 500 models.” — [Reliable Data Engineering](#)

#### On Snowflake cost explosions caused by dbt:

“Snowflake Ate My Budget: The ‘Quick’ Query That Turned Into an \$18K Surprise ... a single ad-hoc query that ran for almost 5 hours, scanned hundreds of terabytes, and burned \$18,300 in credits since midnight — a week’s worth of spend in just a few hours.” — Abhishek Kumar Gupta, [Snowflake Ate My Budget](#)

“Storing raw, intermediate, and historical tables for years can silently bloat costs, and dbt’s habit of generating ‘backup’ or ‘snapshot’ tables causes storage charges to creep up. ... Nested models, redundant CTEs, and wide joins can cause Snowflake to re-scan the same data multiple times, with dbt DAGs sometimes looking clean but having wasteful queries under the hood.” — Manik Hossain, [The Hidden Costs of dbt + Snowflake](#)

“We didn’t realize dev environment refreshes counted as runs.” — anonymous data team lead, recounted in same article — re: dbt Cloud consumption-based billing surprise.

#### On onboarding pain:

“Onboarding business people/analysts to dbt requires teaching them SQL, dbt, command line, git, pull requests, CI, and unless you are on dbt Cloud it also requires a basic knowledge of python environments.” — r/dataengineering, summarized at [Orchestra](#)

“When analysts first encounter dbt, they often feel confused and uncertain, and dbt’s technical terminology can induce impostor syndrome, even for those who know SQL and data modeling

concepts.” — dbt Labs’ own [“Learning dbt as an analyst”](#) blog post — a tell that even the vendor acknowledges the cliff.

“Setting up a DBT project is challenging, and managing versioning, syntax, spacing, and linting across developers is a nightmare.” — recurring r/dataengineering complaint ([source](#))

### **On scale beyond ~500 models:**

“<500 models ... reflects the point at which dbt Labs’ own internal analytics project went from feeling ‘manageable’ to ‘there’s too much going on.’” — [dbt Mesh: Who is dbt Mesh For](#) — i.e., the vendor’s own bar for “this got hard” is 500 models, and “over the past year, the number of ‘large’ projects (>500 models) has tripled.”

### **On the Cloud price increases as a maintenance cost shock:**

“dbt increased the price of dbt Cloud by 100% on December 15, 2022 on non-enterprise pricing tiers, though the product had not materially changed or was not necessarily providing more value with the price increase. ... With the Team tier and 8 seats, customers would now be paying 225% more than before.” — Paradime, [What’s the new dbt Cloud price increase about?](#)

### **On debugging at scale:**

“Debugging in dbt can be less straightforward than in other environments, with errors propagating through the transformation pipeline and making pinpointed debugging tedious.” — recurring complaint, summarized at [Orchestra: Downsides of dbt](#)

“Debugging tasks are difficult because error messages lack clarity, resulting in analysts spending a lot of time on logs.” — same source

### **Failure stories**

1. **The 6-8 hour daily refresh.** “A dbt Cloud user was spending 6-8 hours per day on pipeline refreshes before migrating to dbt.” The framing is a success story (they got out of stored procs) but the reality of the pre-state — most of a workday spent watching pipelines — is the maintenance cost of the legacy approach dbt was meant to displace, and many of those teams now spend the equivalent on dbt full refreshes. ([source](#))
2. **The “we lost an analyst” pattern.** The recurring r/dataengineering complaint is that analytics-engineering onboarding now requires SQL + Jinja + YAML + git + PRs + CI + virtual envs. Every additional layer is a place where a junior analyst hits a “why doesn’t this work” wall, has nobody to ask, and decides they’d rather be in the BI tool. The talent attrition isn’t from dbt failing; it’s from dbt being a five-tool job advertised as one tool.

3. **The full-refresh outage.** dbt-core issue [#12467](#) (“[Bug] full-refresh broken for too large data”) — running `dbt run --full-refresh` on roughly 5 million records in 35,000 partitions resulted in `HIVE_PATH_ALREADY_EXISTS` after a long time, because temporary table directories already existed. The dbt Discourse “How to prevent (accidental) full refreshes” thread (issue [#1008](#)) exists because operators have learned the hard way that `--full-refresh` on the wrong night brings the warehouse to its knees and leaves a half-built table behind.
4. **The 4-minute dbt deps.** Issue [#11479](#): “dbt deps runs very slow, each request waits 4 min before receiving response.” Multiplied across CI runs, dev rebuilds, and onboarding sessions, this is a per-team-per-day tax in pure waiting.
5. **The 26% partial-parse regression.** Issue [#10127](#): “[Performance] 1.8 slower partial parsing than 1.7.” Going from 1.7.3 to 1.8.0 raised partial-parse from 6.00s to 7.58s — a small absolute number that, multiplied by every dev iteration of every analyst at scale, becomes hours of waiting per week. Even the Fusion-engine marketing leans on this: “After years of community complaints about dbt Core’s performance bottlenecks in larger projects, dbt Labs has delivered a solution.” ([source](#))

## Indirect signals

- **The existence of dbt-autofix** — a CLI dbt Labs ships explicitly to walk projects and fix deprecations automatically — admits that the upgrade tax is high enough to need its own automation.
  - **The dbt-coverage package** exists because operators want a measurable answer to “how much of our SQL is tested,” and core dbt doesn’t supply one.
  - **The dbt Mesh product is itself an indirect signal:** the “graceful way to scale beyond a single repo” is necessary because a single repo beyond ~500 models stops working. The 2024 data shows “the number of ‘large’ projects (>500 models) has tripled” — i.e., the number of projects in the failure zone is growing fast and dbt’s prescription is “split into many projects with cross-project contracts,” which trades one complexity for another.
  - **The Fusion engine ships under Elastic License 2.0, not Apache 2.0.** Adapter ecosystem fragmentation (“Fusion adapters must be written in Rust rather than Python, making existing Core adapters incompatible without complete rewrites ... For community contributors who may lack Rust expertise, this represents a significant barrier to participation”) signals the maintenance burden is moving from users to adapter maintainers, who may not be there to take it. ([source](#))
-

## 5. Patterns Across All Four

**What experienced dbt users complain about that newcomers haven't hit yet**

- **Silent dependency bugs from forgotten `ref()`s.** A newcomer reads every line of every model. A team with 800 models cannot — they trust the DAG. The DAG is only as complete as the `ref()` discipline at every keystroke, and there is no compile-time enforcement.
- **The macro time-bomb.** At 50 models, macros feel clever. At 500, the original author left, every model touches one of three macros, and nobody can refactor without breaking forty things.
- **Incremental-model edge cases.** The full-refresh path and the incremental path are two implementations of the same logic. They drift. The first time you notice is when the sales team asks why ARR halved between Tuesday and Wednesday.
- **The dev/prod cost asymmetry.** A junior runs `dbt build` in dev and burns nothing they can see. The bill arrives at the end of the month. By the time the team installs `dbt-snowflake-monitoring` and query tags, the spend chart already has a step function in it.
- **The Mesh tax at scale.** Teams that “got it working” with a monorepo + dbt Cloud + Slim CI hit a wall around 500-800 models and discover that the next mile requires Mesh, contracts, model versions, cross-project orchestration, and a tax on every change.

**What pain has been “solved” by tooling but still shows up because adoption is hard**

- **Unit tests** (dbt 1.8) — solved on paper, “nobody does it” in practice because the YAML fixture authoring is too painful and there is no equivalent to test coverage to nag you about gaps.
- **Source freshness** — has been in dbt forever, requires opting in per source, and remains the “we lost two and a half weeks of data because we never wired it up” story.
- **Model contracts** — solved the schema-drift case for opted-in models, but require explicit declaration that most teams skip during initial development “to ship faster.”
- **Column-level lineage** — exists in dbt Cloud Explorer and several third-party catalogs (Select Star, Atlan, Datahub, Castor), but requires the catalog to be wired in and refreshed; the “what dashboards depend on this column” question still routinely takes a Slack thread to answer.
- **Slim CI** — solved the “don't rebuild the world for every PR” problem, but in doing so introduced a new problem (you're CI-testing against a manifest you didn't author, on data the developer can't see), which sells the row-level-diff layer (Datafold, Advanced CI).

**What pain is structural to dbt's design philosophy**

- **SQL-as-text + Jinja-on-top.** Because dbt treats SQL as a string the Jinja engine renders, it cannot in general parse the result for references, type-check across columns, or detect breaking renames at



compile time. Fusion (Rust + a real SQL parser) is dbt Labs' acknowledgment of the cost — "Unlike dbt Core, which treats SQL as templated text, Fusion parses and understands SQL syntax and semantics across data platforms" — but the install base of Core-and-Jinja still owns the long tail.

- **ref() as the dependency anchor.** Max Halford's rant cuts to it: the explicit `ref()` is the only thing connecting your DAG to reality, and forgetting it produces silent breakage. SQLMesh (and Fusion) auto-detect dependencies from parsed SQL because the manual contract is the wrong default. dbt cannot change this without breaking every project ever written.
- **YAML configuration alongside SQL files.** The bifurcation between `model.sql` and `_model.yml` is the source of every "the docs are out of date" complaint, every "I forgot to add the column to the schema" failure, every "the test was on the old name" outage. Pedram Navid: "How often have you updated a column in your model and not bothered to edit the schema file because it was too far away?"
- **Tests run after the model materializes.** The test executes against the result table, not against the SQL. So the test can only catch data shapes, never code intent. That is why unit tests (introduced in 1.8) had to be a whole new construct — and why nobody uses them.
- **No first-class downstream awareness.** dbt is a pre-warehouse tool; it does not know about Looker, Tableau, Mode, Hex, reverse-ETL syncs, ML feature stores, or anything that reads its output. Exposures exist as a manual workaround. "What breaks if I rename this column?" is structurally unanswerable inside dbt; that is the entire reason observability vendors exist as a category.
- **Cost surfaces at the warehouse, not at dbt.** dbt has no native notion of "this model is expensive" until you wire in `dbt-snowflake-monitoring`, `dbt-bigquery-monitoring`, or a `query-tag`
  - per-warehouse dashboard. By design, dbt produces SQL and then walks away. By consequence, cost surprise is the single most common "we didn't realize" complaint in the entire ecosystem.
- **The dbt Cloud commercial trajectory.** Every structural pain above is an opportunity to sell a paid layer (Advanced CI, Explorer, Mesh + governance, Cost Insights, Fusion engine licensing). The 100-700% price increase between Dec 2022 and 2024 reads, in this light, as the monetization of pains the open-source layer was always going to leave on the table — and the community reaction (Pedram Navid's posts; Tristan Handy's response post; SQLMesh's rise; Fusion's Elastic License 2.0) reflects an ecosystem that is no longer convinced the vendor and the user incentives are aligned.

---

## Source index

- Pedram Navid, [We need to talk about dbt](#)
- Pedram Navid, [dbt Reimagined](#)
- Pedram Navid, [What the hell is going on with data?](#)
- Tristan Handy, [The response you deserve!](#)
- Benn Stancil, [How dbt fails](#)
- Max Halford, [A rant against dbt ref](#)

- Christophe Oudar, [dbt Fusion: The Double-Edged Sword](#)
- Hacker News, [I can't argue that dbt isn't great...](#)
- Hacker News, [Ask HN: What Is the Point of Dbt?](#)
- David Rubio, [dbt test coverage: the missing SLI in your data platform](#)
- Adrienne Vermorel, [Unit Testing dbt Models: Real-World Examples and Patterns](#)
- Elementary Data, [dbt observability 101](#)
- Elementary Data, [dbt tests: How to write fewer and better data tests](#)
- Elementary Data, [My dbt test failed - now what?](#)
- Datafold, [7 dbt testing best practices](#)
- Datafold, [Slim CI: The Cost-Effective Solution for Successful Deployments](#)
- Select\* From, [The Cost of Silent dbt Failures](#)
- DataChef, [The Missing Right Side of Your dbt DAG](#)
- Mikkel Dengsøe, [Learnings from running hundreds of data incidents at SYNQ](#)
- Manik Hossain, [The Hidden Costs of dbt + Snowflake](#)
- Abhishek Kumar Gupta, [Snowflake Ate My Budget](#)
- Paradime, [dbt Cloud price increase analysis](#)
- Reliable Data Engineering, [Your dbt Macros Are Technical Debt Time Bombs](#)
- Alwyn DSouza, [Don't Nest Your Curlies: Jinja in dbt](#)
- Darshan Pathak, [dbt Incremental Models Can Quietly Break Your Data](#)
- Ojo Joseph, [Solving a Silent Bug in dbt](#)
- Harness, [Transitioning from dbt to SQLMesh](#)
- The Data Prism, [dbt Fusion vs dbt Core: A Complete Comparison \(2025\)](#)
- Orchestra, [dbt Best Practices: Insights from the Reddit Community](#)
- Orchestra, [Downsides of dbt: Challenges & Solutions](#)
- dbt-core issue [#12467](#) — full-refresh broken for too large data
- dbt-core issue [#11479](#) — dbt deps runs very slow
- dbt-core issue [#10127](#) — 1.8 slower partial parsing than 1.7
- dbt-core issue [#7873](#) — incremental model duplicates with NULL unique keys
- dbt-core issue [#7597](#) — duplicates if any unique\_key fields are null
- dbt-fusion issue [#1532](#) — Safe Incremental Rebuild with DDL Detection
- dbt Discourse, [incremental model + unique constraint still allows duplicates](#)
- dbt Discourse, [Clean your warehouse of old and deprecated models](#)
- dbt Discourse, [How to prevent \(accidental\) full refreshes](#)
- dbt Labs, [Learning dbt as an analyst](#)
- dbt Labs, [The 2024 State of Analytics Engineering](#)
- dbt Labs, [Updates to improving the dbt Cloud IDE performance](#)
- BigDataWire, [Data Quality Getting Worse, Report Says](#)
- TowardsDataScience, [How Fresh Are Your Data Sources?](#)
- Stellans, [dbt Project Structure Conventions](#)
- Vendr, [Datafold pricing listing](#)
- G2, [Datafold reviews](#)
- Omni Community, [Testing dbt changes in Omni before pushing to production](#)
- dbt Labs, [Stored procedures to dbt: a modern migration playbook](#)
- dbt-core docs Discussion, [Multi-project collaboration #6725](#)
- dbt Mesh docs, [Who is dbt Mesh for?](#)



# Appendix D — AI/LLM Tools Landscape

## The dbt AI/LLM Tooling Ecosystem — Deep Dive (April 2026)

**Scope.** Every meaningful AI/LLM-powered tool in or adjacent to the dbt ecosystem as of April 2026 — what it does, who’s actually using it, what users complain about, and where the gaps are. Written in the wake of the Fivetran ↔ dbt Labs merger (announced Oct 13, 2025), the dbt Fusion engine GA push, the Anthropic Skills standard adoption (Dec 2025), and Snowflake Cortex Code GA (Mar 9, 2026).

### 0. Executive summary

Three things happened between mid-2024 and now that re-shaped the entire competitive map:

- dbt Labs internalized AI as a platform feature.** dbt Copilot (GA Mar 2025), dbt Insights’ Analyst agent, the official dbt-mcp server, and dbt-agent-skills (430 stars, published via the Anthropic Skills standard) make dbt Labs the center of gravity for “AI on a dbt project.” Pre-2025, this surface area was almost entirely owned by Paradime and Altimate.
- The Fivetran merger** turned the AI story into an ingestion-to-activation play, leaving daylight underneath at the quality of the AI output layer — which essentially nobody owns.
- MCP + Skills won as the integration substrate.** Every serious tool now ships an MCP server (dbt Labs, Altimate, Cube, Monte Carlo, Ragstar, Snowflake Cortex Code) and many ship skills. The differentiation has moved from “do you have AI features” to “is the AI output any good.”

The gap nobody fills: **systematic evaluation of LLM output quality on dbt artifacts** (docs, tests, models, semantic layer definitions). Paradime’s dbt-llm-evals is the only OSS attempt and it’s at 25 GitHub stars. Everyone else ships AI features and asks users to eyeball the result.

### A. Comparative matrix

Tool	What it does	Who it’s for	LLM features	Pricing
<b>dbt Copilot</b> (dbt Labs)	Inline assistance + Developer agent inside dbt Cloud Studio IDE, Canvas, Insights	dbt Cloud customers (Starter+)	Code/docs/tests/semantic-model gen; refactor; multi-step Developer agent	Bundled w Starter (\$100/seat mo), Enterprise

Tool	What it does	Who it's for	LLM features	Pricing
				Enterprise BYOK on Enterprise tiers only
<b>dbt-mcp</b> (dbt-labs/dbt-mcp)	MCP server exposing dbt Cloud + Core + Fusion to AI agents	Anyone using Claude/Cursor against a dbt project	~50 tools across SQL, Semantic Layer, Discovery, CLI, Admin, codegen, LSP, Docs	OSS Apache-2.0
<b>dbt-agent-skills</b> (dbt-labs/dbt-agent-skills)	Anthropic-Skills format markdown packages teaching agents dbt best practices	Claude Code, Cursor, Codex, Factory, Kilo Code users	Skills auto-load on prompt match (not slash commands); includes evals/ A/B testing harness	OSS
<b>Paradime DinoAI</b>	"Cursor for Data" — full IDE with inline AI, voice, .dinoprompts, Mermaid, GitHub PR mgmt	Teams that don't want dbt Cloud or want a richer IDE	Inline gen, agentic refactor, cost-aware context truncation, voice-to-text, .dinoprompts library	\$25/user/month (infrequent); \$55/user/month (standard); Enterprise custom; DinoAI was preview-bundled, now add-on
<b>Altimate dbt Power User</b> (AltimateAI/vscode-dbt-power-user)	VSCoDe/Cursor extension: lineage, AI docs, health checks, cost est., embedded MCP server	Local-IDE-first dbt Core teams	AI docs gen, AI test gen, SQL translation, embedded MCP for Cursor	Free OSS extension; "Altimate Code" SaaS (10M tokens of Sonnet/Opus/GPT-free)

Tool	What it does	Who it's for	LLM features	Pricing
<b>Datafold Migration Agent</b>	AI-powered SQL translation + cross-DB diff for legacy → dbt and dbt → dbt repointing	Enterprises migrating from Informatica/SSIS/Talend/stored procs to dbt; Snowflake → Databricks	LLM SQL translation handles 300k-char stored procs; cross-DB diffing for parity validation	<b>Outcome-based:</b> fixed price per legacy object + complexity tier; “no hourly billing, no scope creep”
<b>Snowflake Cortex Code</b>	AI coding agent inside Snowsight; supports dbt + Airflow workflows	Snowflake-first shops	Scaffold dbt models, add tests, run dbt commands, generate docs; Cortex Analyst for NL→SQL; warehouse-native judge models	Bundled with Snowflake credits
<b>Hex Magic</b>	Notebook-native AI generating chained SQL/Python/Chart cells	Analysts in Hex notebooks	NL→SQL, auto-doc, error fix, Magic cell chains; consumes dbt Cloud metadata	Bundled (incl. free tier)
<b>Cube MCP Server</b>	MCP exposing the Cube semantic layer to AI agents	Teams using Cube for semantic layer	NL questions answered against governed metrics	Cube pricing (free OSS)

Tool	What it does	Who it's for	LLM features	Pricing
			with chart + summary	Core, paid (Cloud)
<b>Monte Carlo MC Agent Toolkit</b>	MCP server exposing data observability state to coding agents	dbt teams already on MC	"Asset Health" skill answers "how is table X doing" with monitor coverage + lineage health	Monte Carlo enterprise pricing
<b>Wobby / Actian AI Analyst</b>	NL→SQL agents on top of warehouse + dbt	Business users on Slack/Teams	Agentic NL→insight, semantic-layer-aware, governance hooks	Acquired by HCLSoftware (now Actian) enterprise pricing
<b>Ragstar / dbt-llm-agent</b> (pragunbhutani/dbt-llm-agent)	Self-hosted AI data analyst: web dashboard + Slack + MCP server	Teams that want to self-host	RAG over dbt models with pgvector; OAuth-authenticated MCP for Claude.ai	OSS (self-host)

Tool	What it does	Who it's for	LLM features	Pricing
<b>dbt-llm-evals</b> (paradime-io/dbt-llm-evals)	Warehouse-native LLM-as-a-judge eval framework for AI output in dbt projects	Teams shipping AI-generated content (support replies, classifications, etc.)	Snowflake Cortex / Vertex AI / Databricks AI Functions as judge; 5 built-in dimensions (accuracy, relevance, tone, completeness, consistency); baseline detection	OSS
<b>Elementary ai_data_validation</b>	Plain-English AI-powered data tests in dbt	Existing Elementary users	Test by natural language (“There should be no contract date in the future”); warehouse-native	OSS + Elementary Cloud
<b>dbt-coves</b> (datacoves/dbt-coves)	CLI codegen for sources/staging/yml; AI-assisted via OpenAI/Anthropic/Azure/Gemini	Datacoves customers + OSS users	LLM-assisted yml + staging gen; Airflow DAG gen	OSS
<b>dbt-codegen</b> (dbt-labs/dbt-codegen)	dbt-Labs Jinja macros for boilerplate	All dbt users	None — pure macros	OSS
<b>GitHub Copilot / Cursor / Claude Code (general)</b>	Generalist AI coding agents used on dbt repos	All devs	Whatever the agent supports — code completion, chat, agent mode	Per-seat (\$10-39/se mo for Copilot Business/Enterprise; Cursor \$20/seat; Claude Code via Anthropic API/sub)

---

## B. The “incumbent map” — jobs-to-be-done

For each common AE job, who’s the leader, who’s challenger, who’s nobody’s-yet:

<b>Job-to-be-done</b>	<b>Leader</b>	<b>Challengers</b>	<b>Nobody-yet / weak coverage</b> “Generate schema.yml and verify it round-trips against dbt parse” — every tool ships generation; almost nothing ships verification <b>Useful tests.</b> Universal complaint that AI generates trivial not_null/unique noise (see C below). No tool reliably proposes tests that catch real bugs <b>Verifiably accurate docs.</b> Generic-sounding output is the dominant complaint Cross-project explanations that respect mesh boundaries Mid-complexity (hundreds of models, no migration budget) — Datafold overfits high-end,
<b>Generate schema.yml from model</b>	dbt Copilot (in-Cloud), Altimate Power User (in-VSCode)	Paradime DinoAI; dbt-coves (CLI); Loblaw Digital’s pattern (Cortex-based)	
<b>Generate dbt tests</b>	dbt Copilot; Altimate Power User	Paradime DinoAI; Elementary ai_data_validation (NL → warehouse-native AI tests)	
<b>Generate model docs</b>	dbt Copilot; Paradime DinoAI; Altimate	dbt-coves; Hex (in-notebook); Loblaw blog post pattern	
<b>Explain a model in plain English</b>	Hex Magic (in-notebook); Cortex Analyst	dbt Insights Analyst agent; Wobby/Actian; Ragstar	
<b>Refactor SQL → dbt</b>	Datafold Migration Agent (best at scale); Claude Code (best for ad-hoc); Cortex Code	Cursor; Paradime DinoAI; Altimate skill	

<b>Job-to-be-done</b>	<b>Leader</b>	<b>Challengers</b>	<b>Nobody-yet / weak coverage</b> Claude Code under-fits scale LLM-mediated breaking-change explanation (not just detection) dbt-aware PR review that comments on semantic-layer drift, mesh-contract violations, materialization regressions <b>Wide open.</b> Any tool that combines manifest + query history could do this; nobody productizes it Specific spend-attribution tied to model lineage with LLM-narrated remediation Project-tour generators (pull manifest + docs → guided walkthrough) LLM-narrated impact analysis ("if I drop column X, here's what breaks and why")
<b>Detect breaking changes</b>	Datafold (the diff product, not the agent)	dbt Cloud's CI checks; SQLMesh's intrinsic versioning	
<b>Write the PR review comment</b>	Paradime DinoAI's GitHub PR actions; CodeRabbit (general); Claude Code	dbt Copilot Developer agent	
<b>Detect stale/dead code</b>	None purpose-built for dbt	Manual --select introspection + dbt-project-evaluator	
<b>Cost optimization advice</b>	Altimate Power User (cost estimation feature); Cortex Code	dbt Labs cost-optimization features (Fusion-powered)	
<b>Onboarding new users to a project</b>	dbt Insights Analyst agent; Wobby; Ragstar; Hex Magic	Snowflake Cortex Analyst	
<b>Lineage Q&amp;A</b>	dbt-mcp (get_lineage, get_column_lineage); Atlan; SelectStar	Altimate Power User column lineage; Monte Carlo MC Agent Toolkit	
<b>Test quality evaluation</b>	<b>paradime-io/dbt-llm-evals</b> (only entrant)	DeepEval (general LLM eval, not dbt-specific); dbt-agent-	<b>Almost nobody.</b> This is the gap

Job-to-be-done	Leader	Challengers	Nobody-yet / weak coverage
		skills' eval/ A/B harness	

---

## C. Quality of LLM output — what users actually say

The single most consistent finding across every source: **AI-generated tests and docs are noisy, generic, or trivially wrong, and users have learned to treat them as drafts.**

Specific patterns from the substack reviews, dbt Developer Blog, Reddit, GitHub issues, and Paradime/Altimate's own marketing copy:

- **Generic docs.** "AI-generated documentation can be effective with the right approach. AI can generate accurate definitions based on the column names, data types, upstream sources, and transformations applied. The key differentiator is context." — restated everywhere, but the implicit admission is that without rich context (warehouse schema + lineage + macros), output is generic.
- **Trivial tests.** From Mikkel Dengsoe's substack and the dbt blog: "Don't blindly accept tests but instead spend time reviewing each of the suggestions... riff with the AI but chip in with your own knowledge and context." Translation: the default output is not directly mergeable.
- **The slop tax.** Industry-wide quote that applies directly: "If an AI reviewer posts 20 comments and 18 of them are trivial style suggestions, developers will start ignoring all 20 — including the 2 that catch real bugs." This describes dbt-Copilot-generated test suites in practice.
- **Hallucinated table names.** The Feb 2026 index.app ranking explicitly notes that all three top tools (Claude Code, Cursor, Copilot) "hallucinate table names" without proper MCP/skills grounding. The fix is the dbt-mcp server + dbt-agent-skills, not better models.
- **Stability over quality.** For Altimate dbt Power User specifically, the dominant GitHub-issue complaint is not about AI quality — it's about extension crashes ("Extension host terminated unexpectedly 3 times within the last 5 minutes," issue #1798), endless parse loops on dbt 1.11.2, and file-watcher misses inside dev containers. Quality matters less when the tool isn't running.
- **dbt Copilot is well-reviewed inside official channels.** Customer quotes from dbt Labs ("Cody McLean, Sr. Data Engineer at Hard Rock Digital: dbt Copilot has completely changed how we approach documentation and query optimization") are positive but vendor-curated. Independent Reddit/HN coverage is sparse — surprisingly so given dbt's profile.
- **Semantic-layer recommendations rate higher than test-gen.** Multiple secondary sources say the semantic-model and metric gen quality in dbt Copilot is "perhaps the highlight" — better than the test/docs generation. This is consistent with the pattern that AI does best when the schema constrains the search space.



## The verdict from the Feb 2026 index.app ranking, paraphrased:

Claude Code wins for backend dbt refactors (terminal-native, multi-file changes), Cursor wins for dashboard UIs, Copilot wins for enterprise speed-to-merge (55% faster task completion, 9.6→2.4 day cycle time). For pure dbt work, Claude Code + dbt-mcp + dbt-agent-skills is the recommended combo.

---

## D. Architectural patterns

The eight visible distribution shapes, with pros/cons:

Shape	Examples	Pros	Cons
<b>MCP server (local)</b>	dbt-mcp, Altimate embedded MCP, Cube MCP, Monte Carlo MC Agent Toolkit	Composable; works with any MCP host; easy to ship as uvx / npx	Each user installs separately; tool discoverability lives outside the server
<b>MCP server (remote/managed)</b>	dbt-mcp remote, Snowflake Cortex, Ragstar	No install; auth handled centrally; works from web agents	Network latency; vendor lock-in for the auth surface
<b>Anthropic Skills (markdown)</b>	dbt-agent-skills (13 skills), Altimate's Claude Code Skills, dbt Model Generator	Encodes workflow knowledge, not just APIs; auto-loads on prompt match; portable across 30+ agents	Quality entirely depends on the prose; no enforcement mechanism — the skill can say “always X” but the agent may ignore
<b>VSCode/Cursor extension</b>	Altimate dbt Power User, dbt Power User (innoverio fork)	Lives where AEs already work; can ship UI panels (lineage, lineage diff)	Stability burden — every dbt version bump can break parsing; Microsoft owns the marketplace
<b>CLI tool</b>	dbt-coves, dbt-codegen, Cortex Code CLI	Scriptable; works in CI; no IDE coupling	Not where most AEs live day-to-day
<b>Web SaaS IDE</b>	Paradime, dbt Cloud Studio, Hex	Lock-in is a feature for buyers; tightest integration with own backend	Deep moat to displace; users skeptical of vendor-only IDEs
<b>GitHub App / CI bot</b>	CodeRabbit (general), some dbt teams' homegrown bots	PR-time is the right moment for AI review; async	Latency to merge; review-noise tax
<b>Slack/Teams agent</b>	Wobby/Actian AI Analyst,	Meets business users where they live	Hard to provide rich context

Shape	Examples	Pros	Cons
	Ragstar's /ask, Hex Magic		(lineage graphs, SQL diffs) in-chat

**Distribution insight.** The serious players are stacking shapes: dbt Labs ships an MCP server and skills and a Cloud-native Copilot. Altimate ships a VSCode extension and an embedded MCP and Claude Code skills. Single-shape entrants (just-an-MCP, just-an-extension) are losing footprint.

**Stickiness ranking (informally observed):** Web IDE > VSCode extension > MCP server > Skills > CLI. Web IDE moves contracts; CLI moves stars.

---

## E. The eval gap

The single weakest link in this entire ecosystem is **systematic measurement of LLM-output quality on dbt artifacts**. Nobody is doing this well.

### Who evaluates?

- **Paradime dbt-llm-evals** is the only OSS, dbt-native eval framework. It's the warehouse-native LLM-as-a-judge pattern: judge model is one of llama3-70b / mistral-large / mixtral-8x7b (Snowflake), gemini-pro (BQ), or llama-2-70b-chat (Databricks). 5 built-in dimensions (accuracy, relevance, tone, completeness, consistency) on a 1-10 scale, with baseline detection and drift alerts. **25 stars**. Designed primarily for application output (support replies, classifications) — using it on schema.yml or dbt-test output requires the user to supply baselines.
- **dbt Developer Blog (docs.getdbt.com/blog/ai-eval-in-dbt)** describes a hand-rolled pattern using Snowflake Cortex Complete as a judge against IMDB sentiment data. Walks through dbt-tests-as-quality-gates with a 75% accuracy threshold. **No reusable framework — it's a worked example.**
- **dbt-agent-skills/evals/** ships an A/B testing harness for skill variations (per the README). This evaluates skill prompts, not generated artifacts.
- **DeepEval / Confident AI** are the general-purpose alternatives. Powerful, but require the user to design dbt-specific metrics from scratch.
- **Elementary ai\_data\_validation** evaluates data, not AI output about data — adjacent but different problem.
- **Monte Carlo Agent Observability** (announced March 12, 2026) covers context/performance/behavior/output across 4 pillars — but framed as observability, not as a graded eval that ships with reproducible scores per artifact.

### Why isn't this everywhere?

1. **It's the part nobody wants to fund.** Eval frameworks don't sell themselves; they make the other product look worse before it gets better.

2. **The judge problem is genuinely hard.** Grading “is this dbt test useful” requires domain knowledge the judge LLM mostly lacks. Schema-extraction (does the YAML parse? does it round-trip?) is doable; quality grading needs rubric design that nobody’s standardized.
3. **Vendor incentive misalignment.** dbt Copilot, DinoAI, and Altimate are all generators. Publishing a rigorous eval would mean publishing their own miss rates.
4. **The standard workflow is “human reviews.”** Every blog post says “always validate AI output.” That’s a dodge — it pushes the cost onto the user instead of automating verification.

**The opening:** A deterministic + LLM-graded eval framework that operates on dbt artifacts directly (schema.yml, tests, model SQL), with reproducible scoring and CI integration, is missing. Paradime’s framework is closest but is application-output-shaped, not artifact-output-shaped.

---

## F. Strategic openings — where a 2026 entrant can play

Three macro shifts make 2026 the right time:

1. **Fivetran ↔ dbt merger consolidates the platform layer.** Buyers are now considering one vendor for ingest+transform. Smaller AI tools that orbit either side will be either acquired or marginalized. The opening is outside the merged platform’s gravity well: warehouse-native, OSS-first, or IDE-first plays.
2. **dbt Fusion adoption flips the runtime cost curve.** 30x faster parse means agents can iterate against dbt parse / dbt compile in tight loops without burning CI minutes. Tools that exploit this (proposer + verifier loops on local Fusion) get a step-change improvement that pre-Fusion tools can’t match.
3. **Agentic coding (Claude Code, Cursor agents, Codex) is the new default.** The Feb 2026 ranking is unambiguous that agent-mode is winning. dbt-mcp + skills make the agent dbt-aware. The remaining gap is quality control on what the agent produces.

**Specific openings, ranked by tractability:**

### F1. Eval-as-a-product for dbt AI output (highest leverage)

Nobody owns “did the AI’s schema.yml actually fit my data?” Paradime’s dbt-llm-evals is application-shaped; Monte Carlo’s Agent Observability is enterprise-priced; the dbt Developer Blog’s example is hand-rolled per project.

A tool that takes a dbt project, runs the candidate AI generator (any of dbt Copilot, DinoAI, Altimate, Claude Code), grades the output deterministically (does it parse, do tests pass, does the lineage stay coherent) and via LLM-as-judge (is the doc accurate to the SQL, does the test catch real bugs), and ships reproducible scores into CI — would have no direct competitor today. This is the clauditor pattern applied to dbt.

## **F2. The “useful test” generator**

Universal complaint: AI-generated tests are trivial. A generator that uses query history + production-failure history + schema to propose tests that catch historical incidents would be differentiated. Datafold has the diff piece; nobody combines it with AI test proposal.

## **F3. Mesh-aware PR reviewer**

dbt Mesh + contracts opened a class of breaking-change failures (downstream contract violations, semantic-model drift) that current PR reviewers (CodeRabbit, dbt Cloud CI) catch only structurally. An LLM-mediated reviewer that explains why a contract change is breaking, with downstream impact narration, would slot above generic Copilot review.

## **F4. Stale-code / dead-model detector**

Wide open. Combine `manifest.json` + warehouse query history → “these 47 models haven’t been queried in 90 days; here’s the kill list with rationale.” LLM is the narrator, not the detector.

## **F5. Local Fusion-powered proposer + verifier loop**

Fusion’s 30x-faster parse makes proposer/verifier-style agents cheap. A CLI agent that proposes a model change, verifies via `dbt parse` + `dbt build` against a sample, and iterates until the verification passes — would beat any current single-shot generator on quality. `dbt-mcp` gives the surface; nobody’s shipping the loop.

## **F6. Warehouse-cost-attributed advisor**

Altimate’s cost estimation is the first stab. The full version: ingest warehouse query/credit history, attribute to dbt models, narrate optimization opportunities (“change this view to incremental, save \$X/month”). LLM does the narration and the SQL rewrite; the attribution math is the moat.

## **F7. Skills-marketplace differentiation**

The Anthropic Skills standard is now adopted by OpenAI/Codex too (Dec 2025). 160k+ skills indexed on SkillsMP. `dbt-agent-skills` is the official 13-skill collection. The opening: domain-specific skill packages (industry verticals, warehouse-specific, BI-tool-specific) that ship with their own evals so users can trust them. “Verified skills” is a market that doesn’t exist yet.

---

## **G. Sources & repos**

**Official dbt Labs** - dbt Copilot docs: <https://docs.getdbt.com/docs/cloud/dbt-copilot> - dbt Copilot blog (GA): <https://www.getdbt.com/blog/dbt-copilot-is-ga> - `dbt-mcp` repo (544□, 39 issues, last push 2026-04-24): <https://>

github.com/dbt-labs/dbt-mcp - dbt-agent-skills repo (430, 23 issues): <https://github.com/dbt-labs/dbt-agent-skills> - dbt-fusion repo (690, 386 issues): <https://github.com/dbt-labs/dbt-fusion> - Agentic coding blog: <https://www.getdbt.com/blog/agentic-coding-in-analytics-engineering> - Bring structured context to agentic data dev: <https://www.getdbt.com/blog/bring-structured-context-to-agentic-data-development-with-dbt> - AI eval in dbt blog: <https://docs.getdbt.com/blog/ai-eval-in-dbt> - dbt-codegen: <https://github.com/dbt-labs/dbt-codegen> - Analyst agent docs: <https://docs.getdbt.com/docs/dbt-ai/analyst-agent> - Fivetran merger announcement: <https://www.getdbt.com/blog/dbt-labs-and-fivetran-sign-definitive-agreement-to-merge>

**Paradime** - DinoAI vs dbt Copilot comparison: <https://www.paradime.io/blog/paradime-dinoai-vs-dbt-copilot-a-comparative-analysis> - DinoAI features: <https://www.paradime.io/blog/dinoai-features-build-faster-spend-less-dbt-development> - DinoAI Context: <https://www.paradime.io/blog/introducing-dinoai-context-supercharging-analytics-engineering-workflows> - dbt-llm-evals repo (25): <https://github.com/paradime-io/dbt-llm-evals> - dbt-llm-evals blog: <https://www.paradime.io/blog/get-started-with-dbt%E2%84%A2-llm-evals-warehouse-native-llm-evaluation-in-15-minutes> - LLM eval criteria: <https://www.paradime.io/blog/llm-evaluation-criteria-how-to-measure-ai-quality>

**Altimate** - vscode-dbt-power-user (572, 132 issues): <https://github.com/AltimateAI/vscode-dbt-power-user> - Embedded MCP for Cursor: <https://blog.altimate.ai/supercharging-cursor-ide-how-the-dbt-power-user-extensions-embedded-mcp-server-unlocks-ai-driven-dbt-development> - Claude Code Skills: <https://blog.altimate.ai/teaching-claude-code-the-art-of-data-engineering-introducing-altimate-skills> - Stability issue #1798: <https://github.com/AltimateAI/vscode-dbt-power-user/issues/1798> - Endless parse loop: <https://discourse.getdbt.com/t/dbt-power-user-extension-endless-parsing-loop-on-dbt-1-11-2/20563>

**Datafold** - Migration agent blog: <https://www.datafold.com/blog/data-migrations-reimagined-introducing-the-ai-powered-datafold-migration-agent/> - Pricing: <https://www.datafold.com/pricing> - G2 reviews: <https://www.g2.com/products/datafold/reviews>

**Snowflake Cortex** - Cortex Code GA Mar 2026: <https://docs.snowflake.com/en/release-notes/2026/other/2026-03-09-cortex-code-snowsight-ga> - Cortex + dbt blog: <https://www.snowflake.com/en/blog/cortex-code-governed-agent-data-stack/> - Cortex Analyst docs: <https://docs.snowflake.com/en/user-guide/snowflake-cortex/cortex-analyst> - Conversational analytics with dbt + Cortex: <https://docs.getdbt.com/blog/semantic-layer-cortex> - LLM-powered AE with Cortex (dbt Developer Blog): <https://docs.getdbt.com/blog/dbt-models-with-snowflake-cortex>

**Other tools / repos** - Hex Magic: <https://hex.tech/product/magic-ai/> - Hex x dbt: <https://hex.tech/product/integrations/dbt/> - Cube MCP server docs: <https://cube.dev/docs/product/apis-integrations/mcp-server> - Monte Carlo MC Agent Toolkit: <https://www.montecarlodata.com/mc-agent-toolkit/> - MC Agent Observability launch: <https://www.apmdigest.com/monte-carlo>

launches-agent-observability - Wobby × dbt: <https://www.wobby.ai/integrations/dbt> - Wobby acquisition by HCLSoftware: <https://www.actian.com/company/press-releases/hclsoftware-to-acquire-ai-data-analyst-agents-startup-wobby/> - Ragstar / dbt-llm-agent (1700): <https://github.com/pragunbhutani/dbt-llm-agent> - dbt-coves: <https://github.com/datacoves/dbt-coves> - Elementary AI data validation docs: [https://docs.elementary-data.com/data-tests/ai-data-tests/ai\\_data\\_validations](https://docs.elementary-data.com/data-tests/ai-data-tests/ai_data_validations) - Atlan dbt catalog: <https://atlan.com/dbt-data-catalog/> - SelectStar dbt docs sync: <https://www.selectstar.com/resources/dbt-docs>

**Independent commentary** - Feb 2026 AI coding tools ranking for data teams: <https://index.app/blog/ai-coding-tools-data-teams-claude-cursor-copilot-ranked> - Why we aren't achieving the Copilot experience in data systems: <https://clkao.substack.com/p/why-we-arent-achieving-the-copilot> - Cursor for analytics (paywalled): <https://learnanalyticsengineering.substack.com/p/cursor-for-analytics-where-it-fails> - Mikkel Dengsoe — AI for testing: <https://mikkeldengsoe.substack.com/p/using-ai-to-build-a-robust-testing> - Loblaw Digital — LLM dbt docs: <https://medium.com/loblaw-digital/leveraging-llms-to-generate-ai-driven-dbt-documentation-c4735faa6ca5> - "AI-Generated dbt Models Are Actually Good Now (I Tested 50 of Them)": <https://medium.com/@reliabledataengineering/ai-generated-dbt-models-are-actually-good-now-i-tested-50-of-them-b87bd82bc7c2> - Awesome-dbt curated list: <https://github.com/Hiflylabs/awesome-dbt>

---

## H. One-paragraph TL;DR for a strategy memo

dbt Labs has effectively boxed in the AI-on-dbt-Cloud surface area with Copilot, the Insights Analyst agent, dbt-mcp, and dbt-agent-skills. Paradime and Altimate own the "richer-than-Cloud IDE" and "VSCode/Cursor extension" niches respectively; both ship MCP servers and skills now. Datafold owns large-scale migration. Snowflake Cortex Code (GA Mar 2026) and Hex Magic own warehouse-native and notebook-native generation. The Fivetran merger turned the platform story into ingest-to-activation; the runway underneath is **quality assurance on AI-generated dbt artifacts** — schema.yml, tests, docs, semantic-model definitions — where Paradime's dbt-llm-evals (25 stars) is the only OSS attempt and is application-output-shaped, not artifact-output-shaped. A clauiditor-style framework that treats dbt AI generation as something to evaluate, not just invoke, has no direct incumbent and aligns with the "verifier-cheap on Fusion" runtime shift coming through 2026.

# Appendix E — Tool Design Sketches

## dbt LLM Tool Design Sketches

Three concrete designs evaluated for viability. Aim: pick one to build.

---

## OPPORTUNITY 1: PR Review Companion (dbt-pr-companion)

A GitHub Action that reads a dbt PR, queries the warehouse for impact/cost, and posts ONE structured comment summarizing what changed, what's at risk, and what's missing.

### 1. User journey

**Sarah, senior analytics engineer at FinLoop (a 280-person fintech).** Stack: dbt Cloud, Snowflake, GitHub PRs, Hightouch (reverse ETL to Salesforce + Braze), Looker. ~620 dbt models, 14 sources, 4 mart domains. Team of 6 analytics engineers reporting into Sarah; 3 of them are juniors who joined in the last 8 months.

#### Today, on a normal Tuesday:

A junior opens PR #1247, "fix: refactor fct\_subscription\_events to handle proration." The PR touches one model. Sarah's review process:

1. Read the SQL diff. The case statement got longer; she squints to figure out which branch is new.
2. Open dbt-docs in another tab to find downstream models. There are 23 of them. She doesn't read them all — she scans for `mart_*` and `dim_customer_facts`.
3. She **forgets** that `fct_subscription_events` is the source for the Hightouch sync to Braze. That's not in dbt-docs; it's in a separate Hightouch UI nobody opens during code review.
4. She scans for new tests. There are none. She comments "can we add a test for the proration logic?" — but she doesn't say which test shape, so the junior adds `not_null` on a column that's already tested.
5. The PR merges. Three days later a Braze segment fires for 12,000 users who shouldn't have received an upsell email because the new proration branch nulled `subscription_tier` for prorated users.

This happens once a quarter. Each time, Sarah's team writes a postmortem and adds another bullet to the PR review checklist. The checklist is now 17 items long. Nobody follows it.

#### With dbt-pr-companion installed:

Sarah opens PR #1247. There's already a comment from dbt-pr-companion posted 90 seconds after the PR opened. It says:

**Logic change:** The case statement for `subscription_status` now emits 'prorated' for the window between `proration_start` and the next `billing_cycle_start`. Previously these rows fell through to `else null`.

**Downstream blast radius (4 high-risk consumers):** - `dim_customer_facts` — joins on `subscription_status`; the new

'prorated' value is not in any accepted\_values test -  
mart\_revenue\_daily — sum(amount) where  
subscription\_status = 'active' will now exclude prorated  
rows - **Hightouch sync braze\_subscription\_segments** (detected  
via meta.hightouch\_sync on the model) — segment definitions  
reference subscription\_status IN ('active', 'trial') - Looker  
LookML subscription\_status dimension — flagged via the  
Looker Git integration (manual confirmation needed)

### Suggested test (1):

```
- name: subscription_status
  tests:
    - accepted_values:
      values: ['active', 'trial', 'cancelled',
              'prorated', 'expired']
```

Rationale: the new 'prorated' enum value silently breaks any downstream consumer that filters on subscription\_status. An accepted\_values test catches future enum drift.

**Cost class change:** NEUTRAL. Estimated bytes-scanned via EXPLAIN: 2.4 GB → 2.5 GB (+4%). Estimated \$ delta on the daily run: +\$0.08.

**Doc/yaml drift:** □ The subscription\_status column description in schema.yml still says “active|trial|cancelled|expired”. Update needed.

Sarah skims the comment in 30 seconds. She approves with one addition (“can you also update the Hightouch sync’s IN clause?”). The junior fixes it. PR merges. Braze sends the right emails.

**The change:** Sarah’s review goes from 12 minutes of tab-switching context-rebuilding to 2 minutes of validating an LLM-prepared summary. The 17-item checklist is replaced by a structured comment that always runs.

## 2. Concrete UX

### GitHub Action installation (one-time):

```
## .github/workflows/dbt-pr-companion.yml
name: dbt-pr-companion
on:
  pull_request:
    paths: ['models/**', 'tests/**', 'macros/**', '**.yaml']
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
```



```

- uses: actions/checkout@v4
  with:
    fetch-depth: 0 # need base ref for diff
- uses: dbt-pr-companion/action@v1
  with:
    warehouse: snowflake
    warehouse-creds: $
    {{ secrets.SNOWFLAKE_DBT_PR_COMPANION }}
    anthropic-api-key: ${{ secrets.ANTHROPIC_API_KEY }}
    # Optional integrations
    hightouch-api-key: ${{ secrets.HIGHTOUCH_API_KEY }}
    looker-git-repo: my-org/looker-config

```

**Configuration** (.dbt-pr-companion.yml at repo root):

```

version: 1
profile: prod # which dbt profile to run --target against
analysis:
  cost_estimation: true
  column_lineage: true
  enum_drift_detection: true
  reverse_etl_integrations:
    - hightouch
  bi_integrations:
    - looker
risk_thresholds:
  # Models tagged 'critical' always get a top-level comment block
  critical_tags: ['mart', 'finance', 'pii']
  # Cost change above this triggers warning
  cost_delta_warning_pct: 25
review_persona: senior # affects verbosity: senior|junior|
                    verbose

```

**CLI mode** (for local dry-runs before opening a PR):

```

$ dbt-pr-companion review --base main
[1/5] Compiling dbt... ✓
      8.3s
[2/5] Diffing manifest.json (base vs HEAD)... ✓
      0.4s
      → 1 model changed, 0 added, 0 removed
[3/5] Resolving downstream impact (column-level)... ✓
      1.1s
      → 23 immediate consumers, 4 high-risk
[4/5] Querying warehouse for cost estimate (EXPLAIN)... ✓
      2.7s
[5/5] Generating LLM review (claude-sonnet-4-7)... ✓
      11.4s

```

=== PR REVIEW DRAFT ===

[same content as PR comment above]

Tokens: 18.4k in / 2.1k out • Cost: \$0.062 • Total: 24s

**Where it lives:** GitHub Action (primary), with CLI for local preview. NOT a VSCode extension v1 — the value is in the comment showing up automatically without anyone remembering to invoke it.

### 3. Technical architecture

#### Inputs read at runtime:

- `target/manifest.json` — built from `dbt parse` on both base and HEAD refs. Used for: model graph, column-level lineage (where dbt has it), tags, meta fields, descriptions.
- `target/catalog.json` (optional, if `dbt docs generate` cached) — warehouse-resolved column types and stats.
- `target/run_results.json` from the most recent prod run (fetched from S3 / dbt Cloud API) — used for cost-class baseline.
- Git diff of `models/**/*.sql` and `**/*.yml` — the actual change.
- For each touched model: `dbt compile` to get the rendered SQL.

#### Warehouse queries (all run as a low-privilege role `DBT_PR_COMPANION_RO`):

- `EXPLAIN <compiled-sql>` for cost estimation. Snowflake: `EXPLAIN USING JSON`. BigQuery: `--dry_run`. Redshift: `EXPLAIN`. Databricks: `EXPLAIN COST`.
- `INFORMATION_SCHEMA.COLUMNS` query to detect rename/add/drop against the model's currently materialized shape.
- Optionally: `QUERY_HISTORY` lookup (last 30 days) to estimate how often this model is read and by whom — feeds the “blast radius” scoring.

NO sample data queries by default. The warehouse never sees row contents. (See Hard Problems §1 for why this matters.)

**LLM API surface:** Anthropic SDK direct via `claude-sonnet-4-7`. Single-shot per PR (NOT an agent loop). One call assembles the comment from a prompt that includes:

- The compiled SQL diff (trimmed to changed regions  $\pm 20$  lines context)
- Manifest excerpt for the changed model + its 4 highest-risk consumers
- Column-level diff (computed deterministically before the LLM call)
- Cost estimate (computed deterministically)
- A structured-output schema (JSON) that the LLM fills in

The LLM is NOT used as an agent. It's used as a narrative generator over a deterministically-prepared input. This is the key architectural choice: the impact analysis, cost estimate, and column diff are all computed by deterministic code that doesn't need an LLM. The LLM's job is to (a) explain the SQL change in plain English, (b) suggest tests with rationale, (c) write the prose.

**Output schema** the LLM fills:

```
{
  "logic_change_summary": "string, 2-4 sentences, plain English",
  "suggested_tests": [
    {"yaml_block": "...", "rationale": "...", "criticality":
      "high|medium"}
  ],
  "doc_drift_findings": [
    {"file": "...", "current_text": "...", "should_be": "..."}
  ]
}
```

Deterministic blocks (downstream impact, cost class, column diff) are templated into the comment without going through the LLM at all.

### Caching:

- Manifest diff cached by (base\_sha, head\_sha) tuple → reuse on re-runs of the same PR.
- LLM response cached by hash of (compiled SQL diff + manifest excerpt + schema version). A force-push that doesn't change semantic SQL won't burn another API call.
- Warehouse EXPLAIN cached by SQL hash for 24h.

**Cost per invocation:** ~\$0.05–\$0.15 per PR comment at claude-sonnet-4-7 pricing (avg ~20k input, ~2k output). Heavy PRs touching 5+ models: ~\$0.40. A team doing 200 dbt PRs/month: ~\$15–\$30/month in API costs.

**Idempotency:** comments are upserted (one comment per PR, edited on each commit) keyed by a hidden HTML marker. Re-runs replace the prior comment, never spam.

## 4. The hard problems

### Technical risks:

1. **Column-level lineage is shallow in dbt.** dbt only tracks model-to-model lineage natively; column-level requires sqlglot-based parsing of compiled SQL or paid integrations (Datafold has the best engine here). MVP can use sqlglot but it'll mis-handle macros and complex CTEs in 5–10% of cases.
2. **Reverse ETL / BI integrations are bespoke per vendor.** Hightouch, Census, Polytomic, Looker, Tableau, Mode each need custom adapters. v1 picks ONE (Hightouch — best API, biggest overlap with dbt users).
3. **Cost estimation accuracy varies by warehouse.** Snowflake's EXPLAIN USING JSON is decent; Redshift's is misleading; BigQuery's --dry\_run only gives bytes-scanned (no slot-time); Databricks needs a Photon-aware estimator. v1: Snowflake + BigQuery only.
4. **Compiled SQL needs warehouse credentials at PR time.** Means either dbt Cloud API integration OR running dbt parse (compile-only,

no warehouse) in the action. Latter is preferred but loses materialization/relation resolution.

### Quality risks:

1. **Suggested tests that are technically correct but socially wrong** — e.g. suggesting a `not_null` test on a column the team has explicitly decided can be null for business reasons. Mitigation: read existing tests + model `meta.testing_philosophy` field.
2. **Hallucinated downstream consumers.** The deterministic-first architecture mitigates this — the impact list comes from manifest parsing, not the LLM. The LLM only writes prose around a pre-computed list.
3. **Comment fatigue.** If the comment is wrong 1 in 5 times, reviewers stop reading. Need a “Was this helpful?” ☐/ ☐ reaction capture and an explicit confidence indicator on the suggested tests.

### Adoption risks:

1. **GitHub permissions.** Posting comments needs `pull-requests:write`. Many enterprise GitHub orgs disable third-party Actions from posting comments. Workaround: action runs in customer’s own Action environment (no SaaS), uses `{{ github.token }}` — no third-party trust needed.
2. **Snowflake credentials in CI.** Some teams flat-out won’t put prod warehouse creds in GitHub Action secrets. Mitigation: support a read-only “review” warehouse account; document the minimum-privilege role.
3. **“We already have Datafold.”** Datafold’s value-diff is excellent but expensive (\$60k+/yr enterprise). This tool’s pitch is “Datafold-lite for the 90% of teams that can’t justify Datafold pricing.”

### Maintenance risks:

1. **manifest.json schema changes between dbt versions.** `dbt-core` 1.5 → 1.6 → 1.7 → 1.8 each broke something. Need version-pinned adapters (`dbt-artifacts-parser-style`).
2. **Anthropic model deprecations.** Centralized SDK call (a la `clauditor’s _anthropic.py`) makes upgrades a one-line change.

## 5. Differentiation from incumbents

Tool	Strength	Where it loses to dbt-pr-companion
Datafold	Best column-level lineage + value diffs in the industry	\$60k+/yr; overkill for 90% of teams; doesn’t write prose summaries; no test suggestions
Recce	Open source, great UI for diffs	No GitHub-native PR comments; no LLM-written summaries; requires a separate UI

<b>Tool</b>	<b>Strength</b>	<b>Where it loses to dbt-pr-companion</b>
<b>dbt Copilot (Cloud)</b>	Native integration, autocomplete	Authoring-time, not review-time; no PR comment; dbt Cloud only
<b>Paradime AI</b>	Solid IDE assistant	IDE-bound, not CI; doesn't ground in your manifest
<b>GitHub Copilot Workspace</b>	Generic PR review	No dbt-awareness; no warehouse cost estimate; no Hightouch detection

**The honest case for incumbents winning:** Datafold wins on enterprise teams that need data-diff-on-every-PR and have the budget. Recce wins on teams that prefer a self-hosted UI for exploration. dbt-pr-companion wins on the long tail of mid-market teams (50-500 employees, 200-2000 dbt models) who need something better than a checklist and don't have \$5k/month to spend.

## 6. Ship-in-a-weekend MVP

### In scope (3 days):

- GitHub Action that runs on PR open/sync
- dbt parse to produce manifest, diff against base manifest
- Compiled SQL diff for changed models (dbt compile)
- Deterministic downstream-consumer list (model-level only, not column-level)
- Single claude-sonnet-4-7 call producing logic-change summary + test suggestions
- Markdown comment posted via GitHub API
- Snowflake-only cost estimate via EXPLAIN
- Caching by PR SHA

### Out of scope:

- Hightouch / Looker integrations
- Column-level lineage (model-level is the demo)
- BigQuery / Redshift / Databricks
- Doc-drift detection (v1.1)
- Confidence scoring / █ capture (v1.2)

The MVP gives you a screenshot for the launch tweet. That's enough.

## 7. Path-to-100-users

1. **Build it on a real OSS dbt project first** (e.g. dbt-labs/jaffle\_shop, or contribute it to a fast-moving OSS dbt project that has PRs to demo against). Get screenshots of real comments.
2. **Show, don't tell, on dbt Slack** — #tools-and-integrations channel. Post a 60-second screen recording of a real PR comment.
3. **Coalesce 2026 lightning talk submission.** "How we replaced our 17-item PR checklist with one LLM comment."

4. **Listing on Awesome dbt + dbt package hub** (the action wraps a dbt-pr-companion package).
5. **Targeted outreach to 20 mid-market data teams** via LinkedIn — teams between 100 and 1000 employees that have public dbt repos on GitHub. Offer to install for free in exchange for a quote.
6. **Pricing:** free OSS for solo / hobby / public repos. Paid (\$199/mo per 10 contributors) for private repos with the SaaS features (BI integration adapters, cost analytics dashboard, 📊 telemetry).

OSS-first because the value is seeing the comment on a real PR. You can't gate the demo behind a sales call.

## 8. 12-month evolution

- **v1 (month 0-2):** GitHub Action, Snowflake, model-level impact, single LLM call, free OSS.
- **v1.5 (month 3-4):** Column-level lineage via sqlglot. BigQuery support. Doc-drift detection. First paid tier.
- **v2 (month 6-8):** Hightouch + Looker integrations. Historical cost-trend dashboard. Slack integration ("PR #1247 looks risky, want to ping the data team?"). \$499/mo per team plan.
- **v2.5 (month 9-12):** "PR companion" expands to "release companion" — same engine reviews dbt model deploys to prod, posts to Slack with a daily/weekly digest of risk patterns.

**Moat that compounds:** the warehouse query history + 📊 telemetry. After 6 months you know which test suggestions actually got accepted, which downstream warnings actually correlated with production incidents. That feedback loop is unobtainable from a cold start.

## 9. Why-not-build-this

- **Datafold has been doing column-level lineage for 5 years and has the engine.** They could add an LLM summary in a sprint and bundle it into existing contracts. The bear case: Datafold ships this exact feature in Q2 2026 and crushes the OSS option.
- **dbt Labs has the manifest, the Cloud product, and a roadmap.** dbt Cloud's "Explorer" already does column lineage. Adding a PR comment is two engineers and a quarter.
- **GitHub Copilot Workspace is going to swallow generic PR review.** A GH-native PR review feature with claude-sonnet underneath is plausible by end of 2026.
- **The Hightouch / Looker integration story is gnarly.** Each vendor has its own auth, schema, rate limits. Building 6 integrations is a year of work.

What kills it: dbt Labs ships PR review in dbt Cloud and bundles it free. Realistic mitigation: stay open-source-first, dbt-Cloud-agnostic (works with dbt-core too), and own the "non-dbt-Cloud half of the market" segment.

---

## OPPORTUNITY 2: Schema/Tests/Docs Generator (dbt-yaml-forge)

LLM drafts `schema.yml` + tests + docs for a model, samples warehouse data, runs candidate tests against real data, drops always-pass / uninformative tests, emits only signal-bearing artifacts. Each artifact graded by an LLM rubric for usefulness before being kept.

### 1. User journey

**Marcus, staff data engineer at HelmCorp (a 1,400-person logistics company).** Stack: dbt-core (self-hosted), BigQuery, Airflow, GitHub. ~1,800 models inherited from a 4-year-old project. Test coverage on `dim_*` and `fct_*` is decent. Test coverage on `stg_*` and `int_*` (which together are 60% of models) is abysmal — most have zero tests, no descriptions, no column docs.

#### Today, on a normal sprint:

His team has been told by leadership “improve data quality.” Marcus spends 2 hours writing a `schema.yml` for `int_orders_unioned`. The model has 47 columns. He:

1. Eyeballs the SQL to figure out what each column means
2. Adds `not_null` to columns that seem like they shouldn't be null
3. Adds `unique` to `order_id` because it's an `int_model` and “should be” unique
4. Half the tests fail on the next CI run because his eyeballing was wrong (3 columns are nullable on Tuesdays due to a known pipeline race; `order_id` is unique within (`order_id`, `order_version`) not standalone)
5. He spends another hour debugging false positives
6. He never gets around to the other 28 `int_*` models on his list
7. After 3 sprints, leadership stops asking about “data quality” and the project dies

#### With dbt-yaml-forge:

```
$ dbt-yaml-forge propose --model int_orders_unioned
[1/6] Compiling model... ✓
[2/6] Sampling 5,000 rows from materialized table... ✓
      1.4s
[3/6] Profiling columns (null %, distinct %, ranges)... ✓
      0.8s
[4/6] LLM drafting schema.yml + 18 candidate tests... ✓
      14.2s
[5/6] Running 18 candidate tests against real data...
      ✓ unique: (order_id, order_version) —
        discriminating
      ✗ not_null: order_id — keep (1
        fail)
```

```

x not_null: shipped_at          – DROP (50%
null,                          informative
null)
✓ accepted_values: status (...) –
discriminating
✓ relationships: customer_id → ... –
discriminating
x
not_null: warehouse_code      – DROP (always
passes,
uninformative)
... (18 total: 11 kept, 7 dropped)
[6/6] LLM rubric grading kept artifacts (signal score)... ✓
8.1s
Avg signal score: 7.2/10 (threshold: 6.0)
2 column docs flagged low-signal – rewriting...

Wrote: models/intermediate/_int_orders_unioned__schema.yml
(47 columns documented, 11 tests, est. coverage 78%)

Tokens: 31.4k in / 4.8k out • Cost: $0.18 • Total: 28s

```

The generated schema.yml is checked into the PR. Marcus reviews it — it took him 4 minutes instead of 2 hours, and the tests are better than what he would have written.

**The change:** Marcus’s team can document and test 28 models in a sprint instead of 1. The “data quality initiative” actually finishes.

## 2. Concrete UX

**CLI is the primary surface** (this is a developer tool, not a CI tool):

```

## Single model
dbt-yaml-forge propose --model int_orders_unioned

## Whole directory
dbt-yaml-forge propose --select intermediate

## All untested models
dbt-yaml-forge propose --untested

## Dry-run (no warehouse queries, no LLM calls)
dbt-yaml-forge propose --model X --dry-run

## Re-grade existing schema.yml (no regeneration)
dbt-yaml-forge audit --model X

```



### Configuration (.dbt-yaml-forge.yml):

```
version: 1
sample:
  rows: 5000
  strategy: random # random | recent | stratified
  pii_columns_pattern: "(email|ssn|phone|address|name) $"
  pii_handling: hash # hash | redact | exclude
test_acceptance:
  min_failure_rate: 0.001 # tests must fail on >0.1% of rows to
    be "discriminating"
  max_failure_rate: 0.5 # tests that fail >50% are signal-less
rubric:
  min_score: 6.0 # below this, regenerate
  max_attempts: 3
generation:
  test_types_enabled:
    - unique
    - not_null
    - accepted_values
    - relationships
    - dbt_utils.expression_is_true
  doc_style: terse # terse | descriptive
```

### Generated YAML (excerpt):

```
version: 2
models:
  - name: int_orders_unioned
    description: |
      Union of orders from the legacy `orders_v1` source and the
      new
      `orders_v2` source, with version reconciliation. One row
      per
      `(order_id, order_version)` – orders can have multiple
      versions when amended post-checkout.
    columns:
      - name: order_id
        description: Stable order identifier across both source
          systems.
          # NOTE: not unique alone – see (order_id, order_version)
          test below
      - name: order_version
        description: |
          Monotonically increasing version per `order_id`. Starts
          at 1.
          Incremented when a customer amends shipping address or
          adds/removes line items pre-shipment.
      - name: status
```

```

description: Order lifecycle state.
tests:
  - accepted_values:
      values: ['pending', 'confirmed', 'shipped',
              'delivered',
              'cancelled', 'returned']
      # auto-discovered from sample data; please verify
tests:
  - dbt_utils.unique_combination_of_columns:
      combination_of_columns: ['order_id', 'order_version']
  - relationships:
      to: ref('stg_customers')
      field: customer_id
      column_name: customer_id

```

Each generated test carries a comment: # auto-discovered from sample data; please verify — explicit about provenance.

**Where it lives:** CLI tool, distributed as a dbt package + Python CLI. NOT a GitHub Action (yet). The reason: schema/test generation is a deliberate authoring action, not a passive review. The author chooses when to invoke it.

### 3. Technical architecture

#### Inputs:

- target/manifest.json — model graph, existing tests, existing descriptions (so we don't overwrite)
- target/catalog.json — column types, relations
- Compiled SQL via dbt compile
- A sample of warehouse rows (deterministic seed)
- Source SQL of upstream ref()'d models (for context)

#### Warehouse queries:

- SELECT \* FROM {{ this }} TABLESAMPLE (...) LIMIT N — sample for profiling
- SELECT COUNT(\*), COUNT(col), COUNT(DISTINCT col), MIN(col), MAX(col) FROM {{ this }} GROUP BY col style profiling per column
- For each candidate test: a dbt test --select <model> --store-failures invocation in a sandboxed schema

#### LLM API:

- **Anthropic SDK direct**, single-shot per phase (NOT an agent).
- Phase 1 — Drafter (Sonnet): given compiled SQL + column profile, produce candidate schema.yml + candidate tests
- Phase 2 — Test execution: deterministic, no LLM
- Phase 3 — Grader (Haiku for cost — runs once per artifact): rubric-grade each kept test/description against criteria like “is this description

specific enough to disambiguate from similarly-named columns elsewhere?”, “does this test catch a realistic failure mode?”

- Phase 4 — Regenerator (Sonnet, only if avg score < threshold): rewrite low-scoring artifacts with critique included

**This is where clauditor synergy is enormous.** The grader phase is exactly what clauditor does today — a per-artifact LLM-graded rubric with thresholds and regeneration. dbt-yaml-forge’s grader literally calls a clauditor-style harness:

```
## inside dbt-yaml-forge's grader phase
from clauditor.quality_grader import grade_quality
from clauditor.schemas import EvalSpec

eval_spec = EvalSpec.from_dict({
    "skill_name": "yaml-forge-test-quality",
    "grading_criteria": [
        {"id": "specificity", "criterion": "..."},
        {"id": "non-trivial", "criterion": "..."},
        {"id": "actionable-on-failure", "criterion": "..."},
    ],
    # ...
}, spec_dir=...)

report = await grade_quality(eval_spec, candidate_test_yaml)
if report.pass_rate < 0.6:
    # regenerate
```

### Grounding/context:

- Sample data is profiled (statistics) not passed verbatim to the LLM in v1 — keeps PII out of the prompt.
- Optional --allow-sample-rows N for non-PII columns (whitelist in config).
- The compiled SQL is the primary context (~ 5-50 lines per model). Manifest excerpts are pruned to direct upstream/downstream.

### Caching:

- Per model + manifest version ({model\_name}@{manifest\_hash}).
- Cache invalidated when SQL changes or upstream schema changes.

**Idempotency:** re-running on a model with existing schema.yml augments, never overwrites. Existing descriptions are preserved unless --overwrite is passed.

**Cost per invocation:** \$0.10-\$0.30 per model. A team running forge on 50 models: ~\$10. A one-time backfill across 1,000 models: ~\$200.

## 4. The hard problems

### Technical risks:

1. **Sampling without leaking PII.** Even profiling can leak (a MIN(email) reveals the alphabetically-first email). Need a PII detector + redactor pipeline before any data hits the LLM. Mitigation: pii\_columns\_pattern config + a hardcoded blocklist of common PII patterns + a --no-warehouse-data mode that relies only on schema + SQL.
2. **Test execution in a sandboxed schema.** Can't pollute prod. Need dbt's --target + a writable analyst schema, or dry-run test execution (some tests support it).
3. **Always-pass test detection requires actually running the tests.** This means warehouse compute on every forge invocation. Costs add up for big teams.
4. **dbt\_utils and other macros require dbt project context.** Forge needs to know which packages are installed before suggesting dbt\_utils.expression\_is\_true.

### Quality risks:

1. **Tests that pass on sample but fail on full table.** A 5k-row sample of a 5B-row table might miss the one rogue NULL. Mitigation: profile across the full table (cheap aggregation), only sample rows for context.
2. **Generated descriptions that are "plausible but wrong".** The LLM sees customer\_id BIGINT NOT NULL and writes "Unique identifier for the customer." But maybe in this model customer\_id is the billing customer, not the shipping customer. Mitigation: rubric grades on "specific enough to disambiguate"; force regeneration on generic descriptions.
3. **Suggested tests become a wall of noise.** If forge suggests 18 tests per model and devs accept all, the test suite slows down and tests stop being read. The "drop always-pass / uninformative" pruning is the load-bearing quality move.
4. **Rubric calibration drifts.** Same problem clauditor solves. Same solution: a small held-out set of "known good" / "known bad" YAML examples that the rubric must score correctly, verified on every release.

### Adoption risks:

1. **"I don't trust LLM-generated tests."** The provenance comment on each test (# auto-discovered from sample data; please verify) is a partial answer. Better answer: ship a "show me your work" mode that prints why each test was kept.
2. **Warehouse access at authoring time.** Many devs work locally without prod warehouse access. Need a --warehouse-snapshot mode that can run against a pre-captured profile JSON.
3. **YAML conflicts with hand-edits.** If a user hand-edits the YAML and re-runs forge, what happens? Need explicit merge semantics + a --preserve-existing default.

## Maintenance risks:

1. **dbt test API changes between versions.** Same as opportunity 1.
2. **PII detection regex maintenance** — a never-ending arms race.

## 5. Differentiation from incumbents

Tool	Strength	Where it loses
<b>dbt-codegen</b> (OSS)	Generates boilerplate <code>schema.yml</code> skeleton from a model	No tests, no descriptions, no quality grading; produces literally <code>description: ""</code>
<b>dbt Copilot (Cloud)</b>	Inline suggestions in IDE	dbt Cloud only; doesn't run candidate tests against data; no rubric scoring
<b>Paradime AI Schema Generator</b>	Decent inline generator	No "drop always-pass" pruning; no rubric grading; closed source
<b>dbt-checkpoint pre-commit hooks</b>	Enforces missing tests/docs	Doesn't generate them — just nags
<b>Manually-written schema.yml</b>	Always correct (when correct)	Slow, inconsistent, last-priority

**Where incumbents win:** dbt-codegen wins for teams that just want the skeleton and prefer to write tests by hand. dbt Cloud Copilot wins for dbt Cloud customers who don't want another tool. forge wins on teams with large untested model bases that need a high-quality automated pass.

## 6. Ship-in-a-weekend MVP

### In scope (3 days):

- CLI `dbt-yaml-forge propose --model <name>`
- Snowflake + BigQuery (one each, simplest profiling queries)
- Phase 1 (drafter, Sonnet) + Phase 2 (test execution, real warehouse)
- Drop-always-pass + drop-always-fail pruning
- Output: `__<model>__schema.yml` file
- PII detection: regex against column names only (email, phone, ssn, name, address)

### Out of scope:

- Phase 3 rubric grading (uses simple heuristic in v1: tests must have failure rate in `[0.001, 0.5]`)
- Phase 4 regeneration loop
- Description quality grading (descriptions are emitted but not scored)
- Backfill mode (`--select intermediate` for many models)
- Custom test types beyond the dbt builtins + `dbt_utils`

The MVP demo: pick a real OSS dbt project, run forge on a model with no tests, show the generated YAML and how the dropped tests were the right ones to drop.

## 7. Path-to-100-users

1. **Make it a dbt package** so installation is one line in `packages.yml`. dbt package hub listing.
2. **Demo on jaffle\_shop and dbt\_artifacts** — well-known OSS dbt projects. PRs against them showing forge's output.
3. **dbt Slack #i-made-this** post with a 90-second screen recording.
4. **Coalesce talk submission**: "How we documented and tested 1,200 models in a week."
5. **Targeted blog post**: "An LLM wrote 600 dbt tests — only 142 were worth keeping." (The pruning story IS the story.)
6. **Pricing**: free OSS for the CLI. Paid SaaS (\$299/mo) for teams who want a hosted dashboard showing test coverage trends, PII compliance audit log, and a Slack bot for nightly forge runs against new models.

## 8. 12-month evolution

- **v1 (month 0-2)**: CLI, single-model, Snowflake + BigQuery, basic pruning.
- **v1.5 (month 3-4)**: Rubric grading via clauditor integration. Multi-model --select mode. Backfill report ("forge would improve coverage from 23% → 71% if you accept these 412 generated tests").
- **v2 (month 5-8)**: Source freshness inference, custom test generation (e.g. business-rule tests from natural-language descriptions in `meta.business_rules`). Hosted SaaS dashboard.
- **v2.5 (month 9-12)**: Cross-model invariant detection ("these 3 mart models all reference `customer_id`; forge inferred a shared `dim_customers` ref"). Continuous coverage monitoring with weekly digest.

**Moat that compounds**: a corpus of human-graded test quality labels. After 6 months you have thousands of "this generated test was kept / dropped / edited" decisions. That data trains a much better grader than anyone else can build cold.

## 9. Why-not-build-this

- **dbt Labs has the model context AND the YAML schema**. They can build this and charge dbt Cloud customers for it. Bear case: dbt Cloud ships "Auto-document & test" feature in Q3 2026.
- **Quality is hard to demonstrate without real-world adoption**. The whole pitch is "the tests forge generates are good" — but proving that requires a public benchmark suite or a year of case studies.
- **Warehouse-write access is a real friction**. Test execution needs writable schema. Many enterprise teams won't grant it to a third-party tool.

- **The “always-pass test pruning” idea is novel-seeming but contested.** Some teams want documentation tests (`not null` on every PK column even if it always passes — proves the assumption is checked). The pruning needs to be configurable, not opinionated.
- 

## **OPPORTUNITY 3: Stored-Proc → dbt Migration Agent (procmigrate)**

Open-source LLM agent that parses procedural SQL (T-SQL stored procs, PL/pgSQL functions, BigQuery procedures), clusters logic into staging/intermediate/mart layers, emits dbt models with `ref()`s, and generates `audit_helper`-style row-count/value-diff tests so the migration is verifiable against the source. Iterates until parity.

### **1. User journey**

**Priya, lead data architect at MidWestern Insurance Co. (a 6,200-person insurer founded 1924).** Stack: SQL Server (still), SSIS pipelines, 1,400 stored procedures spanning 280k lines of T-SQL. They bought Snowflake 18 months ago. The directive: “be on dbt + Snowflake by end of 2026.” They have one analytics engineer who knows dbt (Priya) and three SQL Server DBAs who know T-SQL but have never written a dbt model.

### **Today, on a normal Wednesday:**

Priya is migrating `usp_calculate_monthly_premium_adjustments` — a 1,400-line stored procedure that: - declares 23 temp tables - has 4 nested cursors - calls 6 other stored procs - runs on the 1st of each month and writes to 8 tables

She:

1. Spends 2 days reading the proc to understand it
2. Manually traces the data flow on a whiteboard
3. Identifies what should be staging vs intermediate vs mart
4. Writes 14 dbt models translating the logic
5. Spends a week debugging row-count differences (“why does my dbt version produce 12,847 rows when the proc produces 12,851?”)
6. Discovers the proc has an undocumented  
`WHERE policy_status != 'ARCHIVED' AND created_date >= '2018-01-01'` filter buried on line 982 that her dbt models missed
7. Migrates this one proc in 3 weeks
8. Has 1,399 procs left
9. Realizes the project will take 80 person-years at this rate

### **With procmigrate:**

```
$ procmigrate convert \  
  --source-sql ./sqlserver_procs/  
    usp_calculate_monthly_premium_adjustments.sql \  
  --target-project ./dbt_project \  
  --
```

```

--source-warehouse "sqlserver://prod-readonly" \
--target-warehouse "snowflake://migration-sandbox" \
--layer-strategy auto

[1/8] Parsing T-SQL with sqlglot (sqlserver dialect)...      ✓
      0.7s
[2/8] Building dataflow graph (23 temp tables, 6 sub-proc
      calls)... ✓
[3/8] LLM clustering logic into dbt layers (12 turns)...      ✓
      67s
      Proposed: 4 stg_, 6 int_, 3 fct_ models
[4/8] Generating dbt models with ref()s...                  ✓ 8s
[5/8] Running source proc against migration sandbox...      ✓
      142s
      → 12,851 rows in 8 output tables
[6/8] Running generated dbt models against same sandbox...  ✓
      89s
      → 12,847 rows in 8 output tables (DELTA: 4)
[7/8] LLM iterating on parity (audit_helper diffs)...
      Round 1: 4 row delta in fct_premium_adjustments
              → identified missing filter: policy_status !=
                'ARCHIVED'
              → patched int_active_policies model
      Round 2: 0 row delta. ✓ PARITY ACHIEVED.
[8/8] Generating audit_helper tests + dbt docs...          ✓
      12s

Wrote: models/staging/insurance/stg_policies__active.sql (+ 12
      more)
      tests/audit/
      test_usp_calculate_monthly_premium_adjustments.sql
      analyses/migration_notes/usp_calculate_..._notes.md

Tokens: 184k in / 38k out • Cost: $4.20 • Wall time: 5m 32s

```

**The change:** Priya goes from 3 weeks per proc to 5 minutes (automated) + 1–2 hours (human review of the generated dbt). The 80 person-year estimate becomes 6–9 months. The project actually ships.

This is the **highest-pain, highest-stakes, highest-budget** of the three opportunities. Insurance/banking/healthcare teams will literally pay \$50k–\$500k for this.

## 2. Concrete UX

CLI (primary):

```

## Single proc
procmigrate convert --source-sql proc.sql --target-project ./dbt

```



## Batch (the real use case)

```
procmigrate convert --source-dir ./all_procs --target-project ./
    dbt \
        --parallelism 4
```

## Audit-only mode (re-verify parity for already-migrated procs)

```
procmigrate audit --target-project ./dbt --source-warehouse ...
```

## Plan-only (no warehouse, no LLM execution)

```
procmigrate plan --source-sql proc.sql
```

**Configuration** (procmigrate.yml):

```
version: 1
source:
  dialect: sqlserver # sqlserver | postgres | bigquery | oracle
  connection: ${SOURCE_WAREHOUSE_URL}
  read_only: true # safety
target:
  dbt_project_dir: ./dbt_project
  dialect: snowflake
  layer_naming:
    staging_prefix: stg_
    intermediate_prefix: int_
    mart_prefix: fct_
parity:
  sample_size: 10000 # rows to compare for value diff
  row_count_tolerance: 0 # 0 = exact match required
  value_diff_tolerance: 0.001 # 0.1% per numeric column
  max_iterations: 5
agent:
  model: claude-sonnet-4-7 # or claude-opus-4-7 for hard procs
  max_turns: 30
  human_review_required: true # never commit without --yes
```

**Generated artifacts** for each proc:

```
models/
  staging/insurance/stg_policies__active.sql
  staging/insurance/_stg_policies__schema.yml
  intermediate/int_active_policies_with_premiums.sql
  intermediate/int_premium_adjustments_calculated.sql
  marts/insurance/fct_premium_adjustments.sql
tests/audit/
  test_usp_calculate_monthly_premium_adjustments_row_counts.sql
  test_usp_calculate_monthly_premium_adjustments_value_diff.sql
analyses/migration_notes/
  usp_calculate_monthly_premium_adjustments.md # LLM-written
```

**Migration notes excerpt:**

## **usp\_calculate\_monthly\_premium\_adjustments — migration notes**

**Original:** 1,402 lines T-SQL, 23 temp tables, 4 cursors.

**Translated to:** 13 dbt models across stg\_/int\_/fct\_ layers.

**Key transformations:** 1. Cursor on lines 412–478 (per-policy iteration) → INNER JOIN in int\_premium\_adjustments\_calculated. 2. Temp table #archived\_policies (line 89) → CTE inlined, no materialization needed (used only once). 3. Sub-proc call to usp\_get\_policy\_status (line 612) replaced with ref('stg\_policies\_\_active').

**Behavioral notes for reviewer:** - The original proc had a filter WHERE policy\_status != 'ARCHIVED' on line 982 that wasn't in the docstring. We preserved it in int\_active\_policies. Confirm this filter is still desired. - The cursor in lines 1100–1180 implemented a custom rounding strategy (ROUND(x, 2, 1) — banker's rounding). We translated to Snowflake's default rounding; **verify financial impact**.

**Where it lives:** CLI tool. Optionally a hosted SaaS for the “managed migration” use case (insurance company doesn't want to run this themselves; they want a contractor to deliver migrated dbt + parity proof).

### **3. Technical architecture**

#### **Inputs:**

- Source SQL files (the procs)
- Source warehouse connection (read-only, for parity testing)
- Target dbt project structure (to know naming conventions, package list, existing models)
- Target warehouse connection (for parity testing — must be isolated/sandbox)

#### **Parsing:**

- sqlglot for parse-and-translate (SQL Server → Snowflake dialect)
- Custom AST walkers for:
  - Temp table dataflow → “this temp table feeds these 3 later queries” → candidate for an intermediate model
  - Cursor patterns → “this is iteration; can it be set-based?” → candidate for a JOIN

- Sub-proc calls → “what does this proc return?” → candidate for a `ref()`

**LLM API:** this is the ONE opportunity that genuinely needs an **agent loop**, not a single shot. The shape:

- **Claude Agent SDK** (or Claude Code subprocess) — multi-turn, with tools.
- Tools the agent has access to:
  - `read_sql_file(path)` — read source SQL
  - `query_source_warehouse(sql)` — run a SELECT (read-only)
  - `query_target_warehouse(sql)` — run against migration sandbox
  - `write_dbt_model(name, sql, layer)` — write a .sql file
  - `dbt_compile(model)` — compile and check for errors
  - `dbt_run(model)` — materialize in target sandbox
  - `audit_helper_compare(source_table, target_model)` — row count
    - value diff
  - `read_existing_model(name)` — for `ref()` resolution
- The agent runs ~8-30 turns per proc:
  1. Parse + understand the proc (1-3 turns)
  2. Propose a layer breakdown (1 turn)
  3. Write each model (5-15 turns, one per model)
  4. Run parity check (1 turn)
  5. Iterate on diffs (1-10 turns)
  6. Generate tests + notes (1-2 turns)

**Why this needs an agent and the others don’t:** the iteration loop is intrinsic. You don’t know what the right WHERE clause is until you see the row count diff. You don’t know the temp table is unused until you see the dataflow. The agent’s turn-by-turn back-and-forth with the warehouse IS the value.

### Grounding:

- Source warehouse provides **ground truth** — agent can always verify “does this column exist?”, “how many distinct values?”, “what’s the range?”
- `audit_helper` (the dbt package) provides the comparison primitives.
- Migration-sandbox isolation: target warehouse writes go to a schema named `procmigrate_sandbox_<run_id>` — never touch prod.

### Caching:

- Per-proc cache keyed by source SQL hash + dbt project version
- Re-runs that hit cache: \$0, ~5 seconds (just verifies parity again)

**Cost per invocation:** \$2-\$15 per medium proc (1k-3k lines). \$20-\$50 for monsters (5k+ lines, deep iteration). A team migrating 1,400 procs: **~\$10k-\$30k in API costs total**, vs. 80 person-years of T-SQL labor.

## 4. The hard problems

### Technical risks:

1. **T-SQL → Snowflake SQL has nasty edge cases.** Cursors, temp tables, dynamic SQL, MERGE with non-deterministic WHEN MATCHED, recursive CTEs with ordering. sqlglot covers 80%; the other 20% needs LLM repair.
2. **Procs that mutate the same table multiple times in a single transaction don't translate cleanly to dbt's set-based model.** Need to detect these patterns and either (a) refuse to translate and flag for human, or (b) translate to a sequence of incremental models with explicit unique\_key.
3. **Parity proof requires running the original proc.** If the proc has side effects (writes to multiple tables, sends emails, calls REST APIs from T-SQL), running it in a "sandbox" is non-trivial. Need to detect side-affecting procs and fall back to a different verification strategy.
4. **Long-running procs** that take 4 hours on prod can't be run in a tight iteration loop. Need a "snapshot the source tables once, run dbt against the snapshot, compare to a stored reference output" mode.

### Quality risks:

1. **"Parity" is fuzzy.** Floating point diffs, ordering diffs in array-aggregates, NULL-vs-empty-string. The tolerance config helps, but a proc that produces 99.97% identical output might still have a critical 0.03% bug.
2. **The LLM might find a bug in the original proc** and "fix" it in the dbt translation, breaking parity. Need to surface "we believe the original is wrong because X" explicitly, never silently fix.
3. **Generated dbt models that compile but are unreadable.** A 1,400-line proc translating to 13 dbt models can still produce each individual model that's a 200-line CTE chain. Need a secondary "readability" pass.

### Adoption risks:

1. **Trust gap.** Migrating financial/insurance/healthcare procs with an LLM is terrifying to compliance teams. The audit trail (every diff, every iteration, every LLM decision) is the load-bearing feature for adoption. Without it, this tool is dead in the regulated-industry segment that needs it most.
2. **"Why not just rewrite by hand?"** Some teams will. The pitch has to be cost: \$30k of API + 6 months of human review beats 80 person-years.
3. **Source warehouse access is sensitive.** SQL Server, Oracle, on-prem warehouses often live behind firewalls. Need a self-hosted mode (the OSS CLI works fully offline given the credentials).

### Maintenance risks:

1. **sqlglot dialect coverage.** Each warehouse dialect has its own quirks. sqlglot maintainers are great but not infinite.

2. **dbt package versions** (audit\_helper, dbt\_utils) churn.
3. **Agent loop reliability.** 30-turn agents can derail. Need tight per-turn budgets, recovery from tool errors, and a max-cost circuit breaker.

## 5. Differentiation from incumbents

Tool	Strength	Where it loses
<b>Datafold Migration Agent</b>	Recent product, exact same use case, well-resourced	Closed source, expensive (~\$200k-\$500k engagements), enterprise-sales-gated, slow procurement
<b>AWS SCT (Schema Conversion Tool)</b>	Free, official AWS	Translates DDL well, terrible at procedural code, no dbt awareness
<b>Manual rewrite by consulting firm</b>	High quality (when consultants are good)	\$500k-\$5M, 12-24 months, 50% chance of failure
<b>In-house team</b>	Owens the result	Years of work, attrition risk, deep domain knowledge required
<b>dbt Labs (no offering today)</b>	Could build it tomorrow	Hasn't yet; would likely be Cloud-only

**Honest assessment:** Datafold's Migration Agent is the direct competitor and has a 12+ month head start. The OSS angle is the defensible position — Datafold won't open-source theirs. The mid-market segment (\$1k-\$10k procs to migrate, can't afford Datafold's \$200k engagement) is the wedge.

## 6. Ship-in-a-weekend MVP

**This is the hardest one to MVP** because the value requires end-to-end parity proof. A skeleton without parity verification is just "another LLM that writes SQL."

### In scope (3 days):

- CLI `procmigrate convert --source-sql <one-proc>`
- T-SQL → Snowflake dialect translation via `sqlglot` only (no LLM repair of edge cases yet)
- Naive layer detection: every temp table → intermediate model; every final INSERT → mart model
- Single-shot agent loop (max 5 turns) with a hardcoded set of tools
- `audit_helper` row-count comparison (no value diff yet)
- Output: dbt models + a markdown report
- Snowflake target only

### Out of scope:

- Multi-proc batch mode

- Source dialects other than T-SQL
- Cursor-to-set-based translation (just flag and ask human to rewrite)
- Value diff (row count is the v1 parity proof)
- Migration notes generation (just a stub markdown)

The MVP demo: take a real-world OSS T-SQL proc (Microsoft's sample AdventureWorks or Northwind DB), migrate it, show the parity check passing.

## 7. Path-to-100-users

This one is not a 100-user OSS tool. It's a 10-customer paid product. Different distribution playbook:

1. **Build a public demo** on AdventureWorks. Blog the transcript: "We migrated `dbo.uspGetEmployeeManagers` to dbt in 4 minutes."
2. **Cold outreach to data leaders at insurance/banking/healthcare companies on LinkedIn.** Specifically titles like "VP of Data Engineering" at companies in their dbt-migration RFP phase.
3. **Listing on dbt package hub** as `procmigrate` — the audit tests use `audit_helper`, so it gets visibility there.
4. **Coalesce talk:** "We migrated 1,400 stored procs in 6 months with an LLM agent." (Find a willing customer, do it for free, present the case study.)
5. **Pricing:** free OSS CLI for the engine. Paid SaaS for:
  1. hosted multi-proc dashboard with parity audit log (\$2k/mo), (b)
  - "white-glove migration" managed service (\$50k-\$200k per project — bring in real consultants for the human review).

This is a sales-led motion, not a viral OSS motion. 10 customers at \$50k each is \$500k ARR — bigger than 1000 users at \$10/mo.

## 8. 12-month evolution

- **v1 (month 0-3):** T-SQL → Snowflake, single proc, parity proof. CLI only.
- **v1.5 (month 3-6):** PL/pgSQL + BigQuery procedures. Batch mode. Hosted audit dashboard.
- **v2 (month 6-9):** Oracle PL/SQL (the biggest legacy market — insurers and banks). White-glove managed service launches.
- **v2.5 (month 9-12):** SSIS package translation (the other half of the legacy SQL Server world).

**Moat that compounds:** the parity-test corpus. After 50 migrations you have hundreds of patterns of "the LLM tried X, it failed parity, the fix was Y." That training data makes round-2 migrations 10x faster than round-1 for any new customer.

## 9. Why-not-build-this

- **Datafold has Migration Agent and is shipping it in enterprise deals NOW.** They have a 12-month head start, a sales team, and reference customers.
- **The bear case is brutal:** regulated-industry teams will pay Datafold or a consultancy because they need someone to blame if it goes wrong. An OSS tool can't be blamed.
- **Sales cycles in insurance/banking are 9-18 months.** A solo developer can't fund the runway needed.
- **Per-customer support burden is enormous.** Each customer's T-SQL has unique patterns. The first 5 customers will eat a full developer's time forever.
- **The prize is huge but the field is brutal.** Everyone in data tooling sees this opportunity; many will try.

What kills it: Datafold open-sources their migration engine to neutralize the OSS angle, OR dbt Labs ships a "Migrate from SQL Server" wizard in dbt Cloud and bundles it with existing contracts.

---

### Comparison: which opportunity should the user build?

Dimension	PR Companion	YAML Forge	ProcMigrate
Underservedness	7/10 (Datafold/ Recce serve top + DIY long tail)	8/10 (codegen exists, quality grading doesn't)	6/10 (Datafold + consultancies, but premium-only)
Pain severity	7/10 (annoying, occasional incidents)	6/10 (chronic underinvestment, rarely acute)	10/10 (mission-blocking, \$millions on the line)
Tech feasibility (1 dev)	9/10 (deterministic core + thin LLM)	7/10 (warehouse sandbox + PII + grader)	4/10 (agent loop + parity + dialects)
Distribution clarity	9/10 (Action + dbt Slack + hub)	8/10 (package hub + Slack + hands-on devs)	4/10 (sales-led, long cycles)
Defensibility / moat	6/10 (telemetry compounds; replicable)	7/10 (rubric calibration data compounds)	8/10 (parity corpus + regulated trust)
Synergy with clauditor	5/10 (light — could grade the comments)	<b>10/10 (perfect fit — rubric grading IS clauditor)</b>	6/10 (could grade migration notes)
TOTAL	<b>43/60</b>	<b>46/60</b>	<b>38/60</b>

## **Recommendation: Build YAML Forge (Opportunity 2).**

Three reasons, in order of weight:

1. **Synergy with clauditor is unmatched.** The Phase 3 grader is literally the clauditor harness with a different rubric. You'd be building one product that strengthens your existing product rather than two unrelated codebases. Every improvement to clauditor's grading (better rubrics, faster Haiku graders, tier semantics) directly improves YAML Forge. Every dollar of YAML Forge revenue subsidizes clauditor R&D. The two products share an `_anthropic.py`, share an eval-spec schema, share a testing methodology. This is the only opportunity where "build YAML Forge" effectively means "build clauditor v2 with a vertical wedge."
2. **Feasibility for one developer is genuinely there.** PR Companion is also feasible, but its core value (impact analysis, cost estimation, BI integrations) is mostly not LLM work — it's integration plumbing. A solo dev can ship the LLM part in a weekend but the integrations are a year. YAML Forge's core value IS the LLM-graded generation, which is what you already know how to build well. ProcMigrate is brutal for one person.
3. **The "drop always-pass tests" pruning is a defensible idea that nobody else is talking about.** Every other YAML generator (codegen, Copilot, Paradime) emits everything and leaves curation to the human. The forge premise — "an LLM wrote 600 tests, only 142 are worth keeping, here's why" — is a story. Stories drive adoption. PR Companion's story is "another PR review tool"; ProcMigrate's story is "Datafold but cheaper." Forge's story is unique.

**The honest tradeoff:** Forge has the lowest pain severity of the three (chronic underinvestment in tests is a nag, not a crisis). Pain severity drives urgency, which drives willingness to pay. Forge's monetization will be slower than ProcMigrate's and its user count will grow slower than PR Companion's. But the clauditor synergy is decisive — build it as a vertical demonstration of clauditor's grading methodology applied to a real-world artifact class, and you compound your existing work rather than dilute it.

**Second-best pick:** PR Companion. Build it after Forge proves the clauditor-as-grading-engine pattern works in production. By month 6 you'll have grading infrastructure mature enough to plug into PR Companion's "suggested test rationale" grading. That's the year-2 expansion path.

**Don't build:** ProcMigrate. The TAM is real and the dollars are large, but it's a sales-led, capital-intensive product that requires a team. It's the right business for Datafold, not for a solo developer with an existing OSS project to grow.



# Appendix F — Claude + DBT

## Technical Surface

### dbt + Claude: Technical Surface for 2026

Reference material for an implementer building Claude-powered dbt tooling. Focuses on artifact schemas, MCP server surface, SDK options, warehouse integration, CI shapes, and token economics. Not narrative — scan-and-grep oriented.

Companion to docs/temp/dbt-tooling-opportunity-report.md (product framing). This doc is the “what do I actually have to integrate against” answer.

---

#### 1. dbt artifact deep dive

All artifacts land in target/ after a dbt invocation. Their schemas are versioned at <https://schemas.getdbt.com/dbt/<artifact>/<version>/index.json>.

##### 1.1 manifest.json — the project graph

The single most important artifact. Contains a complete representation of every node in the project plus parent/child dependency maps.

##### Version-to-dbt-version mapping:

dbt Core version	Manifest schema	Notes
1.5	v9	
1.6	v10	
1.7	v11	
1.8	v12	First “stable” version for many integrations
1.9 / 1.10 / 1.11	v12	No schema bump “Identical to v12” per docs — same shape, different namespace
Fusion (v2.0)	v20	

Fusion’s v20 means **manifest-shape compatibility is preserved across the Core/Fusion split**, but Fusion-only fields (e.g. column-level lineage from static analysis) appear as additive properties. Core ≠ Fusion at the engine level even when manifests look the same; cross-environment Recce-style diffs break if you mix engines.

##### Top-level keys (v12):

```

{
  "metadata": {
    "dbt_schema_version": "https://schemas.getdbt.com/dbt/manifest/v12.json",
    "dbt_version": "1.11.6",
    "generated_at": "2026-04-24T...",
    "invocation_id": "uuid",
    "project_id": "hash",
    "project_name": "my_project",
    "user_id": "uuid",
    "adapter_type": "snowflake",
    "env": {},
    "send_anonymous_usage_stats": true
  },
  "nodes": { /* models, seeds, snapshots, tests, analyses,
operations, hooks */ },
  "sources": { /* source definitions from sources.yml */ },
  "macros": { /* every macro, including from packages */ },
  "docs": { /* doc blocks */ },
  "exposures": {},
  "metrics": {},          // v1.6+ MetricFlow
  "groups": {},           // v1.5+
  "selectors": {},
  "disabled": {},
  "parent_map": { "model.x.foo": ["source.x.bar"], ... },
  "child_map": { "source.x.bar": ["model.x.foo"], ... },
  "group_map": {},
  "saved_queries": {},    // v1.7+
  "semantic_models": {},  // v1.6+
  "unit_tests": {}        // v1.8+
}

```

### **Per-node fields that matter for an LLM tool:**

```

"model.my_project.dim_users": {
  "resource_type": "model",
  "unique_id": "model.my_project.dim_users",
  "name": "dim_users",
  "fqdn": ["my_project", "marts", "dim_users"],
  "database": "ANALYTICS",
  "schema": "MARTS",
  "alias": "dim_users",
  "package_name": "my_project",
  "path": "marts/dim_users.sql",
  "original_file_path": "models/marts/dim_users.sql",
  "checksum": { "name": "sha256", "checksum": "..." },
  "config": {
    "materialized": "table",
    "on_schema_change": "fail",
    "contract": { "enforced": true, "alias_types": true },
    "pre-hook": [], "post-hook": [],
    "tags": ["pii"], "meta": {}, "grants": {}
  }
}

```

```

},
"tags": ["pii"],
"description": "...",
"columns": {
  "user_id": {
    "name": "user_id",
    "data_type": "varchar(36)",
    "description": "...",
    "constraints": [{"type": "not_null"}, {"type": "unique"}],
    "meta": {}, "tags": []
  }
},
"depends_on": {
  "macros": ["macro.dbt.statement"],
  "nodes": ["source.my_project.raw.users"]
},
"refs": [{"name": "stg_users", "package": null, "version":
null}],
"sources": [{"raw", "users"}],
"compiled": true,
"compiled_code": "select ...", // post-Jinja, pre-warehouse
"raw_code": "{{ config(...) }} select ...",
"language": "sql",
"access": "protected", // v1.5+ public/private/protected
"version": null, // v1.5+ model versioning
"latest_version": null,
"constraints": [], // table-level constraints
"deprecation_date": null,
"primary_key": ["user_id"]
}

```

### Parsing gotchas:

- `manifest.json` is a snapshot; always run `dbt parse` (cheap, no warehouse) or `dbt compile` (more expensive, renders Jinja) before reading.
- `compiled_code` is `null` until compilation has run. After `dbt parse`, only `raw_code` is populated.
- `nodes` is **flat** — every resource type lives in this dict, distinguished by `resource_type`. Tests, seeds, snapshots, models, analyses, operations, and hooks all collide on key prefix (`model.x.y`, `test.x.y`, `seed.x.y`, `snapshot.x.y`, `analysis.x.y`).
- `disabled` is a parallel dict — disabled nodes do NOT appear in `nodes`. If you need full project visibility, walk both.
- `parent_map` / `child_map` are pre-computed by dbt; do NOT rebuild from `depends_on.nodes` (you'll miss source/macro edges).
- Source code: [dbt-labs/dbt-core/core/dbt/contracts/graph/manifest.py](https://github.com/dbt-labs/dbt-core/blob/main/core/dbt/contracts/graph/manifest.py) defines `WritableManifest`.
- Schema: <https://schemas.getdbt.com/dbt/manifest/v12.json>.

### Size in the wild:

Project size	Models	Approx manifest.json size	Approx Claude tokens
Small	~100	500 KB - 2 MB	100k - 400k
Medium	~500	5 - 15 MB	1M - 3M
Large	~2000	30 - 100+ MB	6M - 20M+

A medium-to-large manifest **does not fit in a 1M context window** raw. Strategies: (a) pre-extract a subset to relevant `unique_ids`, (b) stream summary records via tool calls, (c) feed only `parent_map/child_map` for graph reasoning then fetch node details on demand.

## 1.2 catalog.json — warehouse reality

Generated by `dbt docs generate`, which queries `information_schema` (or adapter equivalent). Documents what physically exists in the warehouse for every `model` | `seed` | `snapshot` | `source` in the manifest.

```
{
  "metadata": { /* same shape as manifest metadata */ },
  "nodes": { "model.x.dim_users": {...} },
  "sources": { "source.x.raw.users": {...} },
  "errors": null // populated if information_schema queries failed
}
```

Per-entry shape:

```
{
  "metadata": {
    "type": "TABLE",
    "schema": "MARTS",
    "name": "DIM_USERS",
    "database": "ANALYTICS",
    "comment": null,
    "owner": "TRANSFORMER"
  },
  "columns": {
    "USER_ID": {
      "type": "VARCHAR(36)",
      "index": 1,
      "name": "USER_ID",
      "comment": null
    }
  },
  "stats": {
    "row_count": { "id": "row_count", "label": "Row Count",
    "value": 1234567, "include": true },
    "bytes": { ... }
  },
  "unique_id": "model.x.dim_users"
}
```

## Differs from manifest:

- Manifest = “what dbt thinks it’s building.” Catalog = “what’s actually in the warehouse.”
- Catalog requires a warehouse round-trip; manifest does not.
- Catalog column casing follows warehouse conventions (Snowflake = upper).
- Catalog is the only way to detect actual schema drift between dbt’s declared columns and warehouse reality.

### 1.3 run\_results.json — execution outcomes

Written after dbt run | test | build | seed | snapshot | compile.

```
{
  "metadata": { /* ... */ },
  "results": [
    {
      "status": "success | error | skipped | pass | fail | warn |
runtime error",
      "timing": [
        { "name": "compile", "started_at": "...", "completed_at":
"..."},
        { "name": "execute", "started_at": "...", "completed_at":
"..."}
      ],
      "thread_id": "Thread-1",
      "execution_time": 12.34,
      "adapter_response": {
        "_message": "SUCCESS 1",
        "rows_affected": 1234567,
        "code": "SUCCESS",
        "bytes_processed": 89012345,    // BigQuery
        "query_id": "...",             // Snowflake
      },
      "message": "OK created sql table model
ANALYTICS.MARTS.DIM_USERS [SUCCESS 1 in 12.34s]",
      "failures": null,                 // populated on tests
      "unique_id": "model.x.dim_users",
      "compiled": true,
      "compiled_code": "select ...",
      "relation_name": "ANALYTICS.MARTS.DIM_USERS"
    }
  ],
  "elapsed_time": 123.45,
  "args": { /* invocation args */ }
}
```

**For CI:** join on unique\_id against the manifest to attribute every failure / slow run to a specific model file. Test failures populate failures (count) and message (sample failing rows when configured).

## 1.4 sources.yml / schema.yml — author-controlled YAML

Not artifacts — they're inputs. dbt parses these into `manifest.json` under nodes (tests) and sources. Author surface:

```
## models/marts/_marts__models.yml
version: 2
models:
  - name: dim_users
    description: "..."
    access: public # 1.5+
    config:
      contract: {enforced: true}
      tags: [pii]
    columns:
      - name: user_id
        data_type: varchar(36)
        description: "..."
        constraints:
          - type: not_null
          - type: unique
        data_tests:
          - unique
          - not_null
          - relationships:
              to: ref('stg_users')
              field: user_id

## models/staging/_sources.yml
sources:
  - name: raw
    schema: RAW
    tables:
      - name: users
        loaded_at_field: _ingested_at
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 24, period: hour}
        columns:
          - name: id
            data_type: string
```

Evolution: 1.5 added access and groups. 1.6 added MetricFlow metrics and semantic\_models. 1.7 added saved\_queries, dropped Python 3.7. 1.8 added unit\_tests and renamed tests: to data\_tests: (both still parse). Fusion adds stricter data\_type validation via static analysis.

## 1.5 sources.json — freshness results

Generated only by `dbt source freshness`. Schema v3 today. Per-result:

```
{
  "unique_id": "source.x.raw.users",
  "max_loaded_at": "2026-04-24T14:00:00Z",
  "snapshotted_at": "2026-04-24T14:05:00Z",
  "max_loaded_at_time_ago_in_s": 300,
  "criteria": {"warn_after": {...}, "error_after": {...}},
  "status": "pass | warn | error | runtime error",
  "execution_time": 0.42
}
```

## 1.6 graph.gpickle — the compiled DAG

Pickled `networkx.DiGraph` of `unique_id -> unique_id` edges. Not stable across Python or `networkx` versions; consumers should rebuild from `manifest.parent_map / child_map` rather than depend on the `gpickle`. Fusion may not emit it at all in some configurations.

## 1.7 partial\_parse.msgpack — the perf cache

`dbt`'s incremental-parse cache. Stores hashed config + last-parsed manifest in `msgpack` format under `target/`. **Do not read this from a downstream tool** — internal format, not versioned for external consumers. `dbt clean` deletes it. Useful only as a signal that partial parsing is enabled (skips re-parsing unchanged files between invocations).

## 1.8 Other artifacts

- **semantic\_manifest.json** — `MetricFlow`'s separate manifest for semantic models / metrics. Required for the `dbt Semantic Layer`.
- **graph\_summary.json** — Fusion-only condensed graph for fast loading.
- **.user.yml** — `dbt Cloud`'s per-user UUID file.

---

## 2. dbt MCP Server (dbt-labs/dbt-mcp)

Apache 2.0, Python, ~96% Python codebase. Two deployment shapes: **local** (`uvx dbt-mcp`) and **remote** (HTTP, hosted by `dbt Labs`). The local form is what an OSS CI tool would target.

### 2.1 Tool surface

50+ tools in 8 categories. Names and intent:

**SQL & Semantic Layer (8):** `execute_sql`, `text_to_sql`, `get_dimensions`, `get_entities`, `get_metrics_compiled_sql`, `list_metrics`, `list_saved_queries`, `query_metrics`.

**Discovery API (16+):** `get_all_macros`, `get_all_models`, `get_all_sources`, `get_exposure_details`, `get_exposures`, `get_lineage`, `get_macro_details`, `get_mart_models`, `get_model_children`, `get_model_details`, `get_model_health`, `get_model_parents`, `get_model_performance`, `get_related_models`, `get_seed_details`, `get_semantic_model_details`, `get_snapshot_details`, `get_source_details`, `get_test_details`, `search`. **The `get_*` family reads from dbt Platform's Discovery API, not local artifacts** — requires `DBT_HOST` + `DBT_TOKEN`.

**dbt CLI (9):** `build`, `clone`, `compile`, `docs`, `list`, `parse`, `run`, `show`, `test` plus `get_lineage_dev`, `get_node_details_dev` for local-only manifest reads.

**Admin API (10):** `cancel_job_run`, `get_job_details`, `get_job_run_details`, `get_job_run_error`, `list_job_run_artifacts`, `list_jobs`, `list_jobs_runs`, `list_projects`, `retry_job_run`, `trigger_job_run`. dbt Cloud only.

**Code Generation (3):** `generate_model_yaml`, `generate_source`, `generate_staging_model`. Wraps the `dbt-codegen` package.

**Fusion / LSP (3):** `fusion.compile_sql`, `fusion.get_column_lineage`, `get_column_lineage`.

**Product Docs (2):** `get_product_doc_pages`, `search_product_docs`.

**Metadata (2):** `get_mcp_server_branch`, `get_mcp_server_version`.

## 2.2 Auth

Local CLI category requires:

Var	Purpose
<code>DBT_PROJECT_DIR</code>	Absolute path to dir containing <code>dbt_project.yml</code>
<code>DBT_PATH</code>	Absolute path to the dbt executable
<code>DBT_PROFILES_DIR</code>	Optional, defaults to <code>~/.dbt/</code>
<code>DBT_CLI_TIMEOUT</code>	Seconds, default 60

dbt Platform (Discovery, Admin, Semantic Layer, SQL):

Var	Purpose
<code>DBT_HOST</code>	e.g. <code>cloud.getdbt.com</code>
<code>DBT_TOKEN</code>	Service token or PAT (PAT required for <code>execute_sql</code> )
<code>DBT_PROD_ENV_ID</code>	Numeric environment ID
<code>DBT_DEV_ENV_ID</code>	Required for <code>execute_sql</code>
<code>DBT_USER_ID</code>	Required for <code>execute_sql</code>
<code>DBT_ACCOUNT_ID</code>	Required for Admin API + PAT auth

**Tool category gating:** by default everything is enabled except SQL, Codegen, and metadata. Disable with `DISABLE_DBT_CLI=true` etc. Switch to



allowlist mode with `DBT_MCP_ENABLE_*=true` (the moment you set ANY enable flag, only enabled categories are active).

## 2.3 Install + invoke

```
## Local install via uv
uvx dbt-mcp                # one-shot run, fetches latest
uvx --from dbt-mcp@1.x dbt-mcp  # pin version

## Or via the MCPB bundle
mcpb install dbt-mcp.mcpb      # Anthropic's bundle installer
```

Claude Desktop config (`~/Library/Application Support/Claude/claude_desktop_config.json` or platform equivalent):

```
{
  "mcpServers": {
    "dbt": {
      "command": "uvx",
      "args": ["dbt-mcp"],
      "env": {
        "DBT_PROJECT_DIR": "/path/to/dbt_project",
        "DBT_PATH": "/usr/local/bin/dbt",
        "DISABLE_DBT_CLI": "false",
        "DISABLE_SEMANTIC_LAYER": "true",
        "DISABLE_DISCOVERY": "true",
        "DISABLE_ADMIN_API": "true"
      }
    }
  }
}
```

Claude Code uses the same `mcpServers` schema in `.mcp.json` at the project root, or via `claude mcp add`.

Anthropic SDK invocation: spawn `uvx dbt-mcp` as a stdio child, speak MCP over stdin/stdout. Or set `MCP_TRANSPORT=streamable-http` for an HTTP transport (useful for debugging).

## 2.4 What's missing for an interesting CI tool

- **No diff / impact-analysis tools.** `get_lineage` returns the lineage as-of-now, not “what changed between this PR and main.” Recce-style PR diffs are not in the surface.
- **No data-sample tool with safety controls.** `execute_sql` is raw — no PII redaction, no row-cap enforcement, no warehouse-side sampling helpers. Have to wrap.
- **No PR / GitHub awareness.** MCP server doesn't know about CI context; that surface lives outside.

- **No artifact-version awareness.** Tools assume “the manifest” — no helper for “compare manifest A vs manifest B.”
- **No streaming for large results.** `get_all_models` on a 2k-model project returns a single massive blob.
- **Authentication is per-process.** No way to swap credentials per request — important if you want one server instance to serve multiple repos / accounts.

## 2.5 Stability assessment

dbt-mcp moved to v1.0 in late 2025. Tool names and env-var contract have been stable since. Discovery/Admin tools are tightly coupled to dbt Cloud’s GraphQL API (which IS stable but evolves). LSP/Fusion tools are newer and still labelled experimental. **For an OSS CI tool, depend on the local-CLI subset and the basic Discovery tools; treat LSP/Fusion as moving targets.**

---

## 3. Anthropic SDK / Claude Agent SDK / Claude Code surface

### 3.1 Anthropic SDK (raw)

Direct `messages.create` with tool-use:

```
from anthropic import Anthropic

client = Anthropic()
resp = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=4096,
    tools=[
        {
            "name": "read_manifest",
            "description": "Read a slice of manifest.json by unique_id.",
            "input_schema": {
                "type": "object",
                "properties": {"unique_id": {"type": "string"}},
                "required": ["unique_id"],
            },
        },
    ],
    messages=[{"role": "user", "content": "..."}],
)
## Loop on resp.stop_reason == "tool_use", append tool_result,
recurse.
```

Pros: full control, lowest token overhead per turn, minimal dependencies.  
Cons: you write the agent loop, the file I/O, the parallel-tool-use coordination.

### 3.2 Claude Agent SDK

Higher-level wrapper that gives you Claude Code as a library. Subagents, sessions, hooks, and a built-in tool execution loop. Python and TypeScript.

```
from claude_agent_sdk import ClaudeSDKClient, ClaudeAgentOptions

options = ClaudeAgentOptions(
    system_prompt="You are a dbt CI reviewer...",
    allowed_tools=["Read", "Grep", "Bash(dbt parse:*)",
                  "Bash(dbt compile:*)"],
    max_turns=20,
    permission_mode="acceptEdits",
)
async with ClaudeSDKClient(options) as client:
    await client.query("Review manifest changes between main and HEAD")
    async for message in client.receive_response():
        ... # stream events
```

**Subagents** — define in `.claude/agents/<name>.md` with frontmatter (name, description, tools, model). Main agent invokes via the Task tool; each subagent runs in its own context window with its own permissions. Multiple subagents run in parallel for read-only work.

**Tool annotations** — `readOnlyHint: true` lets Claude batch tools for parallel execution. Important for reading many model files at once.

### 3.3 Claude Code as a subprocess (clauditor's pattern)

Spawn `claude -p "<prompt>" --output-format stream-json --verbose`, read NDJSON from stdout. `clauditor's runner (src/clauditor/runner.py)` is the reference shape — defensive parsing per `.claude/rules/stream-json-schema.md`, four-layer error classification, watchdog timeout. See also `docs/transport-architecture.md`.

**Pros for a CI tool:** zero SDK dependency, leverages user's existing Claude Code config (subagents, MCP servers, permissions), inherits subscription auth (Pro/Max/Teams) without an API key. Same auth surface as developer's local Claude Code.

**Cons:** requires `claude` binary on the runner image, larger surface to mock for tests, subprocess overhead per invocation (~1-2s warm-up), output is stream-json which you parse defensively. Per-call latency is higher than direct SDK.

### 3.4 Token cost considerations

Three things blow up the token budget on a dbt tool:

1. **manifest.json itself.** Already covered: medium projects don't fit raw. Mitigations:
  - Build a thin "node summary" structure (just `unique_id`, `resource_type`, `tags`, `depends_on.nodes`, `description`, `column count`) and feed that as the index.
  - Use tool calls to fetch full node details on demand.
  - Cache the compact summary as a static prompt prefix (see prompt caching below).
2. **Model SQL files.** A 2000-model project has 2000 .sql files, average 50 lines each. Reading them all is 1-3M tokens. Use `grep` / `glob` to narrow first, then Read only the relevant files.
3. **Catalog data** scaling with column count. A 100-model warehouse with avg 20 columns is 2000 column rows; mostly fine to load fully, but skip catalog if you don't need warehouse reality.

### 3.5 Prompt caching

5-minute cache writes cost 1.25x base input; 1-hour cache writes cost 2x; cache reads cost 0.1x. **Caching pays off after one hit at the 5-min tier and two hits at the 1-hour tier.**

For a dbt tool, the caching candidate is the static project context — manifest summary, project conventions doc, style guide. Mark up to 4 cache breakpoints in your prompt; everything before the breakpoint is cached:

```
client.messages.create(  
  model="claude-sonnet-4-6",  
  system=[  
    {"type": "text", "text": "You are a dbt CI reviewer..."},  
    {  
      "type": "text",  
      "text": MANIFEST_SUMMARY,  
      "cache_control": {"type": "ephemeral"}, # 5-min  
      default  
    },  
  ],  
  messages=[...],  
)
```

A 200k-token project context cached at 1-hour TTL: write costs  $\$3/\text{M} * 0.2\text{M} * 2 = \$1.20$  once. Subsequent reads in the same hour:  $\$3/\text{M} * 0.2\text{M} * 0.1 = \$0.06$  per request. Run 50 PR reviews in an hour:  $\$1.20 + 50 * \$0.06 = \$4.20$  vs. uncached  $\$3/\text{M} * 0.2\text{M} * 50 = \$30.00$ . **86% savings.**

Important: starting Feb 2026 caches are isolated per workspace (Claude API + Azure AI Foundry). Bedrock and Vertex remain org-isolated.

### 3.6 Model picks for a dbt CI tool

Tier	Model	Use case
Cheap & fast	Haiku 4.5 (\$1/\$5 per 1M)	Single-file lint, deterministic checks, structured extraction
Workhorse	Sonnet 4.6 (\$3/\$15)	PR review, diff explanation, test proposal
Reasoning	Opus 4.7 (\$5/\$25)	Cross-cutting refactor analysis, root-cause for cascading test failures

For a clauditor-style L2 (extraction) + L3 (rubric grading) split: Haiku does L2, Sonnet does L3. Opus stays on a manual escalation tier.

---

## 4. Warehouse integration patterns

A dbt-aware tool wants to see data, not just metadata, for: schema diffs, profile diffs (Recce-style), value-distribution comparisons, sample-row inspection.

### 4.1 Reading via dbt itself

```
dbt show --inline "select * from {{ ref('dim_users') }} limit 10"
dbt run-operation print_models
```

**Pros:** uses the user's existing `profiles.yml` — no new credentials path, inherits dbt's connection pooling. **Cons:** subprocess overhead per query, 60s default timeout, awkward for bulk profiling. Fine for CI sample queries; bad for interactive exploration.

### 4.2 Direct adapter access

```
import snowflake.connector
conn = snowflake.connector.connect(**creds)
cur = conn.cursor()
cur.execute("SELECT * FROM ANALYTICS.MARTS.DIM_USERS TABLESAMPLE
            (1000 ROWS)")
```

Adapter libraries: `snowflake-connector-python`, `google-cloud-bigquery`, `databricks-sql-connector`, `psycopg2` / `psycopg` for Postgres/Redshift, `duckdb` for local.

**Pros:** fast, full SQL surface, parameterized queries. **Cons:** new credentials path (don't reuse `profiles.yml` unless you parse it), per-warehouse SQL dialect differences for sampling, you own the connection lifecycle.

### 4.3 Reading via dbt MCP semantic layer

`query_metrics`, `get_dimensions` — operates on declared metrics in the Semantic Layer. **Pros:** governed, semantic-aware, no SQL injection surface. **Cons:** Semantic Layer is a dbt Cloud feature, OSS users won't have it, only works for declared metrics.

### 4.4 Sampling strategies

#### Warehouse Sampling syntax

Snowflake	<code>TABLESAMPLE (1000 ROWS) or TABLESAMPLE BERNOULLI (1)</code>
BigQuery	<code>TABLESAMPLE SYSTEM (1 PERCENT)</code>
Databricks	<code>TABLESAMPLE (1000 ROWS)</code>
Postgres	<code>TABLESAMPLE BERNOULLI (1)</code>
Redshift	No native; use <code>WHERE random() &lt; 0.01</code>

For diff workloads also constrain by time: `WHERE _ingested_at > current_date - 7`. Sampling for an LLM context is bounded by **token budget per row × row count**; 100 rows × 20 cols × ~5 tokens = 10k tokens per sample is a reasonable target.

### 4.5 PII / safety

The single biggest deployment blocker. Options:

- **Schema-only mode.** Never query data; only show column names + types from catalog.
- **Aggregate-only.** `count`, `count(distinct)`, `min`, `max`, `null_count`, value distribution buckets — no raw values.
- **Tag-based redaction.** Honor `dbt meta.pii: true` or `tags: [pii]` and refuse to fetch those columns. Models supporting this: `dbt-snow-mask`, `dbt-tags`, plus Snowflake's `AI_REDACT` Cortex function for unstructured text.
- **Explicit allowlist.** Only sample from `models/marts/safe_for_review/`.
- **Dynamic Data Masking** at the warehouse layer (Snowflake masking policies). The cleanest answer; the tool runs as an LLM-bot role that sees masked values for any tagged column.
- **Client-side redaction.** Run regex-based PII detectors over fetched rows before the LLM sees them. Snowflake's `AI_REDACT` is a server-side equivalent.

For a public-CI shape, **default to schema-only with explicit opt-in for sampling** is the only defensible posture.

---

## 5. CI/CD shapes

### 5.1 GitHub Action

Two-workflow pattern (Recce's shape): a base-branch workflow uploads target/ artifacts; a PR workflow downloads the base and diffs. Reference `action.yml`:

```
## .github/actions/dbt-claude-review/action.yml
name: dbt Claude Review
description: LLM-powered dbt PR review
inputs:
  dbt-project-dir:
    required: true
    default: '.'
  base-artifact:
    required: false
    description: 'Name of the base manifest artifact to download'
  anthropic-api-key:
    required: false
  claude-binary:
    required: false
    default: 'claude'
runs:
  using: composite
  steps:
    - uses: actions/checkout@v4
    - name: Install dbt
      shell: bash
      run: pip install dbt-core dbt-snowflake
    - name: Parse PR manifest
      shell: bash
      working-directory: ${ inputs.dbt-project-dir }
      run: dbt parse
    - name: Download base manifest
      if: inputs.base-artifact != ''
      uses: actions/download-artifact@v4
      with:
        name: ${ inputs.base-artifact }
        path: target-base
    - name: Run review
      shell: bash
      env:
        ANTHROPIC_API_KEY: ${ inputs.anthropic-api-key }
      run: |
        my-tool review \
          --pr-manifest target/manifest.json \
          --base-manifest target-base/manifest.json \
```

```

    --pr-number ${github.event.pull_request.number} \
    --repo ${github.repository}
- name: Post comment
  if: github.event_name == 'pull_request'
  uses: actions/github-script@v7
  with:
    script: |
      const fs = require('fs');
      const body = fs.readFileSync('review.md', 'utf8');
      await github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body
      });

```

Base-branch workflow is symmetric: `dbt parse` → `actions/upload-artifact@v4` with name: `manifest-main`.

**Pros:** native to GH, free for OSS, integrates with status checks / PR comments / annotations. **Cons:** GitHub-only.

## 5.2 GitLab CI

```

dbt_claude_review:
  image: python:3.12
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  script:
    - pip install dbt-core dbt-snowflake my-tool
    - dbt parse
    - my-tool review --mr $CI_MERGE_REQUEST_IID
  artifacts:
    paths: [target/]
    reports:
      codequality: review.json

```

Use the GitLab Code Quality report shape for inline annotations. MR comments via `python-gitlab` against `CI_API_V4_URL`.

## 5.3 Pre-commit hook

```

## .pre-commit-hooks.yaml
- id: dbt-claude-lint
  name: dbt Claude lint
  entry: my-tool lint
  language: python
  files: '\.(sql|yaml)$'
  pass_filenames: true

```



**Pros:** local, fast, no CI roundtrip. **Cons:** no warehouse access in pre-commit (security), so limited to manifest/text checks. Reference: `dbt-checkpoint` (formerly `pre-commit-dbt`) is the canonical OSS hook collection.

## 5.4 dbt Cloud webhook

dbt Cloud emits webhooks on job completion (`run.completed`, `run.errorred`). A Lambda / Cloud Run / Modal endpoint receives the webhook, downloads `manifest.json` + `run_results.json` via the Admin API, runs the review, posts to wherever.

**Pros:** triggered by real production runs, has access to Admin API artifacts.  
**Cons:** requires hosted compute, dbt Cloud only.

## 5.5 Standalone CLI

Plain `pip install` + invocation from any CI. The most portable shape and the lowest-coupling. `clauditor`'s CLI is the model.

## 5.6 Pros/cons summary

Shape	Reach	Setup cost	Warehouse access	Where output goes
GitHub Action	GH only	Low (action.yml)	Yes (secrets)	PR comment / check
GitLab CI	GL only	Low	Yes	MR note / report
Pre-commit	Universal	Trivial	No	Terminal
dbt Cloud webhook	Cloud users only	Med (deploy endpoint)	Via Admin API	Anywhere
Standalone CLI	Universal	Trivial	Yes	Stdout / file

## 6. Output integration

Channel	API surface	Notes
GitHub PR comment	<code>gh pr comment &lt;num&gt; --body-file review.md</code> or <code>actions/github-script@v7</code> (Octokit)	Markdown supported, 65k char limit per comment
GitHub PR review (multi-thread)	<code>POST /repos/{o}/{r}/pulls/{n}/reviews</code> with <code>comments: [{path, line, body}]</code>	Inline annotations on diff lines
GitHub check run	<code>POST /repos/{o}/{r}/check-runs</code> with <code>output.annotations</code>	Pass/fail status + line-anchored annotations; max 50 annotations per request

Channel	API surface	Notes
GitHub job summary	echo "... " >> \$GITHUB_STEP_SUMMARY in the runner	Free-form Markdown on the workflow run page
GitLab MR note	POST /projects/:id/merge_requests/:iid/notes	Same shape as PR comments
GitLab MR diff comment	POST /projects/:id/merge_requests/:iid/discussions with position	Inline on diff
Status check	GitHub: check run conclusion; GitLab: MR pipeline status	Required-check protection rules use this
Artifact upload	actions/upload-artifact@v4, GitLab artifacts:	HTML report, JSON sidecar — survive after run
Slack	Webhook URL or chat.postMessage	Rich blocks, threading
Linear	IssueCreate GraphQL mutation	Auto-file follow-up issues
Email	SES, SMTP	Last-resort, non-interactive

**Recommended default for a CI tool:** check run with summary + annotations (gives both pass/fail status and inline comments) plus a job-summary write for free-form analysis. Gate behavior: never block merge in default configuration; add --required flag for explicit promotion.

---

## 7. Existing OSS templates worth stealing from

Project	Repo	Architectural lesson
Recce	DataRecce/recce	Two-workflow PR-vs-base diff via uploaded artifacts. TypeScript UI + Python backend; CLI for CI (recce server, recce-cloud upload). Profile/Value/Top-K/Histogram diffs are the lingua franca. PR gating via Cloud.
Elementary	elementary-data/elementary + elementary-data/dbt-data-reliability	dbt package writes test results / metadata to warehouse tables; CLI generates HTML reports + Slack alerts. The “package + CLI +

Project	Repo	Architectural lesson
dbt-osmosis	z3z1ma/dbt-osmosis	GitHub Action” trio is a copy-worthy template. YAML inheritance / propagation. Reads manifest, mutates schema.yml. Streamlit workbench for interactive exploration. The “diff schema vs warehouse + propose YAML edits” pattern. Staging-model generation (yaml + sql) from sources. Templating-heavy.
dbt-coves	datacoves/dbt-coves	The canonical pre-commit hook collection. ~30 hooks for column descriptions, model tags, source freshness. Reads manifest.json for context.
dbt-checkpoint	dbt-checkpoint/dbt-checkpoint	Shape for a long-running review agent with subagents and parallel tool use.
Anthropic agent-sdk examples	anthropics/claude-agent-sdk-python examples dir	Patterns for agents that operate inside Actions — secret handling, artifact lifecycle, comment posting.
GitHub agent-toolkit	github/agent-toolkit	

### Patterns to steal:

- Recce’s two-workflow base/PR artifact handoff.
- Elementary’s “dbt package writes reproducible state to warehouse tables” — gives you historical context without re-querying.
- dbt-osmosis’s “read manifest, mutate YAML files, run formatter” loop.
- dbt-checkpoint’s permissioning model — pre-commit hooks gate on local-only checks; warehouse-touching checks defer to CI.

### Patterns NOT to steal:

- Heavy UI bundled with the OSS distribution (Recce’s TS frontend doubles install footprint).
  - Tight coupling to a hosted backend (limits OSS usability).
-

## 8. Token economics

### 8.1 Reading manifest.json

Compact summary (one line per node, ~50 tokens):

#### **Project Models Compact summary tokens**

Small	100	~5k
Medium	500	~25k
Large	2000	~100k

Full manifest.json (raw):

#### **Project Models Full manifest tokens**

Small	100	~100k - 400k
Medium	500	~1M - 3M
Large	2000	~6M - 20M+

**Operational rule:** never load raw manifest into context past the small tier. Build a summary, cache it, and use tool calls for drill-down.

### 8.2 Reading model SQL files

Average model: 30-100 lines, ~500-1500 tokens. Reading 10 changed models in a PR: 5-15k tokens. Reading the upstream/downstream impact set (avg 20 nodes): 10-30k tokens.

### 8.3 PR comment output

A typical clauditor-style review comment runs 1500-3000 output tokens. At Sonnet 4.6 output (\$15/M), that's \$0.022-\$0.045 per comment.

### 8.4 Per-PR cost estimates

Assumptions: medium project (500 models), 10 changed files, prompt caching enabled, Sonnet 4.6 main reviewer + Haiku for L2 extraction.

Phase	Tokens	Cached?	Cost
System prompt + project context	25k	Yes (1-hr cache, write once per hour)	Write: \$0.15 ; reads after first: \$0.0075 each
PR diff + changed file SQL	15k input	No	$3 * 0.015 = \$0.045$
Tool-call drill-down (avg 5 calls)	10k input	No	\$0.030
Output comment	2k output	n/a	$15 * 0.002 = \$0.030$
			<b>~\$0.11 / PR</b>

Phase	Tokens	Cached?	Cost
Per-PR total (cache hit)			
Per-PR total (cache miss)			~\$0.26 / PR

Active team doing 100 PRs/month: **\$11-\$26/month** at Sonnet rates. Add Opus escalation for ~10% of PRs at \$0.50 each → **+\$5/month**. Total roughly **\$15-\$30/month per active dbt repo**. Cheap enough that the bottleneck is human review time, not token budget.

Larger projects (2000 models): summary cost rises to 100k tokens, cached at \$0.60/write, \$0.030/read. Per-PR rises to \$0.40-\$0.80. Still ~\$50-\$100/month per active large repo.

## 8.5 Eval pass with rubric grading

A clauiditor-style eval (rubric-graded by Sonnet, with 5-10 criteria) runs 8-15k input tokens + 2-3k output. Each pass: ~\$0.05. A 20-skill regression suite: ~\$1.

---

## 9. Open questions / unknowns

### 9.1 Worth a spike before committing

- **Manifest size handling at the medium-to-large tier.** Need to prototype the compact-summary builder + tool-call drill-down loop end-to-end on a real 500-model project to confirm the token math.
- **Cache TTL behavior across CI runs.** Cached prompts only survive the cache TTL (5-min or 1-hour) AND require workspace-level continuity (post-Feb-2026 isolation). A burst of 50 PRs in 5 minutes amortizes well; a single PR/week pays full price every time. Need to measure.
- **dbt-mcp vs direct artifact reads.** Open question whether the MCP server adds enough value over `json.load(open("target/manifest.json"))` to justify the dependency. The Discovery API tools are useful only for dbt Cloud users; the local-CLI tools mostly wrap `subprocess.run`. A spike comparing both for the canonical “diff PR vs main” workflow would settle it.
- **Subagent fan-out for parallel model review.** Subagents run in parallel — does that scale to “review 20 changed models concurrently with separate context windows”? Token cost is per-subagent (no shared cache between siblings unless explicitly architected).
- **PII safety posture default.** Schema-only is safe but boring; sampling is interesting but risky. Need a clear opt-in flow.

## 9.2 What dbt Fusion changes

- **Manifest shape:** v20 is “identical to v12” — additive fields only. Existing manifest readers keep working.
- **Static analysis:** Fusion produces validated logical plans for every query. Column-level lineage becomes a first-class artifact (vs Core, where it requires sqlglot post-processing). Tooling that surfaces lineage gets cheaper / more accurate on Fusion.
- **baseline vs strict static-analysis modes:** Fusion defaults to baseline (warnings, not errors). Tools that want to surface the warnings need to read Fusion’s diagnostic output, not just `run_results.json`.
- **Cross-engine artifacts don’t mix:** A Fusion-produced manifest in CI vs Core-produced manifest in dev breaks Recce-style diffs. Tool authors need explicit “engine match” gates.
- **on and unsafe static\_analysis values deprecated, removed May 2026.** Project authors must migrate to strict or baseline. Tools that read `static_analysis` config need updated parsing.

## 9.3 dbt MCP server stability for 2026

Net assessment: **stable enough to depend on for the local-CLI subset and basic Discovery tools; treat LSP/Fusion tools as moving targets; build a thin abstraction so you can swap to direct artifact reads if the MCP surface diverges from what you need.**

- v1.0 shipped late 2025; tool names + env vars stable since.
- Discovery API + Admin API surfaces depend on dbt Cloud (which is stable but proprietary; OSS users are excluded).
- Codegen + LSP tools experimental.
- The MCP protocol itself is still evolving; transport (stdio vs streamable-http) may shift.

## 9.4 Other risks

- **dbt Cloud lock-in for the most interesting tools** (Discovery API, Semantic Layer, lineage). OSS-first tooling has to either skip these or replicate them.
  - **Adapter sprawl.** Snowflake / BigQuery / Databricks / Redshift / Postgres / DuckDB all have different sample syntax, different `information_schema`, different auth. Each adapter you support is real surface.
  - **PII compliance and contract law.** Any tool that fetches warehouse data needs explicit data-handling story (no logging row contents, masking-aware fetches, audit trail of what the LLM saw).
  - **Subscription auth vs API key.** Like clauditor’s #86 / #95 work — operators using Pro/Max via Claude Code as a subprocess get one cost story; operators using API keys get a different one. CI shape determines which.
-

## Sources:

- [dbt manifest.json reference](#)
- [dbt run\\_results.json reference](#)
- [dbt catalog.json reference](#)
- [dbt sources.json reference](#)
- [dbt artifact schemas](#)
- [dbt Project Parsing reference](#)
- [dbt-labs/dbt-mcp on GitHub](#)
- [dbt-mcp environment variables](#)
- [About the dbt MCP server](#)
- [dbt Fusion: static analysis](#)
- [Upgrading to dbt Fusion v2.0](#)
- [Anthropic API pricing](#)
- [Anthropic prompt caching](#)
- [Claude Agent SDK subagents](#)
- [Claude Code custom subagents](#)
- [Recce: Data Review Agent for dbt PRs](#)
- [Recce open-source CI setup](#)
- [Elementary OSS](#)
- [Elementary GitHub Actions integration](#)
- [dbt-osmosis](#)
- [dbt-checkpoint](#)
- [dbt-coves](#)
- [Snowflake AI REDACT for PII](#)
- [Snowflake Dynamic Data Masking](#)
- [Securing data at scale with dbt + Snowflake](#)
- [actions/upload-artifact](#)