coalesce

A PRACTITIONER'S GUIDE

Agentic Data Migration

From the Coalesce practitioner panel: the engineers and delivery leads running agentic migrations in production.

WHO THIS IS FOR

Teams migrating pipeline estates built on SQL Server, Oracle, Postgres, MySQL, Informatica, WhereScape, or SAP HANA to Snowflake, Databricks, Microsoft Fabric, or Google BigQuery. If your system produces SQL, this guide is about your estate.

Transform | Catalog | Quality

ABOUT THIS GUIDE

Migration guides usually come in two flavors: vendor pitches in how-to form, and generic advice too abstract to act on. This guide offers a different approach: a working playbook. It describes how agentic migration actually runs: the strategy call, the toolchain, the six-stage loop and the reconciliation that closes it, the validation architecture, and the program discipline that carries an estate from the first file to the last cutover. FinCo, the running example, is a fictional composite drawn from real migration programs. It is not a real company, and it carries no performance claims anywhere in this guide.

CONTENTS

01 **Introduction** 4

A migration is four jobs, and this guide is about the one where migrations stall: pipelines. What agents change about that work, and why the evidence trail is what makes it governable.

02 **Scope** 7

Two axes predict difficulty better than any platform label. Locate your estate on the map, then choose between porting as-is, starting fresh, and modernizing in flight.

03 **Plan and Manage the Work** 15

Govern what agents can touch before they touch it. Then size the estate, order the sprints by usage, and run the program from a tracker that shows status, velocity, and the days still ahead of you.

04 **Assess and Map Existing Assets** 23

The loop starts here. State your standards, let the agent reconstruct what the source does, and decide the grain of the plan, one Node per CTE or a chain in one, before anything is built.

05 **Convert Scripts Into Governed Pipelines** 30

One file first. Build the Nodes, configure them, and land the work in a Git repository where nothing reaches the mainline except through a pull request. What the first file teaches goes into your team's skills, which carry the migration from one script to the rest.

06 **Build and Validate Transformations** 39

Round one of validation asks whether the code is well formed, not whether the numbers are right. Findings go back into the build loop. The two-tier rule sets what you change during conversion and what waits for parity.

07 **Reconcile Data** 44

Define the comparison, run the quick check, drill into what differs, and let the agent iterate until the numbers match. Coalesce Quality does the comparing where the data lives, so by default the agent sees results, not rows.

08 **Review, Promote, and Resolve Issues** 51

The reconciliation passed; now a person reads the results, the graph, and the Nodes one at a time, marks the pipeline reviewed, and promotes it. Where the humans are, and why the last tenth of the time matters most.

09 **Deploy and Operationalize** 57

Hand a coordinator the batch and let it run the loop file by file, then plan the dual-run window: a freeze-or-sync policy, a parallel run per release, and a rollback you have rehearsed.

10 **Measure Progress and Maintain the Migrated Environment** 64

What the loop learns from having run, and what your team owns once the last release goes live.

11 **Choosing Who Runs Your Migration** 68

Three delivery models, the economics behind them, and a diligence checklist for any vendor call.

CHAPTER 01

Introduction

Nobody plans a two-year migration. They plan a two-quarter migration, and then they discover what the plan didn't cover: the stored procedure nobody understood, the dialect quirk that breaks silently, the validation work that doubles every estimate. Migration projects have a reputation for taking too long, costing too much, and consuming the engineers you can least afford to lose; and the reputation is earned.

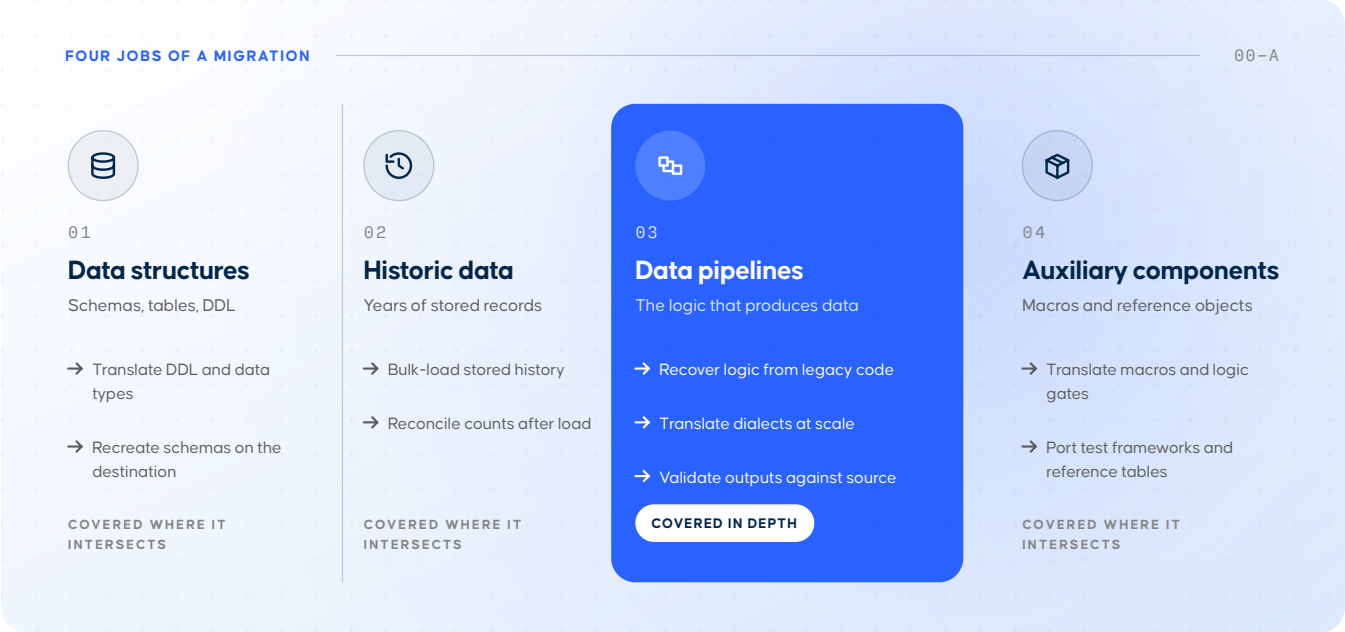
A typical data migration is four jobs:

Data structures. The schemas, tables, and objects that define where data lives. Usually the most mechanical job, and the one commercial tooling handles best.

Historic data. The years of records that have to arrive intact, reconciled, and provably equivalent. This job is less about movement than about proof.

Data pipelines. The transformation logic (stored procedures, ETL jobs, hand-written SQL) that turns raw data into the numbers the business runs on. This is where migrations go to die, because pipelines carry decades of undocumented decisions.

Auxiliary data components. Mapplets, custom logic gates, legacy SQL testing frameworks, macros, and reference objects and tables. These often have to be translated into the destination platform's language, too.



This guide focuses on data pipelines, because that is where agentic methods change the work most fundamentally. Structures, historic data, and auxiliary components show up where they intersect pipeline work. That means schema conversion during the build and parity validation against historic loads. Still, the pipeline estate is the main event.

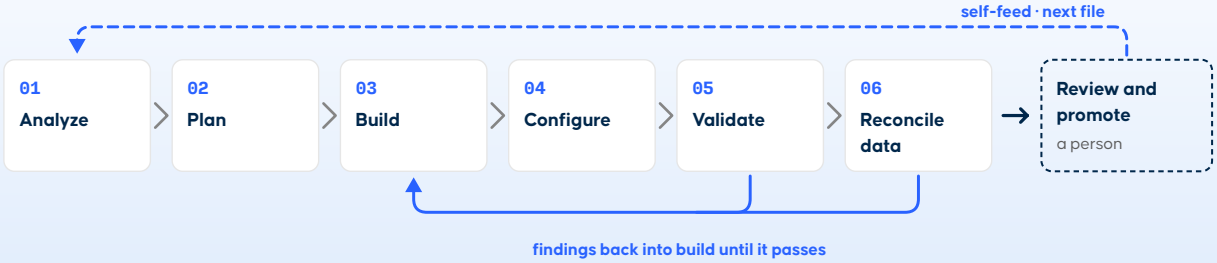
What agents change about the work

Why are migrations famously slow? Engineers read legacy code to reconstruct what it does and why. Then they translate SQL from one dialect to another, line by line. Finally, they check that the new outputs match the old ones, query by query. This work is tedious rather than creative: understanding code, translating between languages, and comparing outputs at scale. Yet those are the specific things large language models do unusually well.

This isn't to say migration is a push-button product, but it does make agentic migration a repeatable playbook. Agents execute a structured sequence of steps, humans review them, and evidence supports every stage. Here is the loop that Chapter 04 opens, Chapters 05 through 07 work through stage by stage, and Chapter 08 closes with the human review:

THE LOOP 03–A

Six stages, one loop-back, a person at the exit



Why regulated industries are the tell

You might expect regulation to be the obstacle here, but it is closer to the opposite, and the reason is the theme of this whole chapter. Regulated industries never had the option of faith: financial services, insurance, and healthcare teams have always had to prove lineage, defend definitions, and keep audit trails for pipelines humans built by hand. That work was the tax less regulated sectors got to skip. An agentic playbook that emits row counts, hash comparisons, an audit log of every query, and a reviewable decision trail is not a new compliance burden on top of that posture. It is a migration method whose evidence is a byproduct of the work, not a project bolted on top of it. Teams already operating at that bar are therefore best positioned to run agentic migration at scale. That is why this guide keeps returning to them as the worked example. If your team already holds that bar, start there. If it does not, adopt the evidence trail as the bar before the first sprint.

CHAPTER 02

Scope

Every migration conversation starts with the same question, usually phrased as cloud-to-cloud or on-prem-to-cloud. The question underneath it is where your logic lives and how far it has to travel. How complex will the migration be given your stack? Should you port as-is, modernize in flight, or start fresh?

The two axes that actually predict difficulty

Dialect distance is the translation problem. Your logic is already SQL; it just speaks Redshift, Teradata, or T-SQL. It still needs to speak your destination platform's language. The work is faithful translation plus adaptation to the destination's idioms: different date functions, different string handling, different performance patterns. Dialect distance varies, because some platform pairs are nearly fluent in each other while others barely share grammar. Still, the logic itself is visible, readable, and portable.

Logic accessibility is the extraction problem, and it is the harder axis. Here, the logic isn't sitting in readable SQL. It is locked inside proprietary tooling: Informatica mappings, WhereScape metadata, SSIS packages, stored procedures tangled with application code. Before any translation can begin, you must first recover the logic. That means reading it out of the proprietary format and reconstructing it as portable SQL. This work has historically justified large consulting engagements. After all, recovering intent from a decade-old ETL estate required expensive humans reading it line by line.

Put the two axes together, and you get a difficulty map for any migration. The lightest lift sits where both axes are short: SQL already running in your destination's dialect family, fully readable, needing structure and governance more than translation. By contrast, the heaviest sits where both are long: proprietary on-prem tooling where extraction, translation, and re-architecture stack on top of each other.

THE MIGRATION DIFFICULTY MATRIX — 01-A

How hard is your migration?



Why agents change the map

Agentic methods compress both axes, and that second one is the surprise.

Dialect translation at scale is the obvious win. An agent translates a thousand lines of Teradata SQL faster than any human. Unlike a one-shot conversion script, it can also iterate: translate, test, compare, then correct until the output holds.

The less obvious win is extraction. Reading legacy code to reconstruct intent is the skill that made migration consultants expensive. Yet language models are remarkably good at it. Hand an agent a 1,400-line stored procedure, and it can tell you what the procedure does. It can then explain why each section exists and how the logic decomposes into discrete steps. The extraction problem doesn't disappear, but it stops being the moat it used to be.

Industry as a diagnostic shortcut

Your vertical will not decide your migration, yet it will predict where you land on the map. Manufacturing and consumer goods estates, for example, skew SAP-heavy: deep extraction problems, transformation logic embedded in proprietary formats. Banking and insurance lean toward mainframe-era stored procedure estates. The logic there is accessible, yet it carries decades of undocumented business rules that make validation the long pole. Digital-native retail skews toward cloud SQL sprawl, with logic scattered across SaaS platforms and hand-written queries: mostly a translation and consolidation job.

Use the vertical to set expectations. Then diagnose the actual estate, because every company is its own exception.

What qualifies

The qualification rule is simpler than most vendors make it. If your system produces SQL and your destination is a modern cloud data platform, it qualifies. That covers stored procedures in SQL Server, Oracle, Postgres, and MySQL. It covers ETL and data platforms such as Informatica, WhereScape, and SAP HANA. It also covers hand-written SQL accumulated over years of ad-hoc analytics, plus compiled dbt models.

Destination platforms: Snowflake, Databricks, Microsoft Fabric, or Google BigQuery.

What does not qualify, or qualifies with caveats, matters just as much. Logic that never produced SQL (custom application code, black-box appliances) needs a different approach. So does logic where the source system is gone entirely. When the source is only out of reach, Chapter 03 covers a metadata-first plan you can build before you can read the code. FinCo is a fictional company that appears throughout this guide to illustrate the full migration journey. Here is its analysis:

THE FINCO FILE *a composite scenario: FinCo is fictional, drawn from real migration programs*

FinCo runs 340 stored procedures on SQL Server, built over 12 years. They feed the reporting layer of a mid-size financial services firm. The logic is fully accessible, since it is all T-SQL, readable and versioned. Yet the dialect distance to its Snowflake destination is real, and a third of the procedures have no documentation and no known owner.

On the matrix, FinCo lands upper-middle: no extraction problem, meaningful translation distance, and a validation burden inflated by all that undocumented logic. You will follow this migration through the rest of the guide.

One distinction first

Throughout this guide, the destination platform (Snowflake, Databricks, Fabric, BigQuery) and the data operating layer (Coalesce) are different things. Plenty of teams conflate them. In practice, "migrating to Coalesce" usually means "migrating to a cloud platform." Coalesce is then the layer where the pipelines live, run, and stay governed. The distinction matters because it explains the easiest migration on the whole map: SQL already running on your destination platform, moving into governed pipelines. That is barely a migration at all. It is an adoption, and Chapter 04 shows how short that path is.

Choose your strategy

Every migration team faces the same three options, whatever they call them. You have heard them as lift and shift, rip and replace, cleanup projects. Agentic data engineering has shifted the default, and most migration advice hasn't caught up.

The three strategies

Port as-is (the industry calls it lift and shift). Translate everything faithfully. Old equals new, validated and cut over. The appeal is certainty: no behavioral change, and no debate about what any pipeline should do. You just prove it does the same thing in the new place. The cost is equally clear: you re-platform your technical debt. Every workaround, every dead branch, every "temporary" fix from 2017 arrives in your new environment with its paperwork in order.

Start fresh (delete and rebuild). Sunset the legacy environment. Build net-new to modern standards, and let the old logic retire in place. It is the right call when the logic genuinely isn't worth recovering. That happens when the estate is small, or so poorly built that reconstruction from requirements beats translation. It is the wrong call more often than teams admit, because years of business rules live only in that code. And "we'll rebuild it from the requirements" assumes requirements that usually no longer exist.

Modernize in flight (the closest industry phrase is clean and shift, but hold that thought, because the sequencing matters). Migrate what is actually used. Refactor to the destination platform's idioms as you go. Drop the dead weight, and land a governed, modern environment rather than old code in a new warehouse.



Why the default changed

For 20 years, porting as-is won by default. Nobody loved it; the cost math simply favored it. Modernizing during a migration meant paying humans to do two jobs at once: translate the logic and improve it. Every refactoring decision added review time, review time that ran up against legacy system sunset deadlines, and every improvement added validation scope. The budget always ran out before the cleanup did. Starting fresh, meanwhile, stranded institutional logic nobody could afford to lose. So teams shipped their debt to the cloud and promised themselves they'd clean it up later, though later rarely came.

Agents collapse that math. The same agent performs extraction, translation, and refactoring in a single pass. A validation harness then tests value-level equivalence, regardless of how the code is shaped. As a result, modernizing stops being the premium option. Decomposing a monolithic procedure into clean, inspectable steps carries near-zero marginal cost when the agent is translating anyway. The same goes for normalizing names to your standards. What used to be the expensive strategy becomes the rational default. Standardizing on one platform compresses the work on its own. In a migration that used no agents, Security Benefit reports completing a key migration step in two weeks instead of the estimated two months, a 4x improvement.

> **"Instead of spending a week and a half building one version and being locked into it, it only takes 15 to 20 minutes to try each option."**
>
> **Brittany Young** · Solution Data Architect, Security Benefit

Still, porting as-is doesn't disappear. It remains right for regulated, parity-first cutovers where any behavioral change is a liability to be documented, not a bonus. Starting fresh remains right for estates that truly aren't worth recovering. Yet both are now the deliberate exceptions. Modernizing in flight is the default, and choosing one of the others should require a reason.

Before, during, or after: when the cleanup happens

Here is where the familiar phrase "clean and shift" leads teams astray. The instinct is understandable: get the estate in order, then move it. In practice nobody refactors the legacy code before a migration, and nobody should. You would be working in a dialect you are about to leave, inside a system you are about to retire, without the agent advantage the migration itself gives you. The refactoring question is real, but it belongs to the new code on the destination platform, and the sequence below says when.

The sequencing that works splits cleanup into three different jobs at three different times:

Before the move: triage, not cleanup. The one decision that genuinely belongs up front is what to leave behind. This is where usage and popularity metadata from Coalesce Catalog earn their keep: instead of debating what's safe to remove, measure it. Assets nobody has queried in a year get documented and dropped rather than translated. At FinCo, that decision alone removes a third of the estate before conversion starts.

During the move: structural modernization. Decomposition, naming standards, and platform idioms reshape code without changing what it produces. These changes are nearly free during translation, and value-level validation tests equivalence regardless

of shape. Chapter 06 turns this into a working rule.

After parity: semantic improvement. Logic fixes, business-rule changes, and anything that alters outputs belong here. These wait until the validation harness is green, then land as documented, evidenced changes. You get improvements you can prove instead of mysteries buried in a migration.

Triage before. Structure during. Semantics after. That ordering is the practical difference between modernize in flight and a cleanup project labeled as a migration.

Choosing honestly

In practice, two factors dominate the decision more than any framework. The first is documented standards. Teams with written conventions for naming, modeling patterns, and review rules modernize faster. The agent can then be taught the standard instead of inferring it. If your standards live in one architect's head, write them down before the migration, not during it. The second is review bandwidth. Modernization is bounded by how much change your engineers can actually review per sprint. Chapter 03 treats that constraint as the program's real throughput limit. Choose the strategy your review capacity can support, not the one the roadmap deck prefers.

THE FINCO FILE **the strategy call**

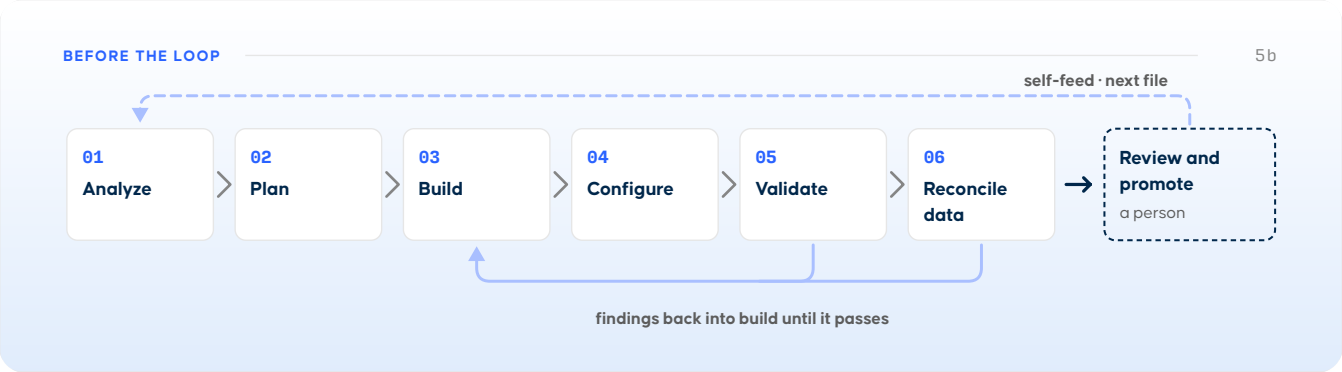
FinCo chooses modernize in flight, on two pieces of evidence. Catalog usage data shows 118 of its 340 procedures haven't run in over a year. Those are documented and dropped rather than migrated. FinCo's team also maintains written SQL standards, so the agents can be taught the house style instead of guessing.

Port as-is was considered and rejected: the estate's undocumented sections would have carried their mysteries into Snowflake intact. Start fresh was never realistic, because too much revenue logic lives only in the code. Neither exception had a case, so the default held.

CHAPTER 03

Plan and Manage the Work

The speed that makes agents useful also produces a failure mode every data leader has already seen. Generated SQL piles up, standards drift, and soon nobody can say with confidence what is running or why. The fix is not slowing the agents down. It is two disciplines, applied in a specific order: govern what agents can touch before they work, then validate what they produce after. This chapter does the first, and the planning around it. Chapter 02 located your estate and chose a strategy; here you set what agents can touch, assemble the toolchain they work with, size the estate, order the sprints, and put the program on the board your team already runs. Chapter 04 takes the first file of the first sprint into the loop.



What you need

- From Chapter 02: your strategy decision and the triage list of what migrates and what gets documented and dropped.
- Read access to the source system's code and to the usage metadata in Coalesce Catalog: query frequency, downstream dependents, and dashboard impact.
- A Coalesce Workspace backed by a Git repository, with Storage Mappings that point it at development schemas, kept separate from production.
- Your written standards: naming conventions, modeling patterns, and review rules.
- The board your team already uses, such as Jira or Linear, and the developers who will review each sprint.

Govern the agents before they work

Permissions default to read. Treat an agent's platform access like production database credentials, because that is what it is. An agent needs to see a great deal: code, metadata, run history, and lineage. Yet it needs to change very little without a human in the loop. Read access is therefore the default posture: the agent can inspect everything and touch nothing. Write access is a deliberate, scoped grant to specific people and workflows, not a blanket setting. On the client side, agent tooling can also gate individual write calls behind per-call approval. So even a granted capability executes one deliberate yes at a time.

None of this is a policy invention for this guide. Coalesce Copilot, the AI assistant built into the platform, ships this way: it creates or changes nothing until a person turns on Allow Edits, and the AI features stay off until an organization admin enables them. On the MCP and CLI path (the toolchain below), the agent acts with the permissions of the

token or warehouse role you give it, so the read-only default is yours to set: give it a read-only role, and keep write and run tools behind per-call approval in the client. Those defaults separate a useful agent from an expensive accident.

Boundaries are structural, not aspirational. Three of them do most of the work. First, agents build against development schemas, never production. Storage Mappings enforce the boundary: the same Storage Location points at a development schema in the Workspace and at production only in the production Environment, so pointing at prod is not a mistake an agent can quietly make. Second, agents work on branches and reach the mainline only through gated pull requests, with per-pipeline sign-off owned by humans; Chapter 05 shows the same rule from the workflow side. Third, agents never promote anything to production. Code moves from development to a UAT or QA Environment, is tested there, and a developer merges it to production with the reconciliation results in front of them. The switch from old pipelines to new is a person's call, too.

The agentic toolchain, in one page

The stack has four layers, each with a defined function.

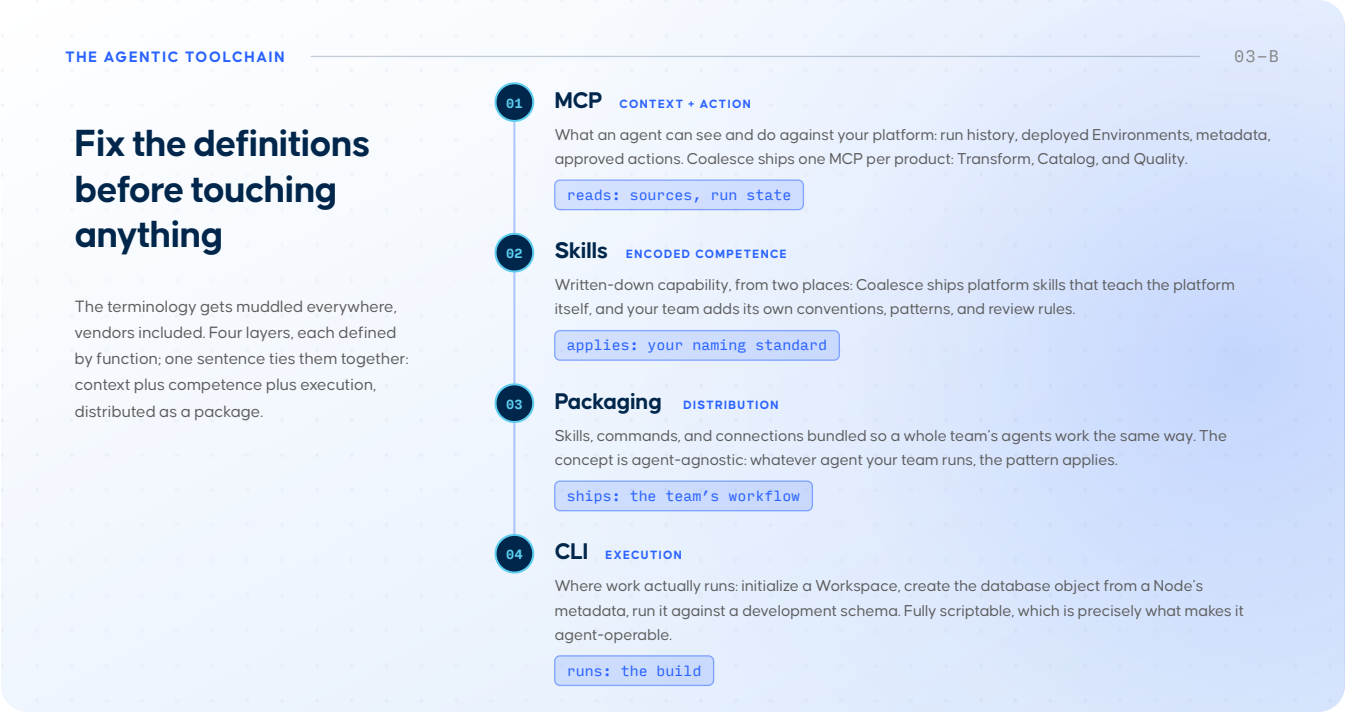
MCP (Model Context Protocol) is the context and action interface. It defines what an agent can see and do against your platform. For example, an agent can read run history, inspect deployed Environments, query metadata, and take approved actions. Coalesce ships one MCP per product (Transform, Catalog, and Quality), so an agent reads pipeline state, governance metadata, and quality status through the same standard. Without it, an agent reasons about your pipelines from the code alone, with no knowledge of what is actually running.

Skills are encoded competence. A skill is a written-down capability, and it comes from two places. Coalesce ships platform skills that teach an agent the platform itself: authoring Nodes, structuring pipelines, using the CLI correctly. Your team builds on top of that: naming conventions, modeling patterns, review rules. Together they are the difference between an agent that can run commands and an agent that is genuinely good at your stack. They also make your standards enforceable. Skills can evolve over time as standards and architecture change.

Packaging is how competence travels. Skills, commands, and connections are bundled so a whole team's agents work the same way. (Claude's plugin system is the worked example here. The concept is agent-agnostic: whatever agent your team runs, the pattern of distributing shared competence applies.)

The CLI is the execution surface. This is where work actually runs. Here, for instance, you initialize a Workspace, create the physical database object from the metadata definition in the Node, and run it against a development schema. The surface is fully scriptable, which is precisely what makes it agent-operable.

One sentence ties the stack together: context plus competence plus execution, distributed as a package. Every migration workflow in this chapter is some arrangement of those four layers.



Size the estate before you promise anything

Migration programs earned their scoping reputation honestly: they were nearly impossible to estimate, because nobody could see inside the estate without reading it. Agents remove that excuse. Parse the entire library up front, every procedure and every script, and the estate stops being a mystery and becomes a distribution. A 1,400-line procedure whose CTE structure will decompose into roughly 20 Nodes (at one Node per CTE) and a 100-line query that becomes one Node are not the same unit of work. Yet now you can see the difference before committing to anything.

The practical method is unglamorous: t-shirt sizes. Bucket every object by the patterns the parse reveals: line counts, CTE structure, join and window-function density, and the dynamic SQL and cursors that resist clean decomposition. Then estimate by bucket rather than by object. Use small, medium, large, and extra large, with a thin-slice

conversion of each bucket to calibrate the estimates against reality. Before the agent classifies the whole repository, have it classify a sample and read the scripts it put in each bucket. When you agree with its sizing, let it run on everything.

Here is the counterintuitive part: you are not sizing to predict conversion effort. Agents convert a large procedure and a small one with roughly the same indifference. You are sizing to predict *proof* effort, the review and validation each object will demand from the humans. That capacity is what the program actually runs out of. Complexity bucketing is review planning wearing an estimation costume. One cost does follow the bucket: model usage. Bigger, denser objects take more tokens to convert and iterate on, so the same sizes give a sponsor a basis for planning model spend.

And one encouragement for the planners, from delivery experience: the human review cost per object falls as the program runs. Early files calibrate the skills, sharpen the conventions, and build justified trust in the checks; later files inherit it all. Plan for a learning curve, not a flat rate.

Sprints, ordered by evidence

Sequence the program in sprints, and let usage data set the order rather than org charts or folder structure. Group the pipelines that go live together into releases: each release is worked in sprints and goes live through the dual-run window in Chapter 09. Coalesce Catalog already knows which assets the business actually touches: query frequency, downstream dependents, dashboard impact. High-usage, high-dependency assets go early, when attention and review capacity are at their peak. Decommissioning is the other hard driver: when a server or a license has an end date, whatever runs on it moves first. The long tail of rarely touched assets goes late. And the dead assets, the ones the usage data shows nobody has queried in a year, get documented and dropped rather than converted. This is Chapter 02's triage, now operationalized.

Two calendar rules sit on top of the usage ordering. First, cutover windows respect the business calendar. For example, retail does not cut over into peak season. Finance, likewise, does not cut over into quarter-close or audit windows, and subscription businesses avoid billing boundaries. Cutover plans that ignore the risk calendar get renegotiated by it. Second, sprints are sized to review bandwidth, the constraint Chapter 02 established. A sprint your engineers cannot review only looks faster; in practice, it is a queue with better marketing.

When you cannot see the source. Sometimes the source system is inaccessible. An Informatica license expires, a vendor relationship ends, or the one server nobody can touch stays off-limits. A metadata-first plan covers exactly this case. You generate it

from lineage, usage patterns, and asset metadata alone, priority-ranked by popularity and downstream impact. For each release, the plan spells out rationale, complexity, and affected dashboards, and it automatically tags each affected table with its place in the plan. You plan the program from what the estate's metadata proves about it. Then you recover the logic release by release as access allows.

The tracker: migration as a managed program

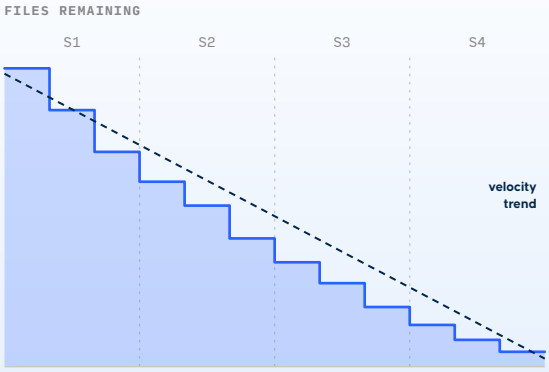
Run the program from the board your team already uses, Jira, Linear, or whatever tracks its tickets. Every file is a ticket carrying its t-shirt size, its sprint, its reviewer, and its sign-off. Plan in two-week sprints: pull tickets by size against the developers you have and what they can review in two weeks, then measure against that. Velocity and time remaining fall out of the burndown. Migrations fail in the dark, in the status meeting where nobody quite knows what percent done means. A board whose tickets close as files pass review replaces the guess with a burndown anyone can read. Agentic migration has a project-management layer, and it is the one your team already runs.

THE MIGRATION BURN-DOWN 05–B

A sprint burndown on the board your team already runs

FILE	SPRINT	REVIEWER	STATUS	SIGN-OFF
SP_INVENTORY_ROLLUP	S1		SIGNED OFF	✓
DIM_CUSTOMER_LOAD	S1		SIGNED OFF	✓
RPT_MARGIN_WEEKLY	S2		IN REVIEW	-
STG_ORDERS_DAILY	S2		CONVERTING	-
FCT_SHIPMENTS	S3	-	QUEUED	-
SP_GL_SUMMARY	S3	-	QUEUED	-

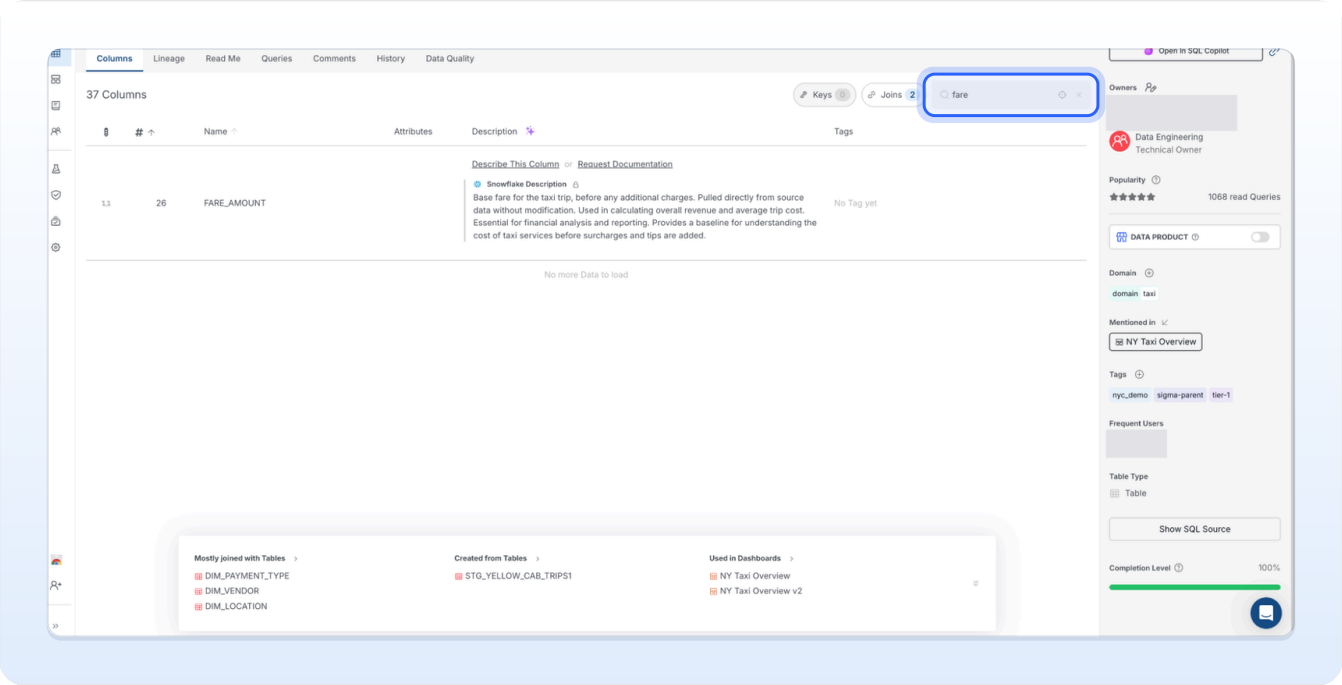
...every file in the release, one ticket each



Velocity and time remaining fall out of the sprint burndown: tickets close as files pass review, and anyone can read it.

In Coalesce

Three platform surfaces carry this chapter. Storage Mappings keep development and production apart, so an agent working in a development Workspace cannot write to production by mistake. The Git repository behind the Workspace gives every agent change a branch, and its pull request waits in your Git provider. And Coalesce Catalog holds the usage metadata that orders your sprints: query frequency, downstream dependents, and dashboard impact. Copilot edits nothing and the AI features stay off until a person turns them on; on the MCP and CLI path, the permissions you grant set the limit.



USAGE DATA SETS THE SPRINT ORDER

What good looks like

You are ready for the first file when the program exists on your board, not in someone's head.

- Agents hold read access by default. Every write grant is named, scoped, and owned by a person.
- Agents build only against development schemas, through the Workspace Storage Mappings, and the mainline accepts changes only through a pull request a person reviews.

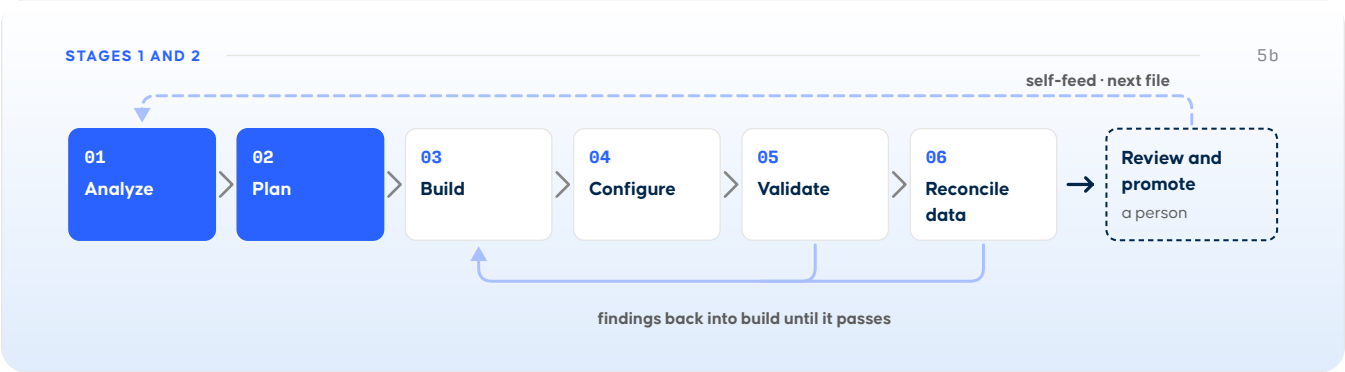
- The toolchain is in place: MCP access to the platform, Coalesce platform skills plus your team's written standards, and the CLI, packaged so every agent works the same way.
- Every object in scope carries a t-shirt size, calibrated by a thin-slice conversion per bucket. Dead assets are documented and dropped, not queued.
- Sprints are ordered by usage and decommissioning dates and sized to what your reviewers can sign off, and every release's cutover falls clear of the business calendar's risk windows.
- Every file is a ticket on your board with its size, sprint, and reviewer, and the first sprint is planned.

The burndown above is what this chapter's output looks like once the first sprint starts.

CHAPTER 04

Assess and Map Existing Assets

With strategy decided and the estate triaged, the work itself begins. This chapter opens the loop: the six stages that carry a file from source SQL to a reconciled pipeline. Every stage is something a senior data engineer would do by hand. The agent simply does that work tirelessly and leaves evidence as it goes. Here you prepare the SQL and run the first two stages, analyze and plan, on the first file of the first sprint. Chapter 05 builds and configures the plan you approve.



What you need

- From Chapter 03: the first sprint on your board, with each file sized and a reviewer named.
- The source SQL for those files, with the tables they read already present in your Workspace.
- An agent with read access to your platform through MCP, plus Coalesce platform skills and your team's own.
- Your standard for column references and your naming conventions, written where the agent can read them.
- Time on the first files to read each plan closely. This is where your standards get set.

Step zero: prepare the SQL

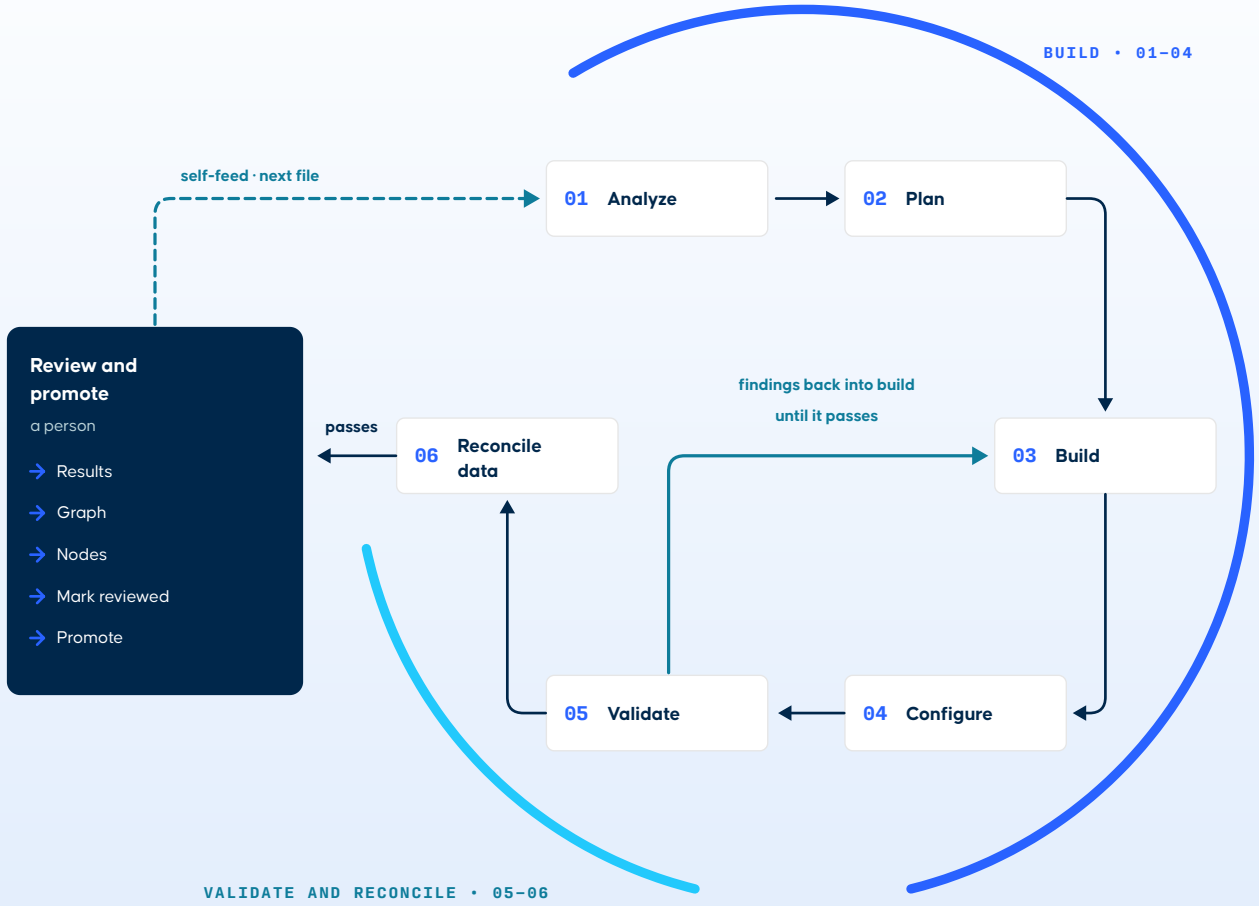
The prep is short. Decide your standard for column references up front: fully qualified, no bare table aliases. On the interactive Coalesce Copilot path this is prep you do by hand, because an alias like t.col arrives as an expression stranded in a transform field instead of a column mapped to its source. On the agentic path the agent qualifies references as it builds, so the standard is something you tell it, and the check moves to the build stage, where every column should map to a source. Compile dbt models to plain SQL first. Split very large procedures into logical chunks. And make sure source tables exist in the Workspace before conversion starts, because conversion maps to existing sources rather than inventing them.

The loop

The playbook runs the same loop a senior engineer would, but makes it explicit. Six stages carry a file from source SQL to a reconciled pipeline: analyze, plan, build, configure, validate, and reconcile data. One loop-back closes it. When validation finds a problem, or the reconciliation does not match, the findings go back into build and the agent iterates until the numbers agree. When they agree, a person reviews the pipeline and promotes it. That human step is the exit from the loop, and it gets its own chapter. One design choice shapes all of it: the model makes its judgment calls in a few places, and the tests that decide whether a file is done are deterministic, with a yes-or-no answer.

THE PLAYBOOK LOOP 03–A

Six stages per file, one loop-back, a person at the exit



The ring shows the order you review in; the agent iterates inside it until the reconciliation passes.

> REVENUE_FACT is FinCo's flagship stored procedure: 1,384 lines long, with no documentation and no living owner. The artifact cards beside each stage follow it through the loop. Every value in the cards is illustrative, because FinCo is a fictional composite.

01 Analyze

The agent parses the procedure before it reasons about it. It reconstructs purpose, grain, dependencies, and side effects. It also catalogs the patterns in the code: which CTEs feed which, where temporary objects appear, which statements do more than produce the final output. The parse reports how much of the file it covered. At this stage, the agent proposes no fixes and changes no logic.

Check the analysis for missing branches, hidden assumptions, temporary objects, and operations that affect more than the final output. Then check coverage. A procedure the parser only partly reads is a procedure the plan will only partly capture.

> **Where it fails**: dynamic SQL and cursor loops resist parsing. They surface here as low coverage or flagged regions, and they route to deliberate handling, sometimes a redesign, rather than automatic decomposition.

A-1 **Analyze** · structural summary — COMPLETE

PURPOSE	Monthly revenue allocation to channel and region, loaded into the reporting fact.
STRUCTURE	18 CTEs feeding one final insert. Sampled: `monthly_bookings` (base grain), `fx_normalized` (currency), `channel_allocated_revenue` (allocation rules).
CONNECTIVE	Two temp-table setups; one post-insert logging step.
RISK NOTES	One branch appears unreachable from current parameters; two column references rely on implicit resolution.

```
CREATE PROCEDURE dbo.REVENUE_FACT
WITH monthly_bookings AS (…)
     fx_normalized AS (…)
     channel_allocated_revenue AS (…)
     … 15 more CTEs
/* temp tables ×2 · logging ×1 */
INSERT INTO reporting.REVENUE_FACT
!! unreachable branch · implicit refs ×2
```

02 Plan

This is where the model's judgment does its main work. The plan translates the source structure into a proposed destination design before building anything. The plan decides the grain. One Node per CTE is the default this guide follows, because each step then carries its own lineage and its own review surface. It is a choice, not a rule. Where a chain of CTEs reads as one unit and splitting it adds nothing to review, a SQL Node Type keeps the chain in one Node; the docs list migrating SQL into Coalesce among its uses. SQL Node Types run on Snowflake and BigQuery today and YAML

Node Types on every platform Coalesce supports, so the grain you choose is also a platform check. The Nodes are ordered so that every dependency is available before the Node that needs it. The plan keeps the source's original dependency edges alongside the rebuilt ones, and it carries a one-line summary of what each CTE does, written by the model. Those summaries make the plan readable for someone who has never seen the procedure.

Review how each CTE or logical stage maps to a Node, and where procedural elements such as temporary-table setup and logging are routed. Compare the rebuilt dependency graph against the source's: the edge sets should match. Keep any uncertain choice visible for review rather than silently resolved. The plan is the cheapest thing in the loop to reject. Read it closely on the first files; this is where you set the standards the agent carries through the rest of the estate. Once the plans hold, let them flow into build. A plan can still come out slightly different each time the agent runs: the Node Types apply the major patterns, such as Type 2 change tracking and the merge method, the same way every time, and the human review in Chapter 08 catches what varies.

> **Where it fails**: the model merges two CTEs that should stay separate, or orders a Node before the one it reads from. Both are visible in the plan and cost nothing to fix there.

A–2 **Plan** · CTE-to-Node mapping — PLAN REVIEWED

SOURCE CTE		DESTINATION NODE
monthly_bookings	→	STG_MONTHLY_BOOKINGS
fx_normalized	→	STG_FX_NORMALIZED
channel_allocated_revenue	→	STG_CHANNEL_ALLOCATED
… 15 further CTEs	→	… 15 further Stage Nodes
final insert	→	FCT_REVENUE

REMAINDER — Temp-table setups → Pre-SQL on their consuming Nodes; logging step → Post-SQL on FCT_REVENUE.

FLAGGED — The unreachable branch is kept visible in the plan for review, not resolved silently.

What decomposition doesn't capture

Not every line of a legacy procedure maps to a Node. Session variables, temp-table setup, and post-insert logging (the connective tissue) land in the Pre-SQL and Post-SQL boxes on the appropriate Nodes. Dynamic SQL and cursor loops need deliberate handling, sometimes a redesign. Expect a remainder bucket on every complex

procedure, a small share of the whole, and handle its contents case by case. Finally, review that bucket as carefully as the Nodes. Guides that promise 100% capture are describing demos, not estates.

In Coalesce

Two things have to be in place on the platform before the first plan. The tables the procedure reads must already exist as sources in your Workspace, because conversion maps to existing sources rather than creating them. And the agent needs read access through MCP, so it can see what is actually running and not just the code.

What good looks like

The first file is ready to build when you would approve its plan without the agent in the room.

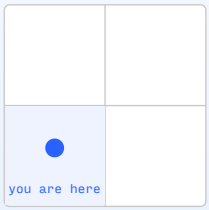
- Your column-reference standard is written where the agent can read it (on the interactive path, the source SQL already follows it), and the tables the file reads exist in the Workspace.
- The analysis reconstructs purpose, grain, dependencies, and side effects, and it states how much of the file the parser covered.
- Dynamic SQL, cursor loops, and low-coverage regions are flagged for deliberate handling, not left for the build to discover.
- The plan maps every CTE or logical step to a Node, or keeps a chain together in one SQL Node on purpose, and its dependency graph has the same edges as the source.
- Temp-table setup, session variables, and logging are routed to Pre-SQL and Post-SQL on named Nodes. The remainder bucket is listed and reviewed.
- Uncertain choices are still visible in the plan, and you have approved it.

The Analyze and Plan cards above show both artifacts for REVENUE_FACT: the structural summary, and the 19-Node plan with its remainder routing.

PANEL · CHAPTER 04

Already on Snowflake?

The lightest square on the difficulty matrix, shown honestly. When your SQL already runs on the destination platform, extraction is zero and translation is nearly zero.



The same holds if you are already on Databricks, Fabric, or BigQuery.

WHAT REMAINS OF THE PLAYBOOK

decomposition	configuration	validation
STRUCTURE	GOVERNANCE	EVIDENCE

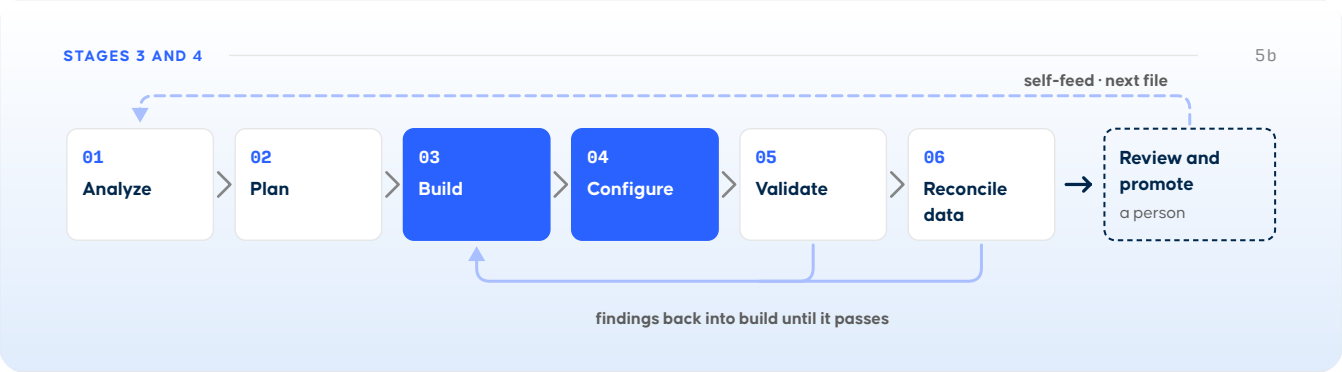
This is barely a migration; it is an adoption, with pipelines built in days, not weeks.

The playbook still pays for itself in the evidence trail alone: even logic that "already works" benefits from becoming 20 inspectable steps with proven parity. Same loop, three steps shorter.

CHAPTER 05

Convert Scripts Into Governed Pipelines

Chapter 04 ended with a plan you approved. This chapter turns it into a pipeline. The agent builds each Node from its predecessors, so lineage survives the conversion, and then configures what the SQL alone cannot say: keys, change tracking, storage, and materialization. The work lands on a branch in your Workspace's Git repository, and nothing reaches the mainline except through a pull request a person reviews. What went in as a script comes out as a set of Nodes you can see, trace, and review. Chapter 06 checks that it is well formed.



What you need

- From Chapter 04: an approved plan for the file, with each CTE or logical step mapped to a Node and the remainder routed.
- Your Node Type definitions. Configuration fills from them rather than inventing settings per Node.
- A working branch, with branch protection on the mainline so the pull request is the only way in.
- Coalesce Copilot for the interactive path, or the coa CLI for the agentic path.
- Development schemas for the agent to build against, set by the Workspace Storage Mappings.

Start with one file

Do not start with the estate. Start with one file, ideally the one most representative of your repository. Pick it by shape, an insert or a merge, so the first file covers most of what the estate does.

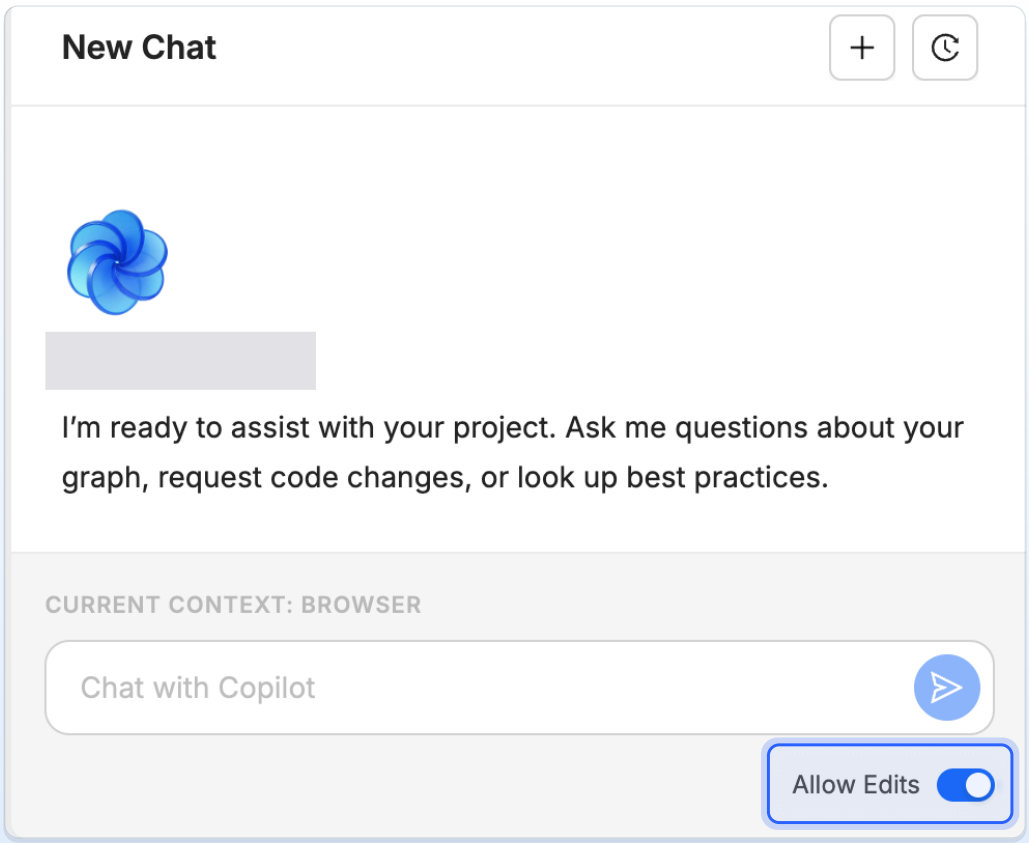
There are two ways to run that first conversion, and they map to different points on the maturity ladder. In the interactive mode, you paste a stored procedure into Coalesce Copilot inside the platform interface, watch it decompose into Nodes, and then interrogate the result. This is the on-ramp: one file, immediate feedback, no setup beyond your Workspace. The Coalesce docs walk through it end to end, including copy-paste instructions. The programmatic mode is the playbook this chapter describes: agents working through the CLI against files, at estate scale. Either way, the plan sets the grain. The documented Copilot walkthrough creates a Node for each CTE, and this guide follows the same default, but a SQL Node can keep a chain of CTEs together where splitting adds nothing to review (Chapter 04).

Your first instruction can be a sentence, not a framework. The example in the docs reads:

PROMPT

```
Reverse engineer this SQL into Coalesce Nodes. Create
a Stage Node for each CTE. Use my existing source
Nodes INVENTORY_MOVEMENTS and PRODUCTS. Map to my
WORK Storage Location.
```

Then paste the procedure. Specify what you care about up front: Node Type preferences, your naming convention, and the target location. Every convention you state is one the agent doesn't have to guess.



THE INTERACTIVE ON-RAMP, ONE FILE IN

The first file teaches you things no plan can. In particular, it shows how the agent reasons about your code, where your conventions are ambiguous, and what your review takes. Question its decisions, and ask why it split a step where it did. Converting the file matters less than calibrating a working relationship with the agent.

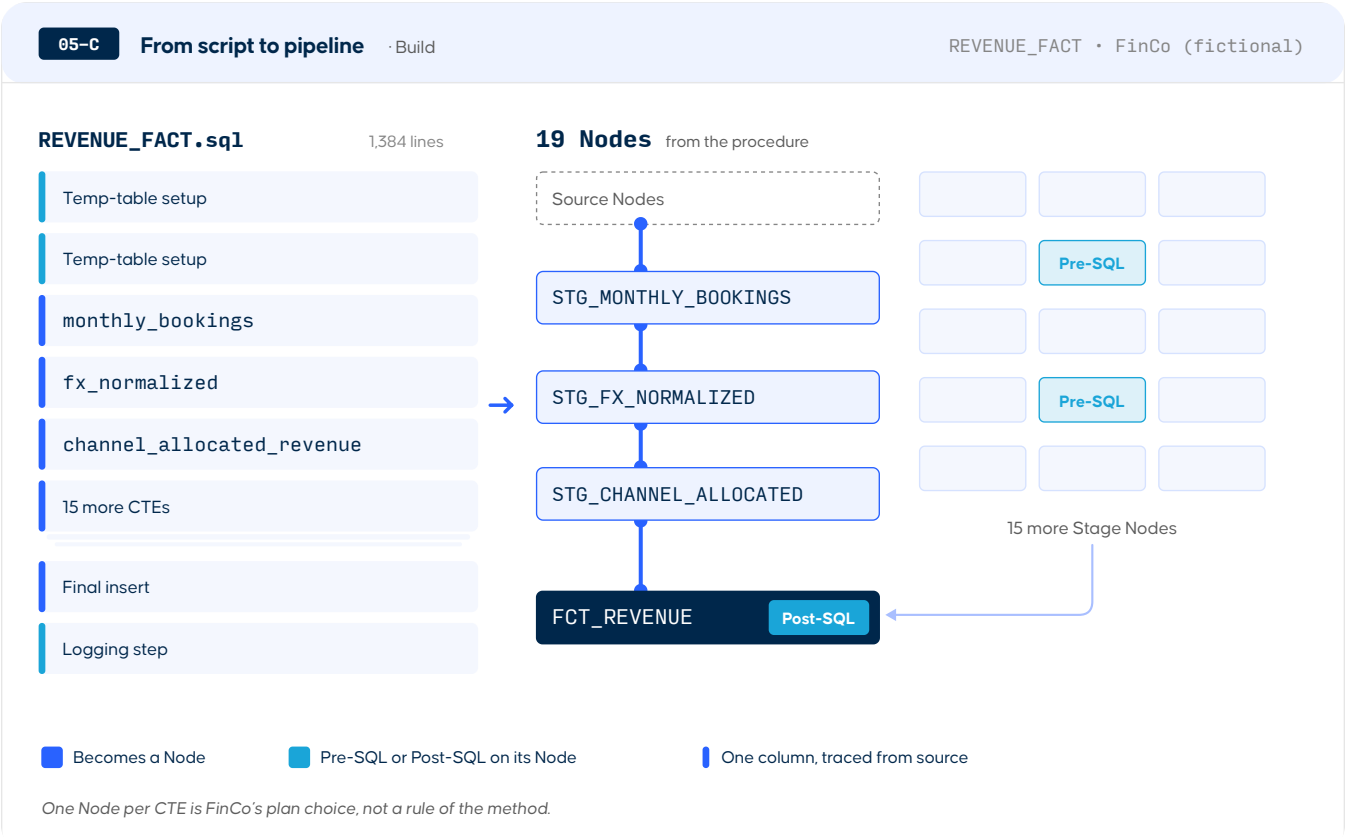
03

Build

Build follows the approved plan. The agent creates each planned Node from its predecessors and wires column-level lineage as it goes. Because Nodes are created from predecessors rather than typed in, lineage survives the conversion. Structural changes belong here: your naming standard and the dialect rewrites the destination requires, provided they preserve what the source produces. When a build step fails, the failure names the Node and the column, so the next iteration can act on it.

Compare the build with the approved plan and confirm that no business-rule change has entered the conversion. Then open the columns. Every column should be enumerated and mapped to a source; a select-star that survived the build is a lineage gap and a review gap.

> **Where it fails**: a column that arrived as an expression in a transform field instead of a mapping to its source, usually because a bare table alias was carried through. The fix is mechanical: remove the transform and map the column to its source. This is the check step zero pointed you at.



A–3 **Build** · before-and-after diff

BUILT • NO LOGIC CHANGES

```
- where load_ts <= GETDATE()
+ WHERE LOAD_TS <= CURRENT_TIMESTAMP()
- select rev_alloc, chan_cd
+ SELECT REV_ALLOC, CHANNEL_CODE
```

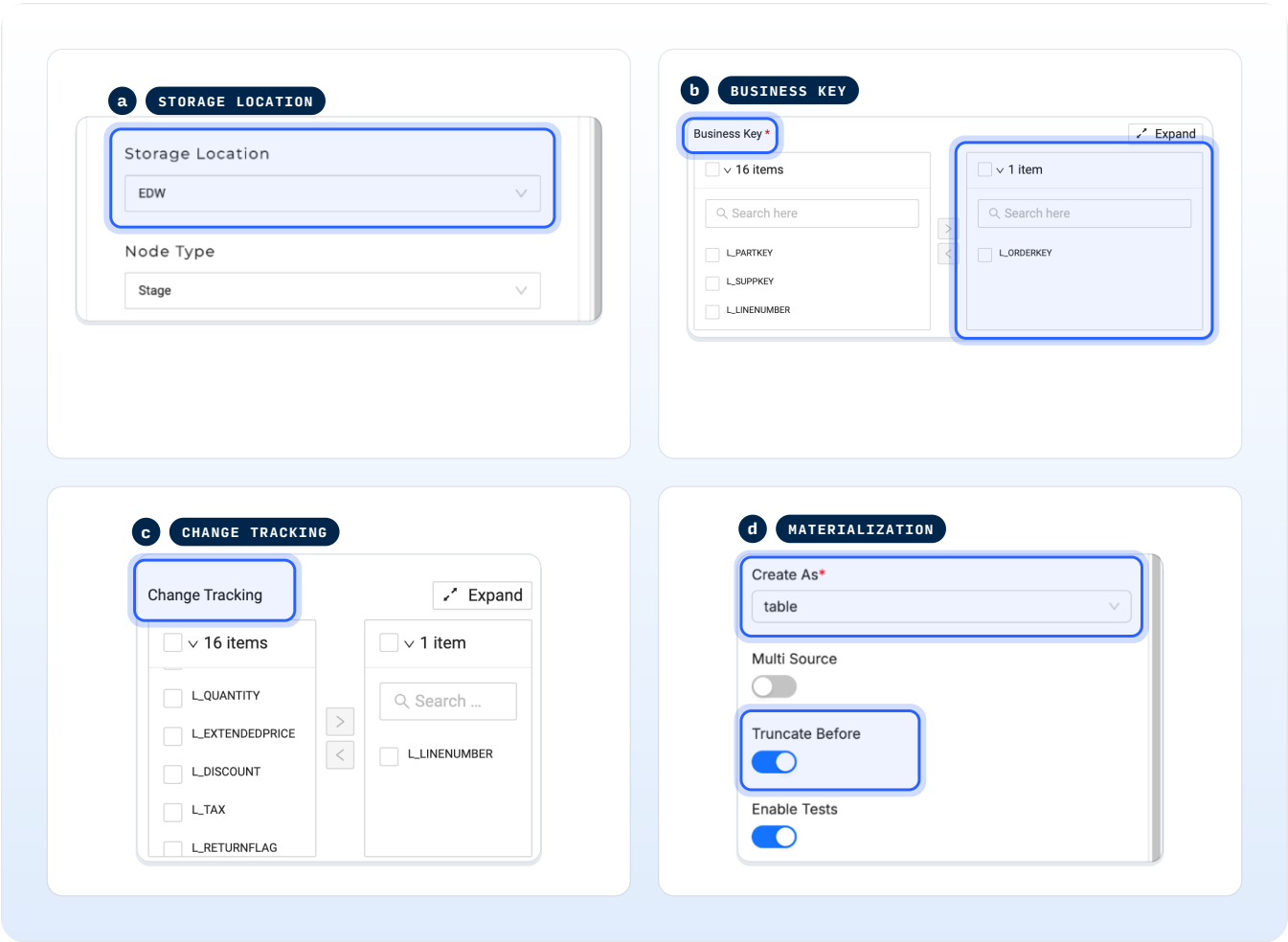
Naming standard plus the platform idiom, across 19 authored Nodes. Logic unchanged; the suspected dead branch is carried as-is, on purpose.

04 Configure

Configuration supplies the operational settings the transformed SQL alone cannot express, and it comes from your own Node Type definitions rather than being invented per Node. For a Fact Node: the business key it merges on. For a Dimension or Persistent Stage Node: the business key and the change-tracking columns if it keeps history. For every Node: the Storage Location, whether it materializes as a table or view, and whether the Node truncates before it loads. The agent rule is that no field is left blank; a blank is a decision nobody made.

Inspect business keys, change-tracking behavior, storage targets, Node Types, and inherited defaults. Two defaults deserve a second look on every migrated procedure. A Stage Node truncates and reloads on every run, so logic that accumulated history in the legacy system needs a Persistent Stage or a keyed Node instead. A Dimension Node requires a business key at the right grain, so a key copied from the source at the wrong grain will pass configuration and fail parity.

> **Where it fails**: an inherited default that is correct for your team's standard and wrong for this procedure. Configuration cannot see that. Parity can.



NO FIELD LEFT BLANK

A-4 **Configure** · configuration card — EXPLICIT

BUSINESS KEY
BOOKING_ID + FISCAL_MONTH

LOAD PATTERN
MERGE on the business key

CHANGE TRACKING
None: a fact keeps no history

MATERIALIZATION
Table, no truncate before load

STORAGE TARGET
WORK Storage Location, per FinCo's environment standards

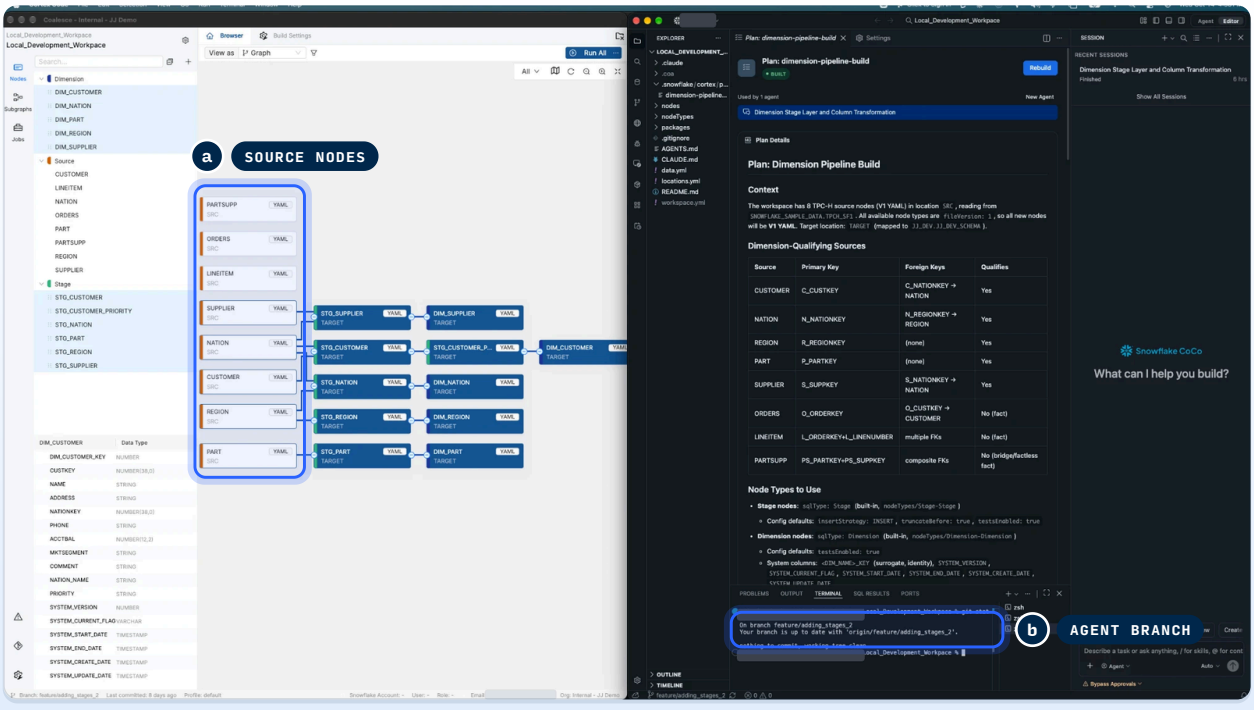
DEFAULTS
Inherited from the team's Node Types

Where the work lands

Each Workspace is backed by a Git repository, and that choice is the substrate of the playbook, not an implementation detail. Every agent action becomes a file change, and every change is committed, reviewed, and merged like any other. That discipline makes the whole model governable.

Agents work on branches, and nothing an agent writes reaches the mainline except through a pull request.

Branch protection makes that boundary structural. An agent cannot push its work straight into the integration branch. The pull request, with its CI gates, is the only door in, and the review of that pull request is the sign-off: a person reads the diff and merges, and nothing merges on a green check alone. The diff shows old code against new. It does not say who wrote which line, and it does not need to; the pull request is the unit of accountability. That pairing translates a century of separation-of-duties thinking into agentic delivery. The entity that authors a change is never the entity that approves the deliverable. Reviewability is the surface the process runs on. After all, these are the same commits, diffs, and reviews your team already applies to everything else it ships.



THE PULL REQUEST IS THE ONLY DOOR IN, AND ITS REVIEW IS THE SIGN-OFF

SIDEBAR

Running the playbook on your machine

The programmatic mode runs on a local toolchain: your pipeline as files in a Git repository, built and run with the coa CLI, reviewed in Coalesce Desktop, with Coalesce platform skills teaching your agent how to author Nodes, structure pipelines, and use coa correctly. The agent works in the terminal; you watch Nodes appear in Desktop before anything deploys.

`pipeline as files in Git` `coa CLI` `Coalesce Desktop` `Coalesce platform skills`

Local development runs on Snowflake, Databricks, and BigQuery; the MCP experience is consistent everywhere.

→ **Local Development Setup Guide**

In Coalesce

The conversion shows up in four places you can open. On the interactive path, the first file goes in through the Coalesce Copilot panel. The graph shows the Nodes as they are built, with lineage from each one back to its predecessors. Each Node's Config tab shows the settings that came from your Node Types. And the Git panel shows the branch the agent works on; its pull request waits for review in your Git provider. The screenshots in this chapter follow those four, in that order.

What good looks like

The build is done when every Node traces back to its source and no setting is left for someone to guess.

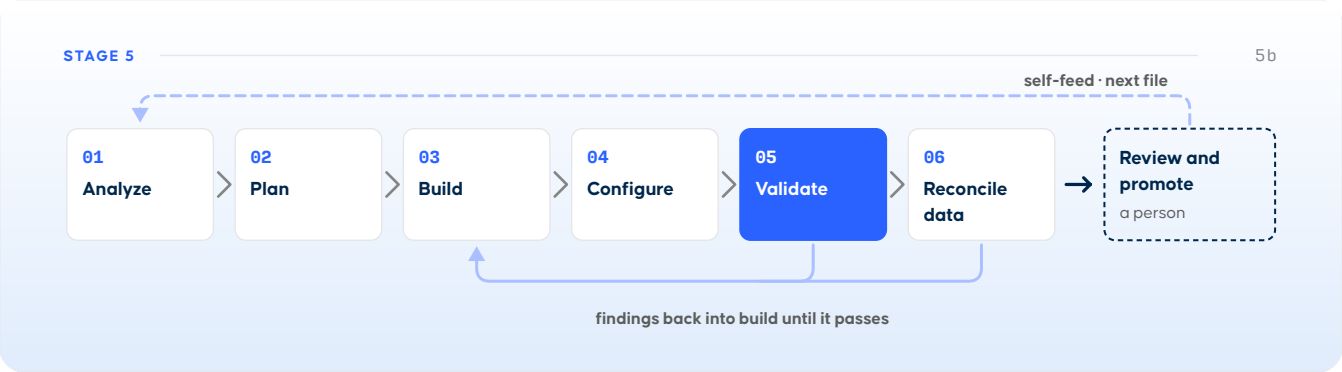
- Every planned Node exists, created from its predecessors, and the graph matches the approved plan.
- Every column is enumerated and mapped to a source. No select-star survived, and no column sits as an expression in a transform field.
- Only structural changes went in: decomposition, naming, and dialect rewrites. No business-rule change entered the conversion.
- Every Node that needs a business key has one at the right grain, change tracking sits only where a Node keeps history, and every Node has its Storage Location, materialization, and truncate behavior, filled from your Node Type definitions, with no field blank.
- Any Stage Node that replaced history-keeping logic is now a Persistent Stage or a keyed Node.
- The work sits on a branch in a pull request, and nothing has reached the mainline.

The Build and Configure cards above show REVENUE_FACT at this point: 19 Nodes authored, the structural changes listed, no logic changes, and FCT_REVENUE's configuration. Diagram 05-C shows the same file before and after.

CHAPTER 06

Build and Validate Transformations

Chapter 05 built and configured the pipeline on its branch. Before you compare any data, this chapter asks whether the code is well formed: sound in structure, references, and configuration. The platform's own validate and the agent's check suite answer that together, and every finding goes back into the build loop until the checks converge. The chapter closes on the rule that decides what you may change during conversion and what waits for parity. Chapter 07 then runs the code and tests whether the numbers match.



What you need

- From Chapter 05: the built and configured pipeline on its branch.
- Your check suite, versioned in the repository, covering columns, joins, references, and metadata.
- The coa CLI, which runs the platform's validate against the Workspace files without touching the warehouse.
- A place for the bugs you find but do not fix yet: one ticket per bug on your board, fixed after the switchover.

Round one of validation

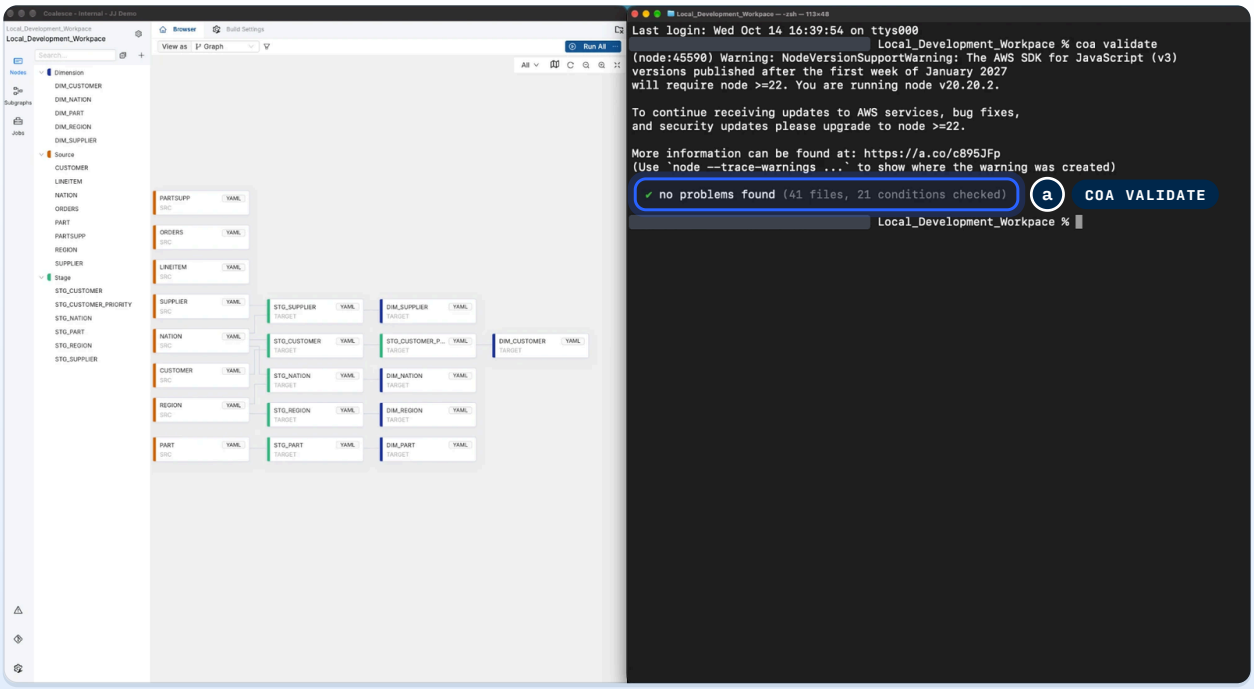
05

Validate

Round one of validation asks whether the code is well formed, not whether the numbers are right. Two surfaces answer it. The platform's own validate checks the Workspace files for structure, configuration, and column references, without touching the warehouse. The agent's own check suite runs alongside it, across four families (columns, joins, references, and metadata), and grades each finding by severity. The families are the checks the delivery playbook runs; a self-directed team writes its own suite from them and versions it beside the pipelines.

Read compile status and findings separately. A pipeline can compile cleanly and still carry three findings. Route every finding, including warnings, back into the build loop until the checks converge; a finding nobody routed is a defect nobody owns. On SQL Nodes, read column-reference findings with coa validate --verbose before routing them; that check can report false positives there.

> **Where it fails**: the finding that hides. An unqualified column reference resolves to the wrong table and compiles cleanly. Structural checks catch it only if the reference family is in the suite, which is why the suite is a versioned artifact and not a habit.



COMPILE STATUS AND FINDINGS READ SEPARATELY

A–5 **Validate** · checklist with findings — **3 FINDINGS**

COA VALIDATE • WORKSPACE FILES

- ✓ Compiles: all 19 Nodes
- ✓ Structure and configuration: clean

FINDINGS • READ SEPARATELY

- ! Unqualified column reference, STG_FX_NORMALIZED
- ! Unqualified column reference, STG_CHANNEL_ALLOCATED
- ! Implicit date cast the destination resolves differently

Route: all three findings back into build until the checks converge. Trial execution waits for reconcile.

The two-tier hygiene rule

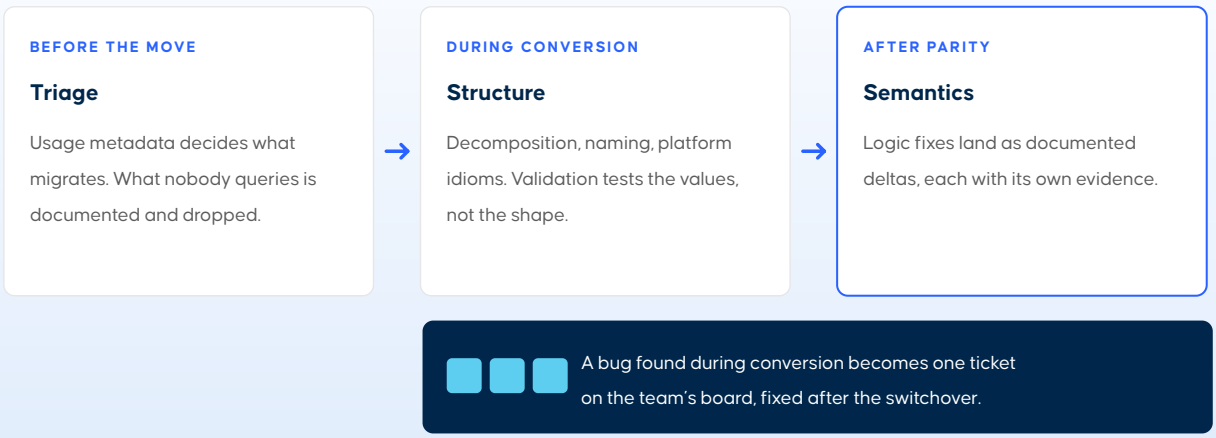
Chapter 02 promised a working rule for modernizing in flight. Here it is.

Structural changes happen during conversion. Decomposition, naming normalization, and platform idioms all reshape code without changing what it produces. These are nearly free while the agent is translating anyway, and they are safe for one reason: value-level validation tests equivalence regardless of shape. The data reconciliation stage does not care whether the logic lives in one monolith or 20 Nodes. It cares whether the numbers match.

Semantic changes wait for parity. Logic fixes, business-rule corrections, and dead-branch removal inside live procedures all change what the code outputs. Convert like for like, open a ticket for each bug you find, and fix it after the switchover. Those tickets are your semantic queue. The business has usually lived with the bug for months or years, and a clean comparison of two data sets is worth more than a fix made mid-flight. These changes land only after the reconciliation is green, each one as a documented delta with its own evidence. The sequence is the discipline: convert with structure, prove parity, then improve against a green reconciliation. Every improvement you make from there is provable.

MODERNIZE IN FLIGHT 06-B

Triage before. Structure during. Semantics after.



In Coalesce

Validation starts at the terminal, where coa validate scans the Workspace files for structural and configuration problems without touching the warehouse. Your own check suite runs beside it and lives in the repository with everything else. Read the platform's result for structure and the suite's findings by severity; the screenshot

beside the Validate stage shows both. The in-app Validate Create and Validate Run actions wrap the SQL in an EXPLAIN against the warehouse, so they belong with the first question of Chapter 07: does the code run?

What good looks like

The pipeline is ready for reconciliation when the checks have nothing left to say.

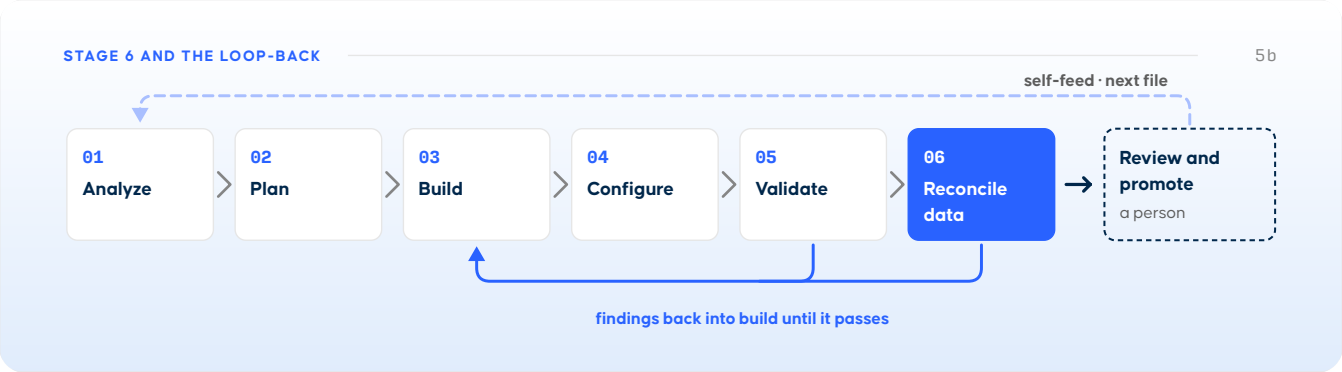
- The platform's validate passes against the Workspace files: structure, configuration, and column references.
- Your versioned check suite has run all four families (columns, joins, references, and metadata). Every finding, including warnings, went back into build and was checked again.
- Compile status and findings are read separately. A pipeline that compiles with open findings is not done.
- Every bug you noticed and did not fix is a ticket on your board, marked for after the switchover.
- The only changes in the pipeline are structural.

The Validate card above shows REVENUE_FACT's first round: every Node compiles, and three findings go back into build.

CHAPTER 07

Reconcile Data

The code runs. This chapter asks the question the business cares about: do the legacy and destination data sets agree? Coalesce Quality compares them where the data lives. When they disagree, the findings go back into build and the agent iterates until the numbers match. That loop-back is the last thing the agent does on its own. Chapter 08 hands the reconciled pipeline to a person to review and promote.



What you need

- From Chapter 06: a pipeline that passes validation, with every finding routed.
- Coalesce Quality, with its CLI and credentials installed beside the coding agent.
- Read access to the legacy output and, where the data owner or your security team allows it, to production source data. On Snowflake the common shape is a read-only secure share.
- The key columns that identify a row in each data set, and your documented rule set for the volatile columns to exclude.
- A place to keep the iteration log, the saved results, and the audit log.

How the comparison works

06

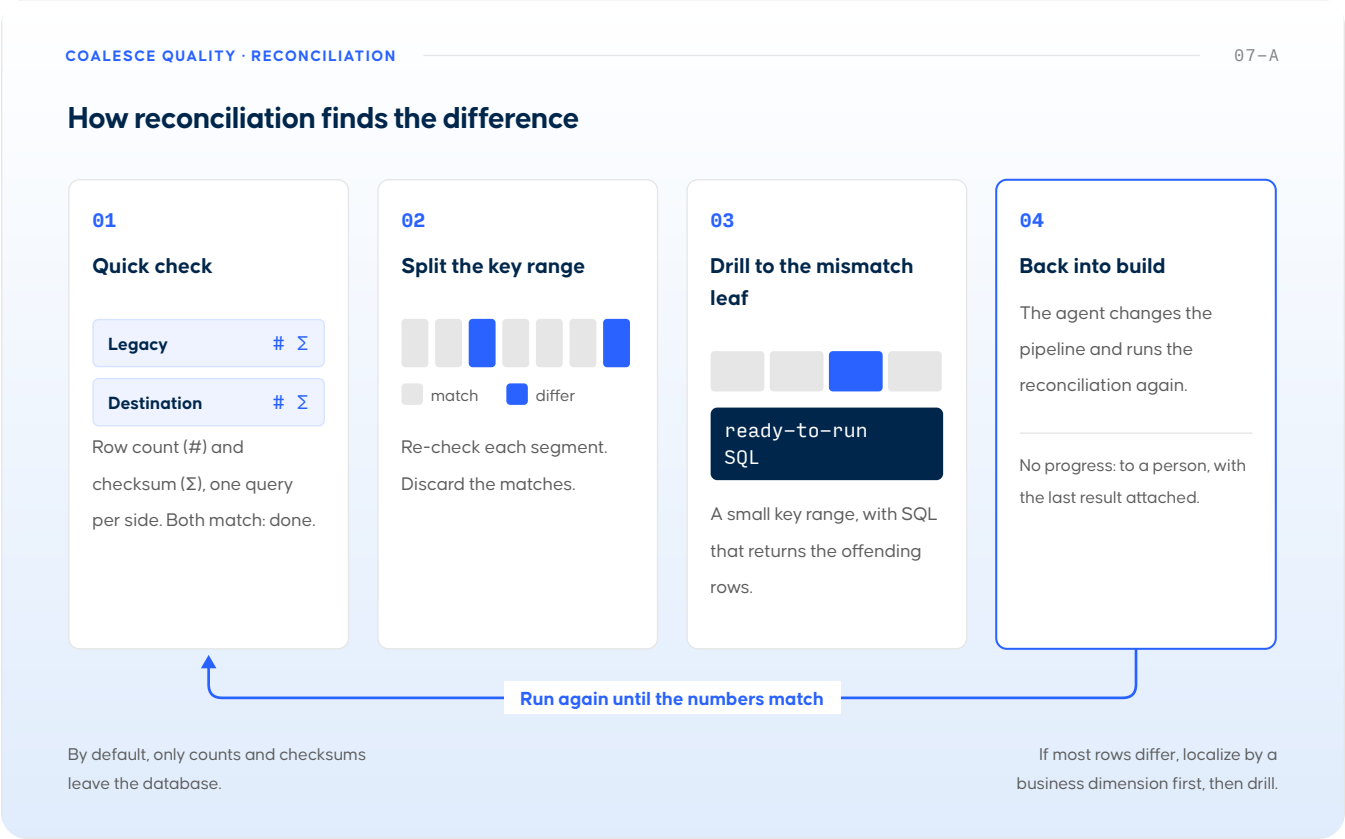
Reconcile data

Reconciliation starts with the cheapest question: does the code run? On Snowflake, for example, a dry-run of the DDL (coa create --dry-run) shows any column whose type did not resolve, and an EXPLAIN or a trial CREATE TABLE shows the code executes before any data comparison spends a cent; the in-app Validate Create and Validate Run actions run that EXPLAIN for you. Then Coalesce Quality answers the question that matters: do the legacy and destination data sets agree? It compares observable results, not code, because modernized code is supposed to look different. Final-output parity is the minimum. Intermediate comparisons add diagnostic evidence where equivalent stages appear on both sides, and they earn their keep on long-running pipelines, where a mismatch at an early stage tells you sooner what final-output parity would only tell you at the end. Five pieces make it work.

Define the comparison. A reconciliation compares one data set against another. Declare the key columns that identify a row, and declare a real composite key rather than concatenating one, because the comparison needs the index. Declare the columns to compare or the columns to exclude, and map columns whose names changed in flight (invoice_amount on one side, total_payment on the other), which is exactly what modernize-in-flight renaming produces. Filter both sides the same way, and set a cutoff for rows still in flight so a lagging replica doesn't look like missing data. The exclusion list is your documented volatile-column rule set: load timestamps, run IDs, ingest dates, anything legitimately different between environments. The agent proposes it; the record shows which columns are excluded and which are compared, and a person adjudicates.

Quick check. One aggregate query per side returns a row count and a checksum over the compared columns. If both match, the reconciliation is complete, whatever the table size. Row counts catch gross failures in seconds. The checksum is the check practitioners trust most. It is only as good as its normalization: types cast to text, nulls kept distinct from empty strings, ordering fixed. Then a difference in the checksum is a difference in the data.

Drill-down. When the aggregates disagree, the key range splits into segments, and both sides are re-checked, one query per side per level. Discard segments that match; split only the ones that differ. Drilling stops at a mismatch leaf: a key range small enough to inspect, carrying ready-to-run SQL that returns the offending rows on either side. This is how you find where a billion-row table differs without comparing a billion rows. It has one failure mode, and the docs name it: if most rows differ, nothing can be discarded and the drill degenerates into thousands of queries. Localize first with an aggregate comparison grouped by a business dimension, then drill into the wrong group.



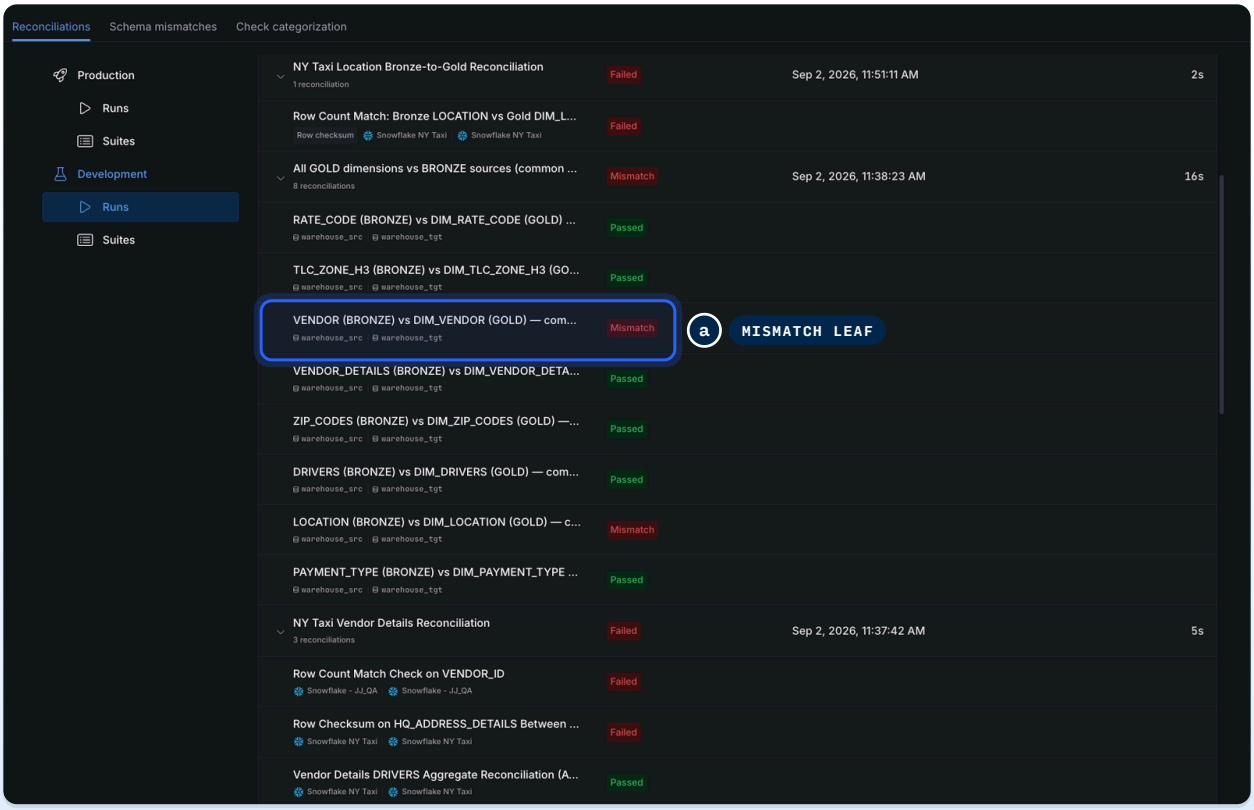
Privacy levels. By default, only counts and checksums leave the database. No row value is retrieved, logged, or shown to the agent. Two opt-in levels exist. One adds the key values of differing rows; one adds sample row values behind an explicit consent flag. You rarely need either. The mismatch leaf already tells you which key ranges differ, and the SQL it hands you runs where the data lives. Where even key boundaries are sensitive, a hash-based segmentation keeps them out of the log.

This is also where the governance posture of this guide meets its one deliberate exception. Agents work against metadata by default, and the build runs in the developer sandbox. The source side is different. Where the data owner allows it, reconcile against production source data, because that is the truest test; on Snowflake the common shape is a read-only secure share of the production source. The target stays in development until it is promoted. When the agent needs to query data, count-only reconciliation lets it validate the columns that matter without the values ever entering its context.

Sensitive data. Here is the question every regulated buyer asks, and most migration content dodges: value-level validation means querying real data, so what happens when the data is patient records, account balances, or claims? Coalesce Quality computes the counts and checksums where the data lives and hands the agent the

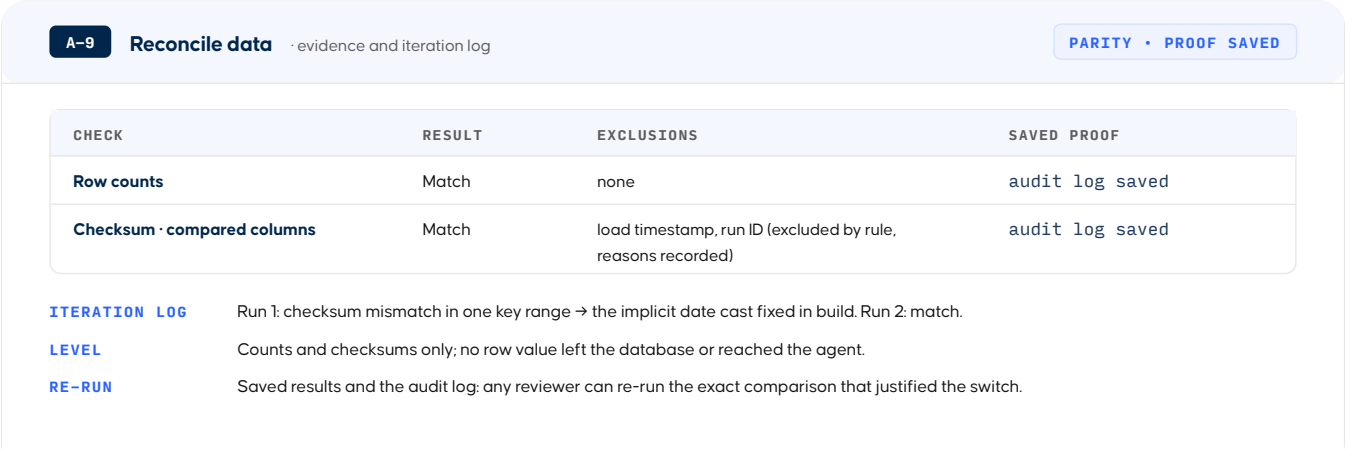
result, not the rows. A regulated estate validates the same way as any other. The agent learns whether the numbers match, and at the default level no source value passes through a model.

Testing suite as code, audit log as evidence. A Coalesce Quality reconciliation suite is a versioned file that declares the connections, data sets, comparison columns, keys, exclusions, and mode. It is planned, so you can read the exact SQL both sides will run before any query is spent, then run. The audit log records every query as it was actually executed, every count, every checksum, every timing, and every mismatch leaf. A suite promoted to production becomes a platform asset that runs on the schedule you set, or when triggered through the API, and opens an issue through your existing alerting and ownership when it fails. That is validation as code, in product form. The same suite runs on a laptop during the first file and on a schedule during the parallel-run window, and the evidence it leaves is a link, not a recollection.



When the numbers do not match. A mismatch is not a verdict; it is the next input. The agent reads the reconciliation result, changes the pipeline, and runs the reconciliation again, and it repeats that until the numbers agree. Bound the loop: it should never one-shot a conversion or run forever, and a loop that stops making progress hands the file to a person with the last result attached. Keep the iteration log, what changed

between runs and why, because that log is the record a reviewer opens later. The setup is small. The Coalesce Quality CLI sits beside whichever coding agent your team runs, with credentials and the team's skills, and the goal is one sentence: make this pipeline match its source. Given that, the agent iterates until it does.



What the record holds. Every exclusion sits in the record with its reason, every accepted mismatch carries the written rationale a person adjudicated, and every run leaves its saved results and audit log, so any auditor, reviewer, or skeptic can re-run the exact comparison that justified the switch. "Old equals new" stops being an assurance and becomes a query someone can execute. Parity and correctness are still different questions. Port as-is validates strict parity. Modernize in flight establishes parity first, then applies the short, documented list of deltas from the semantic queue (Chapter 06), each with its own before-and-after evidence.

In Coalesce

Reconciliation lives in Coalesce Quality: its CLI sits beside the coding agent, and each suite is a versioned file in your repository. Promoted to production, a suite runs on the schedule you set and opens an issue through your alerting when it fails. The screenshot in this chapter shows one run: the suite, the counts and checksums, and one mismatch leaf.

What good looks like

Reconciliation is done when the numbers match and the record says exactly how you know.

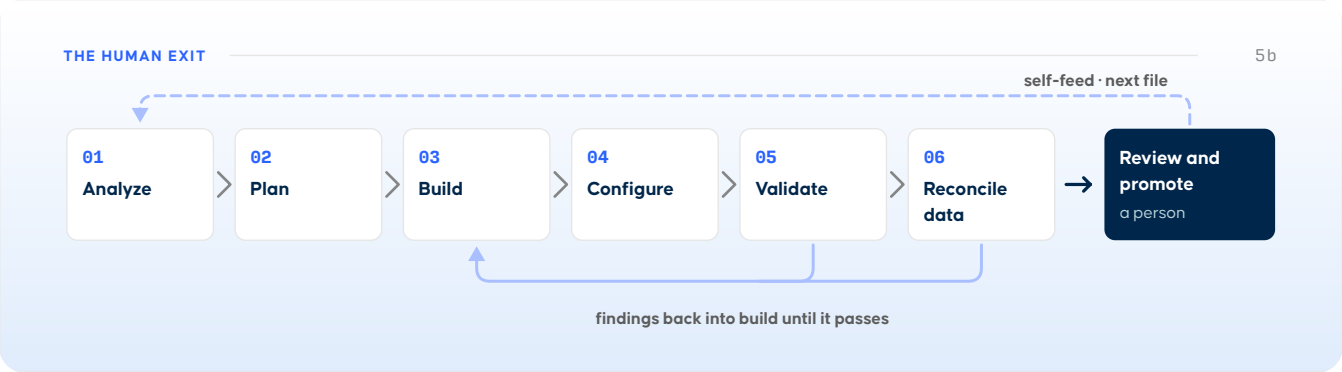
- The code runs on the destination: the dry-run, EXPLAIN, or trial CREATE came back clean before any comparison ran.
- The Coalesce Quality suite is in the repository, and it declares real keys, the compared or excluded columns, renamed-column mappings, matching filters on both sides, and a cutoff for rows in flight.
- Final-output parity holds: row count and checksum match on both sides, with intermediate comparisons where they earn their keep, on long-running pipelines.
- Every excluded column appears in the record with its reason, and a person has adjudicated the list.
- Every mismatch along the way has a line in the iteration log: what differed, what changed, and the result of the re-run.
- The saved results and audit log are where the reviewer will look for them.

The Reconcile data card above shows REVENUE_FACT's evidence: counts and checksums matching, two volatile columns excluded by rule, and the queries saved so anyone can re-run them.

CHAPTER 08

Review, Promote, and Resolve Issues

Everything up to here, the agent could check for itself. This chapter covers what it cannot. A person reads the reconciliation result, the graph, and each Node in turn, marks the pipeline reviewed, and promotes it to the next environment. Anything that would change what the pipeline produces goes to the semantic queue, not into the pipeline. Chapter 09 takes reviewed pipelines from one file to a whole program.



What you need

- From Chapter 07: a passing reconciliation, its saved results and audit log, and the iteration log.
- The pipeline open in Coalesce: the graph, each Node, and column lineage.
- The pull request with its diff, and a reviewer who can approve it.
- Your semantic queue: tickets for any logic you question but do not change before parity.
- A UAT or QA Environment to promote into.

The human review

EXIT

Review and promote.

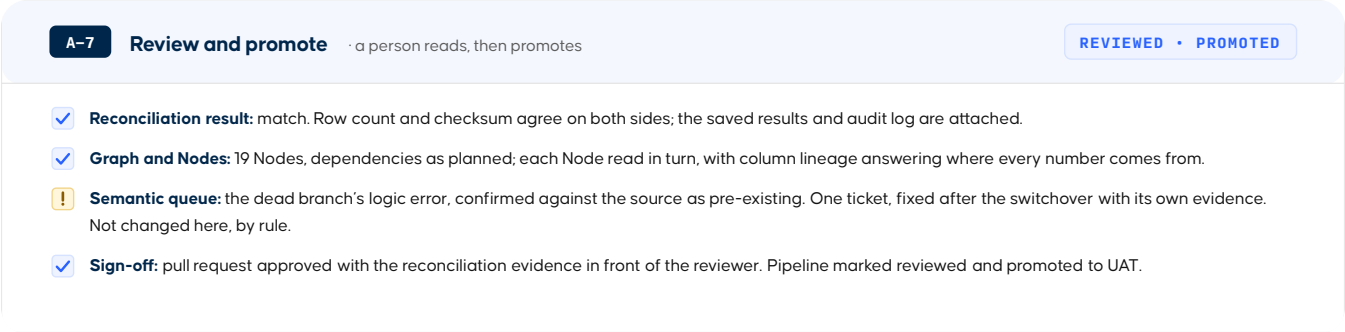
The reconciliation has passed. Now judgment sits on top of a deterministic result. The agent already read the test output on every iteration; that is how the file got here. What remains is the read only a person can do: does this pipeline make sense, and does it do what the procedure did?

If you want one, the agent can build a side-by-side for you: for every Node, the source SQL that fed it next to the SQL the Node renders. In delivery, passing checksums and reconciliations are usually enough, so keep the side-by-side for the Nodes you want to read closely. You are not reading a 1,400-line procedure; you are reading one bite-sized piece and what changed. One caveat governs how you read it. Once structural changes have been applied, the two sides will not match textually, by design. A

select-star expanded into named columns will never diff clean. The side-by-side tells you what changed; only data reconciliation tells you whether it still produces the same numbers.

Anything you notice that changes meaning (a suspected logic error, a dead branch that might not be dead) goes to the semantic queue for after parity; the numbers already match, so it is not a fix. A structural issue, such as a Node label off your naming standard, goes back into build, and the reconciliation runs again before the pipeline is marked reviewed.

> **Where it fails**: logic that was wrong on both sides. A join that compiles, passes every structural check, and returns the same wrong rows the legacy procedure returned will reconcile cleanly, because parity is a comparison, not a proof of correctness. Your read of the Nodes and the semantic queue are the defense; the reconciliation cannot be.



Two species of validation

Sort every check your program runs into one of two buckets, because they behave differently and deserve different trust.

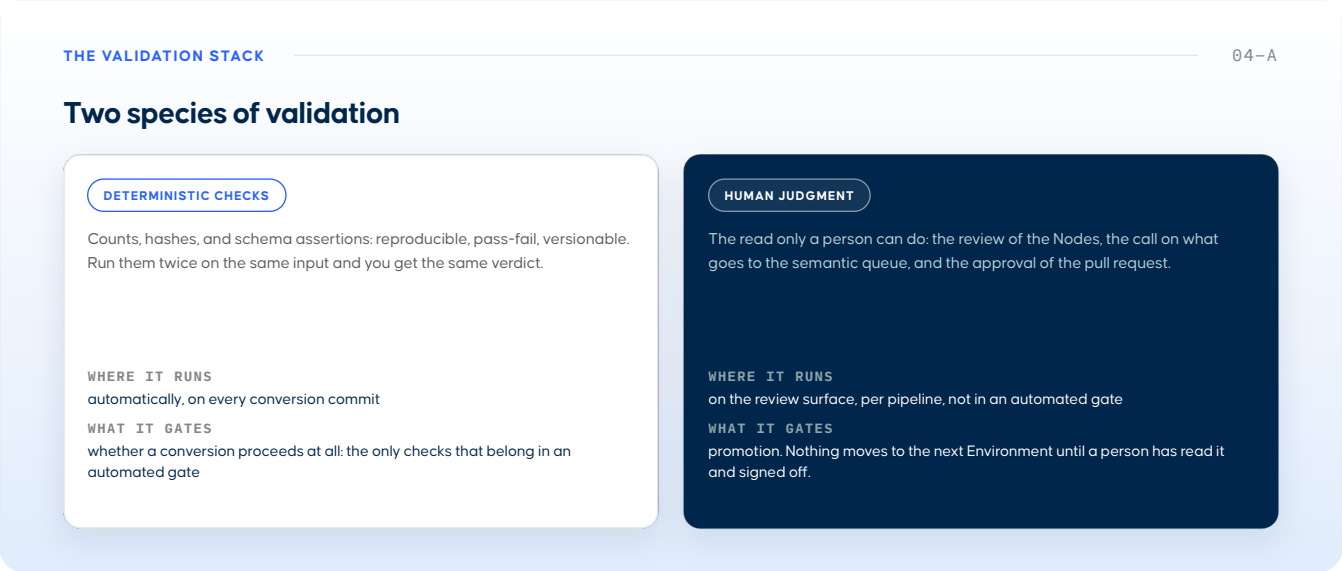
Deterministic checks are the counts, hashes, and schema assertions: reproducible, pass-fail, versionable. Run them twice on the same input, and you get the same verdict, which makes them the only checks that belong in an automated gate.

Human judgment is the read only a person can do: the review of the Nodes, the call on what goes to the semantic queue, and the approval of the pull request. It runs in the review surface, not in an automated gate, and it gates promotion: nothing moves to the next Environment until a person has read it and signed off.

Script sprawl is the anti-pattern that passes for validation. That means one-off comparison scripts, written in the moment, unversioned, impossible to re-run, evidence evaporating the day after cutover. Every migration produces the temptation. A

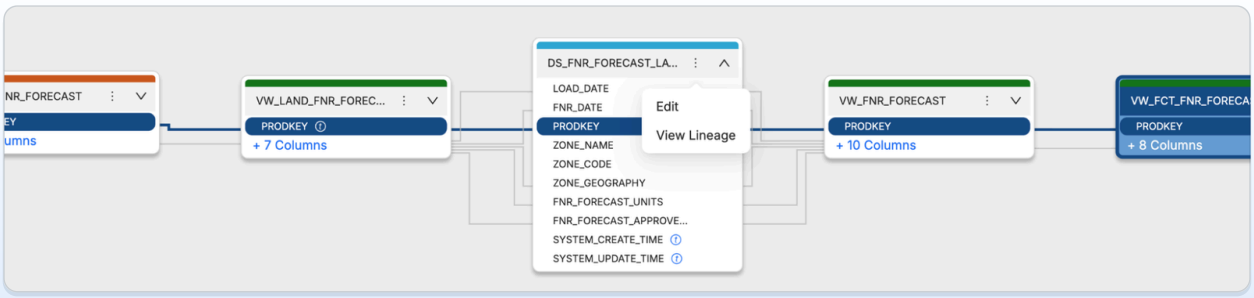
one-off query to test your own hunch is fine. When you catch yourself running it again, template it and add it to the testing suite. The teams that resist sprawl share one practice.

Validation as code. Checks live in the repository, versioned like everything else. Deterministic gates run automatically on every conversion commit; a person reviews before promotion; all results persist. This is the "agent SQL is easy to generate, hard to trust" critique turned into a build system. In practice, it is the same CI discipline your team applies to application code, applied to proof. When someone asks, six months later, how you knew pipeline 214 was right, the answer is a link rather than a recollection.



The review surface

All of it converges on a human looking at work and deciding. That decision is only as fast and as credible as the surface it happens on. This is where reviewing agent output diverges from skimming agent chat logs. The sequence is fixed. First, the reconciliation result, which you expect to show a match. Next, the visual graph, which shows the decomposition at a glance: FinCo's 19 Nodes, their dependencies, their flow. Then each Node in turn, with column lineage answering where a number comes from without archaeology, and the side-by-side open beside it if the agent built one. When every Node has been read, mark the pipeline reviewed, which in practice means approving its pull request and moving its ticket, and promote it by deploying to the next Environment. Engineer sign-off is fast, in exactly the proportion the review surface is built for. Fast, rigorous sign-off is the entire economic argument for keeping humans in the loop at scale.



WHERE DOES THIS NUMBER COME FROM, WITHOUT ARCHAEOLOGY

THE FINCO FILE **the evidence package**

Pipeline REVENUE_FACT, ready for sign-off. Row counts: match. Hash comparison: match across 19 Nodes, with two columns excluded: a load timestamp and a run ID. Both flagged by the program's exclusion rules, both noted in the record with the queries that support them. Semantic queue: one item, the dead-branch logic error, confirmed as pre-existing and ticketed for after the switchover. The audit log is attached to every check.

The reviewing engineer opens the graph, walks the lineage, and signs off. Elapsed review time is short. The confidence is not borrowed from the agent; it is standing on the evidence.

> **"I'll probably spend 90% of my time within the CLI. But the 10%, the part that arguably is the most important, the validation and the human-in-the-loop piece, I'm using a different review surface for a different intent. I want to visually see the data and understand how everything is flowing and how it all ties together."**
>
> — **Jesse Marshall** · Chief Customer Officer, Coalesce

Where the humans are

In delivery, roughly nine-tenths of the time goes to the CLI, working with the agents as they analyze, build, validate, and iterate on reconciliation results. The remaining tenth arguably matters most, and it happens somewhere else on purpose. Building and reviewing are different intents, and they deserve different surfaces: the terminal is where an agent works through files, and the visual graph is where a human absorbs a pipeline, jumping to any grain and any level of detail. You see how everything ties together in a way that reading text at 1,400-line scale never allows.

Around that review, people do the teaching and the judging: skills, standards, and the first files; reviews, sign-offs, and the deltas that change behavior. That work is what turns fast output into trusted output.

In Coalesce

Review uses three surfaces in Coalesce: the pipeline graph, each Node with its rendered SQL, and column lineage, which answers where any number comes from. If the agent built a side-by-side, keep it open beside them. When every Node has been read, approving the pull request is the sign-off. The two screenshots in this chapter show the side-by-side and a Node open in the graph.

What good looks like

The pipeline is done when a person has read it, approved it, and moved it on.

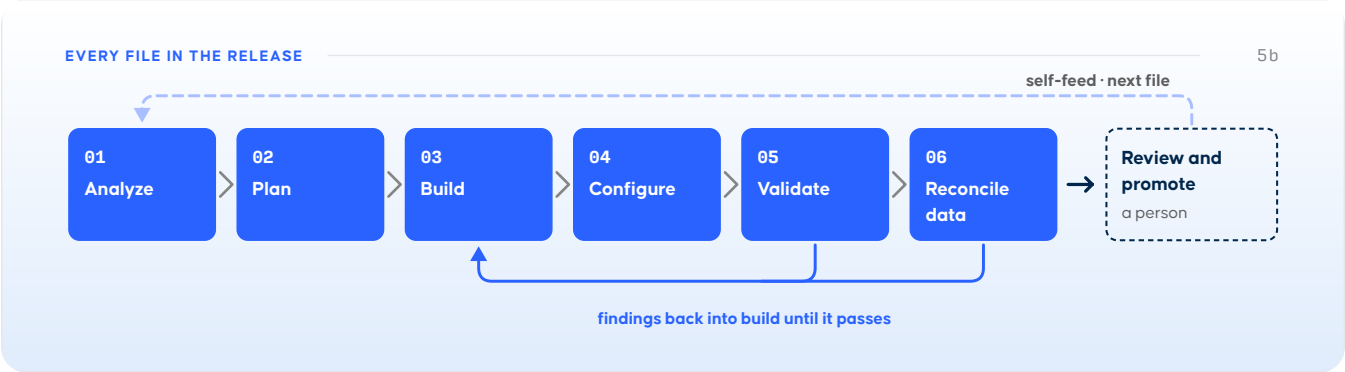
- You read the reconciliation result, then the graph, then every Node, in that order.
- Every number you questioned has an answer in its column lineage.
- Anything that would change what the pipeline produces is a ticket in the semantic queue, not an edit to the pipeline.
- A person approved the pull request with the reconciliation evidence in front of them. Nothing merged on a green check alone.
- The pipeline is marked reviewed and promoted to UAT or QA.

The FinCo card above is REVENUE_FACT's sign-off package: the evidence, the one item in the semantic queue, and the reviewer's sign-off.

CHAPTER 09

Deploy and Operationalize

One file proves the method. Three hundred files is a different discipline, not because the conversion changes but because everything around it does. This chapter is the program layer. Chapter 08 left you with one reviewed, promoted pipeline. Here a coordinating agent runs the loop across a release, file by file, while your reviewers set the pace, and the release cuts over through a dual-run window you planned for. Chapter 10 covers what the program learns along the way and what your team keeps when the last release goes live.



What you need

- From Chapter 08: reviewed and promoted pipelines, and a review pace your team can sustain.
- The release's files on your board, sized to what your reviewers can sign off in a sprint.
- A coordinating agent with the playbook, the batch, and the same toolchain and permissions the single-file loop used.
- Development, UAT or QA, and production Environments, and a developer who owns the merge to production.
- A decision, agreed with the business, on freezing or syncing legacy changes during each release, and the legacy system kept runnable until parity holds through the parallel run.

Orchestrating the agents

A program is a batch of files and a goal. Give a coordinating agent the list, the playbook, and one instruction: run each file through the loop, reconcile it, and move to the next. It dispatches the work to subagents one file at a time and reports back as each one passes. There is no schedule to tune. The work runs for as long as the batch has files in it, which is where the around-the-clock compression everyone cites actually comes from.

Early in the program, run files one at a time with a person watching, while the team calibrates its standards and its trust in the checks. From the second sprint, hand the coordinator a batch.

The constraint is the same as everywhere else in this guide: throughput is bounded by review bandwidth, not agent capacity. Size the batch to what your humans can read and sign off, or the queue piles up behind the review gate: converted, reconciled, and unpromoted.

PROGRAM ORCHESTRATION 05–A

One file at a time first, then a coordinator and a batch



Throughput is bounded by review bandwidth, not agent capacity, so size the batch to what your humans can read and sign off.

Failsafes and the dual-run problem

Every long migration lives through a dual-run window, and it comes per release, not once for the program. A developer merges each release's pipelines to production after they have been tested in UAT or QA, and legacy stays the truth until the new production pipelines have proven accurate over the parallel run. So for days or weeks, the old estate keeps running and keeps changing while the new estate fills in behind it. Plan for that window explicitly:

A freeze-or-sync policy for in-flight changes. The legacy estate will not hold still. Decide which your business can live with. A freeze per release is the easiest to manage and the one to prefer. If the business cannot stand still, define the delta process instead: legacy changes during a release get logged and the affected conversions re-run before that release's cutover. It works, and it is twice the work. Silent legacy drift (the change nobody re-converted) is the quietest way a validated migration ships wrong numbers.

A parallel-run period per release. Old and new run side by side. Parity stays green across real operating days, not just the validation snapshot, before legacy retires. This period is boring by design.

Rollback that is real. Because every change is a commit, reversion is a Git operation plus a re-point. Yet that holds only if legacy is still runnable. Keep it runnable until parity has held through the parallel window. Then rehearse the rollback once per program, early, when it costs nothing. Rehearse it, and expect not to use it: a dual-run window of days to weeks is what surfaces the surprises before switchover rather than after.

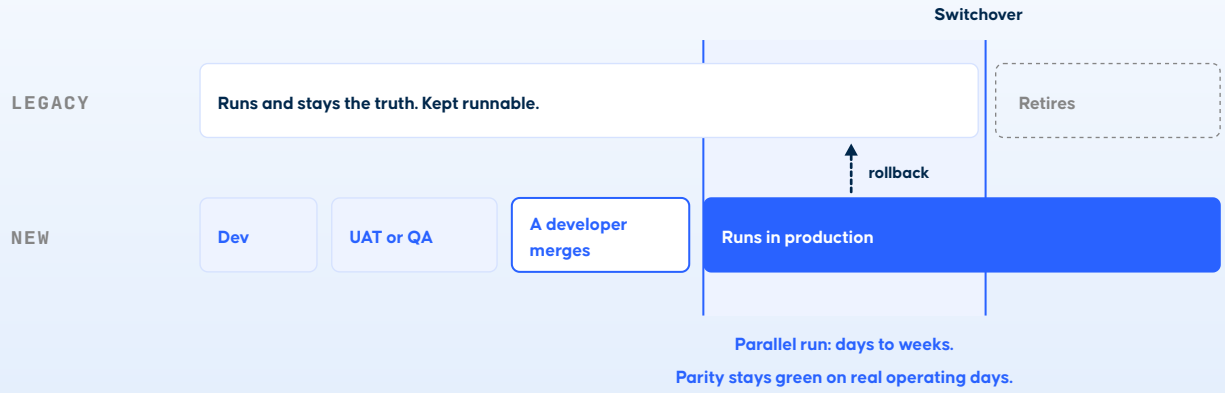
ONE RELEASE · REPEAT PER RELEASE 09–A

The dual-run window

Before the release: freeze or sync?

Freeze (preferred). Legacy holds still for the release. The easiest to manage.

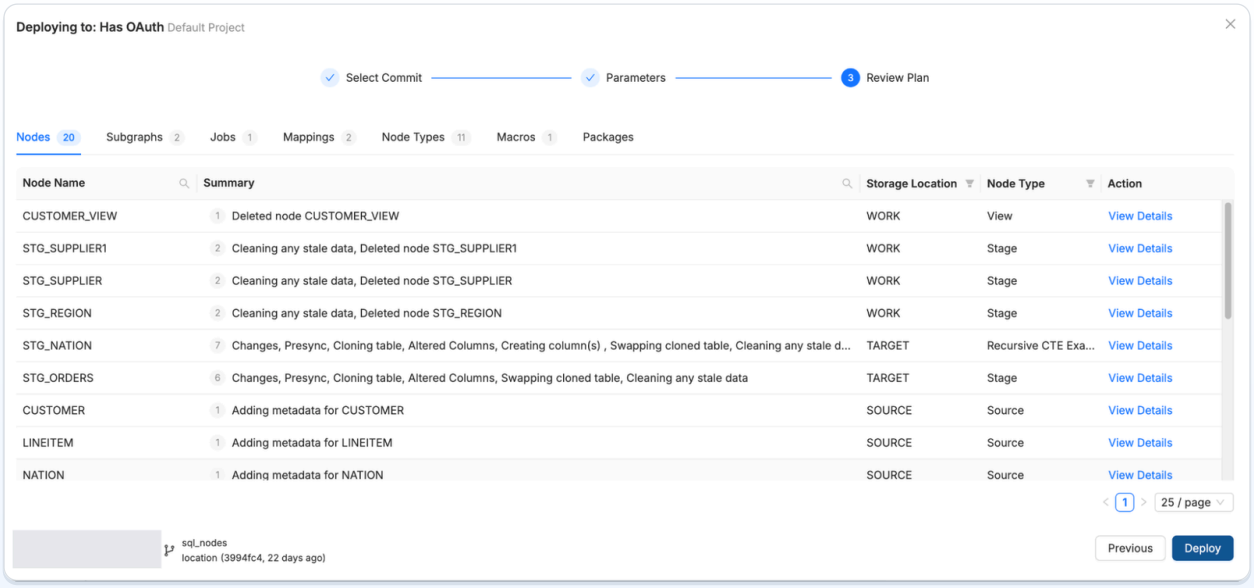
Sync. Log legacy changes and re-run the affected conversions. Twice the work.



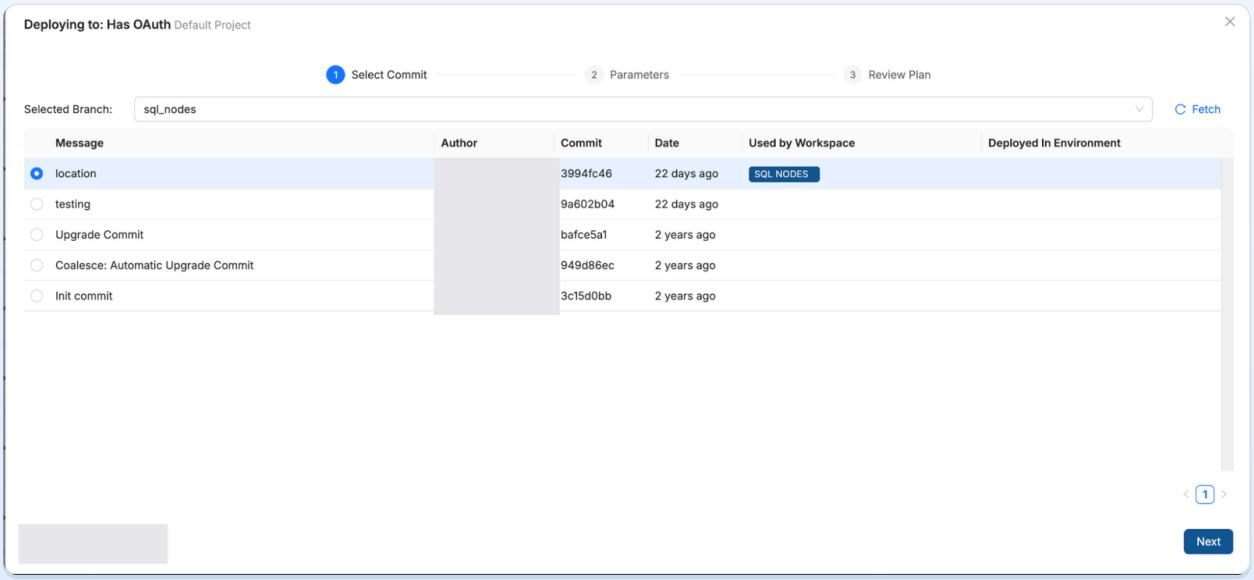
Rollback (dashed): a Git operation plus a re-point. Real only while legacy stays runnable.

Rehearse it once per program, early. Expect not to use it.

a THE DEPLOY PLAN FOR ONE RELEASE



b A PRIOR COMMIT SELECTED FOR THE ROLLBACK REDEPLOY



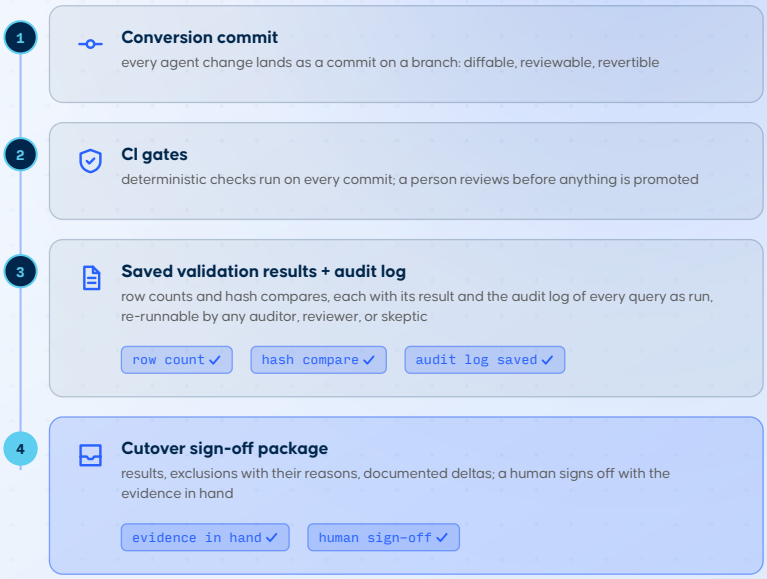
REVERSION IS A GIT OPERATION PLUS A RE-POINT

THE EVIDENCE TRAIL 04–B

Every stage leaves an artifact behind

By cutover, the proof already exists, assembled from the work itself rather than reconstructed after the fact.

"Old equals new" stops being an assurance and becomes a query someone can execute.



THE FINCO FILE the program

340 procedures parsed and bucketed; 118 dropped on usage evidence before conversion began. The remaining 222 go live in five releases, ordered by dashboard impact, with release three deliberately scheduled clear of quarter-close. Files ran one at a time, with a person watching, through the first sprint while the team calibrated. From the second sprint a coordinating agent worked the batch, file by file. The batch was cut once, when the review queue backed up behind the sign-off gate.

One legacy change landed in the middle of release four; the delta process caught it, and two conversions re-ran before cutover. The rollback rehearsal in release one took an afternoon and was never needed again. What FinCo's team owns now: the pipelines, the skills, the tracker, and the habit.

In Coalesce

The program layer runs on the deployment path of the platform: the coordinating agent works through the CLI with the same permissions the single-file loop had, and each release deploys to a UAT or QA Environment, then to production. The screenshots

in this chapter show one release's deploy plan and the redeploy of a prior commit, which is what a rollback is.

What good looks like

A release is done when its pipelines run in production, legacy has retired behind them, and nothing surprised you in between.

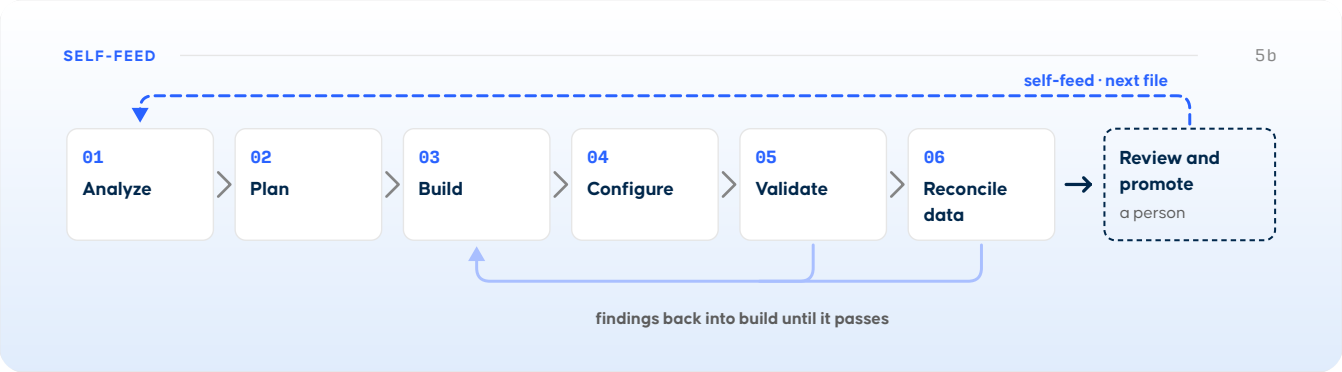
- The freeze-or-sync decision for the release is written down and agreed with the business. Under sync, every logged legacy change has a re-run conversion.
- The coordinator ran the release in batches your reviewers could clear, and no converted pipeline is waiting unreviewed.
- The pipelines moved from development through UAT or QA, and a developer merged them to production.
- Parity held through the parallel run on real operating days, and the cutover stayed clear of the business calendar's risk windows.
- Legacy stayed runnable until parity held, and the rollback was rehearsed once, early in the program.
- The release's tickets are closed on the board, and the burndown shows it.

The FinCo program card above follows the whole estate through its releases, including the one legacy change mid-release that the delta process caught.

CHAPTER 10

Measure Progress and Maintain the Migrated Environment

The releases have cut over and legacy is retiring behind them. This chapter looks at what the program leaves: the standards the loop taught you along the way, and the governed estate your team now runs. It is the last chapter of the method. Chapter 11 closes the guide with the question the method leaves open: who runs it.



What you need

- From Chapter 09: the releases cut over, and the burndown from your board.
- The iteration logs from the loop, where recurring fixes show up.
- The repository where your standards and your team's skills live, and someone who owns changes to them.
- The evidence trail: saved reconciliation results and audit logs for every pipeline that cut over.

Measure progress

Read the board at the end of every sprint. The burndown from Chapter 03 is the clock of the program: files closed against the plan, and the sprints still ahead. Two trends tell you the method is working. The review each file needs should keep falling as the skills and standards absorb what the early files taught, and validation and reconciliation should stay the longest part of the work, because that is where the proof lives.

Self-feed

After the loop, self-feed records what the conversion taught the program. When the agent fixes the same kind of mismatch on a third file, that fix is a standard waiting to be written down. Clarified conventions go back into the team's skills. Reusable dialect rewrites join the idiom set. Treat adopting a standard as a review gate: a person reads the candidate, tests it, and adds it deliberately to the standards the agent follows. Otherwise, one successful fix becomes an unexamined standard for every future

migration. Some teams answer the same problem by fine-tuning a model on its failures. The playbook promotes rules instead, because a rule is inspectable and a weight is not.

A–10 **Self-feed** · standards after a person reviews

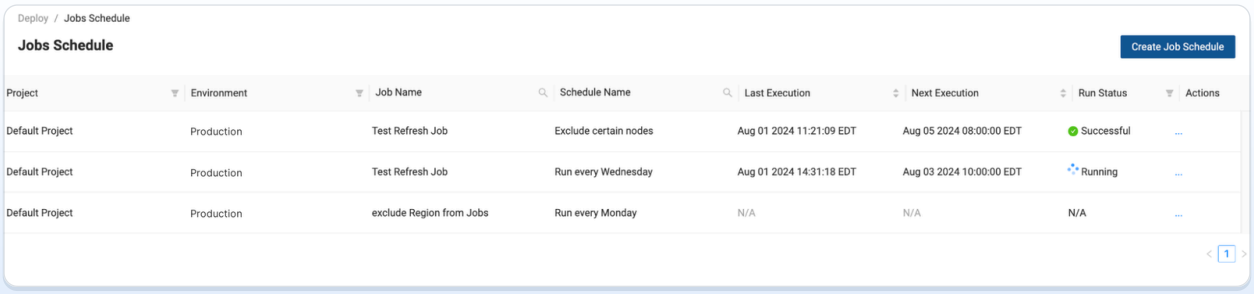
1 STANDARD ADDED

RECURRING FIX → CANDIDATE STANDARD → A PERSON REVIEWS → STANDARD + SKILL

THIS FILE	The date-cast fix, seen on a third file, written up as a candidate standard, read and tested by a person, then added to the rules the agent follows. The next occurrence is a rule, not a model call.
ALSO KEPT	The naming convention clarified in FinCo's skills; the GETDATE() rewrite joins the reusable idiom set.

What lands at the end

Cut over the last release, and what remains is not old code in a new warehouse. It is a governed estate: pipelines decomposed into inspectable steps, documented and lineage-mapped as a byproduct of how they were built. An evidence trail runs from first conversion to final cutover. The migration ends, but the way of working it forced becomes the way the estate runs from now on: agents building, humans reviewing, evidence deciding. You moved the pipelines, and along the way you changed what maintaining them looks like.



AFTER CUTOVER, THE PIPELINES RUN LIKE ANY OTHER

In Coalesce

After cutover, the migrated pipelines run like any other: as scheduled Jobs, with a run history your team already knows how to read. The reconciliation suites that guarded the parallel run keep running on their schedule and open issues through your alerting

when they fail. And your standards and skills live in the repository, versioned like the pipelines they came from.

What good looks like

The program is done when the estate runs on the new way of working without a migration team around it.

- Recurring fixes from the iteration logs have been read, tested, and added to your standards and skills by a person, not adopted because they recurred.
- Reusable dialect rewrites are in the shared idiom set, and the skills repository has an owner.
- Every pipeline that cut over has its saved reconciliation results and audit logs in the evidence trail.
- The reconciliation suites that guarded the parallel run now run on a schedule and open issues through your alerting when they fail.
- The board shows the last release cut over, and legacy is retired.

The Self-feed card above shows one lesson from REVENUE_FACT becoming a standard: the date-cast fix, reviewed and added to the rules the agent follows.

CHAPTER 11

Choosing Who Runs Your Migration

Everything in this guide so far is method, and the method is open. The last decision is operational: who runs it. There are three honest configurations, and the playbook is the same in all three:

- **Self-directed.** Your team drives, on the platform, with the toolchain from Chapter 03.
- **Coalesce-delivered.** Coalesce AI Migration is a fixed-price engagement: purpose-built agents do the build, and Coalesce engineers sign off on every pipeline.
- **Partner-delivered.** The ecosystem runs the same pattern with its own accelerators and enterprise-specific skills.

What differs is who operates it, and how the risk is priced. The customer proof that follows comes from 1-800-Flowers.com's migration onto the Coalesce platform, which did not use agents.

1-800-FLOWERS.COM'S MIGRATION, BY THE NUMBERS

88
pipelines migrated from a legacy system in 4 months

75%
faster nightly batch processing after migration

12x
faster deployments, enabling true CI/CD practices

Where migration engagements actually bloat

If you take one thing from this chapter into a vendor conversation, take this: the open-ended part of every migration is proof. Conversion is bounded; the estate has a size, and the parse tells you what it is. Validation and reconciliation are where discovered edge cases live, and edge cases are discovered, not scoped. Agents write the code fast. A person still reviews it, and although the share that needs review is falling as agent output improves, validation and reconciliation remain the longest part of the work. Under time-and-materials pricing, that open-endedness is billable by construction. Under fixed pricing, it is the vendor's problem by design. Neither model is dishonest. But before signing, you should know which side of that risk you are holding, because the contract decides it.

The incentive mechanics follow directly. Hourly models reward duration; outcome models reward completion. Scope creep is not a character flaw in your vendor; it is the equilibrium of an open-ended contract meeting discovered complexity. Which is also why migration, of all projects, *can* be fixed-priced: it is a known requirement. The end

state is definable up front: these objects, converted and proven, by this date. When the work is definable, and the quote is still "it depends," the quote is telling you something about the model, not the work.

> **"It was a much easier migration than I thought it would be. The learning curve for Coalesce was minimal."**
>
> — **VP of IT** · 1-800-Flowers.com

The question procurement hasn't caught up to

When agents do the translation and the validation, a rate card stops describing the engagement. Asking for the staffing table behind an agentic migration is asking the wrong document to explain the price. The evaluation that works is the next-best-alternative: what would this cost in internal team-quarters, as an outside firm's hours, or as building your own agentic capability, and what is each path's risk of not finishing? Price the outcome against the alternatives, not against a headcount model that no longer resembles delivery.

The diligence checklist

Eight questions, for anyone, including us:

- [ ] **How is validation scoped and priced: included in the quote, or hourly on top?**
  The open-ended part is the part to ask about.
- [ ] **Is this quote based on an actual parse of our estate, or a rate-card model?**
  A vendor that has not parsed the estate is guessing at the price, and Chapter 03 showed there is no longer an excuse for that.
- [ ] **Is the timeline contractual, and what happens if it slips?**

- [ ] **What evidence accompanies cutover?**
  Saved reconciliation results and audit logs per pipeline, the Chapter 07 standard, or a summary deck?

- [ ] **Who signs off on each pipeline, and does your team review in-flight or only at the end?**

- [ ] **What can your agents read and write in our environment, and how does value-level validation handle our sensitive data?**
  Chapter 03's posture is the good answer: read-only defaults, scoped write grants, human-owned cutover.

- [ ] **What transfers to your team when it ends (playbooks, skills, the tracker), and what leaves with the vendor?**

- [ ] **What does exit look like mid-program?**
  Releases make this answerable: completed releases are yours, validated and running. If the answer is murkier than that, ask why.

A good answer to all eight is what accountability looks like in this category. You leave the conversation with real information either way.

One more file

Wherever the decision lands, the first step is the same one the method starts with: one file, one conversion, one review. The method is open, the docs are written for your agent as much as for you, and the most representative procedure in your estate is a better evaluation than any vendor call.

To run it yourself, start with the Local Development Setup Guide, which installs Coalesce Desktop and the coa CLI and connects your warehouse. To have Coalesce run it, book a 30-minute scoping call for Coalesce AI Migration: the team assesses your environment and maps the path to a fixed quote.

ABOUT COALESCE

Coalesce is the data operating layer for analytics and AI. It unifies data transformation, cataloging, and quality in one governed platform, so data teams and AI agents build to the same standard, in the same flow, with the same context.

NEXT STEP

Book a 30-minute scoping call →

Local Development Setup Guide →