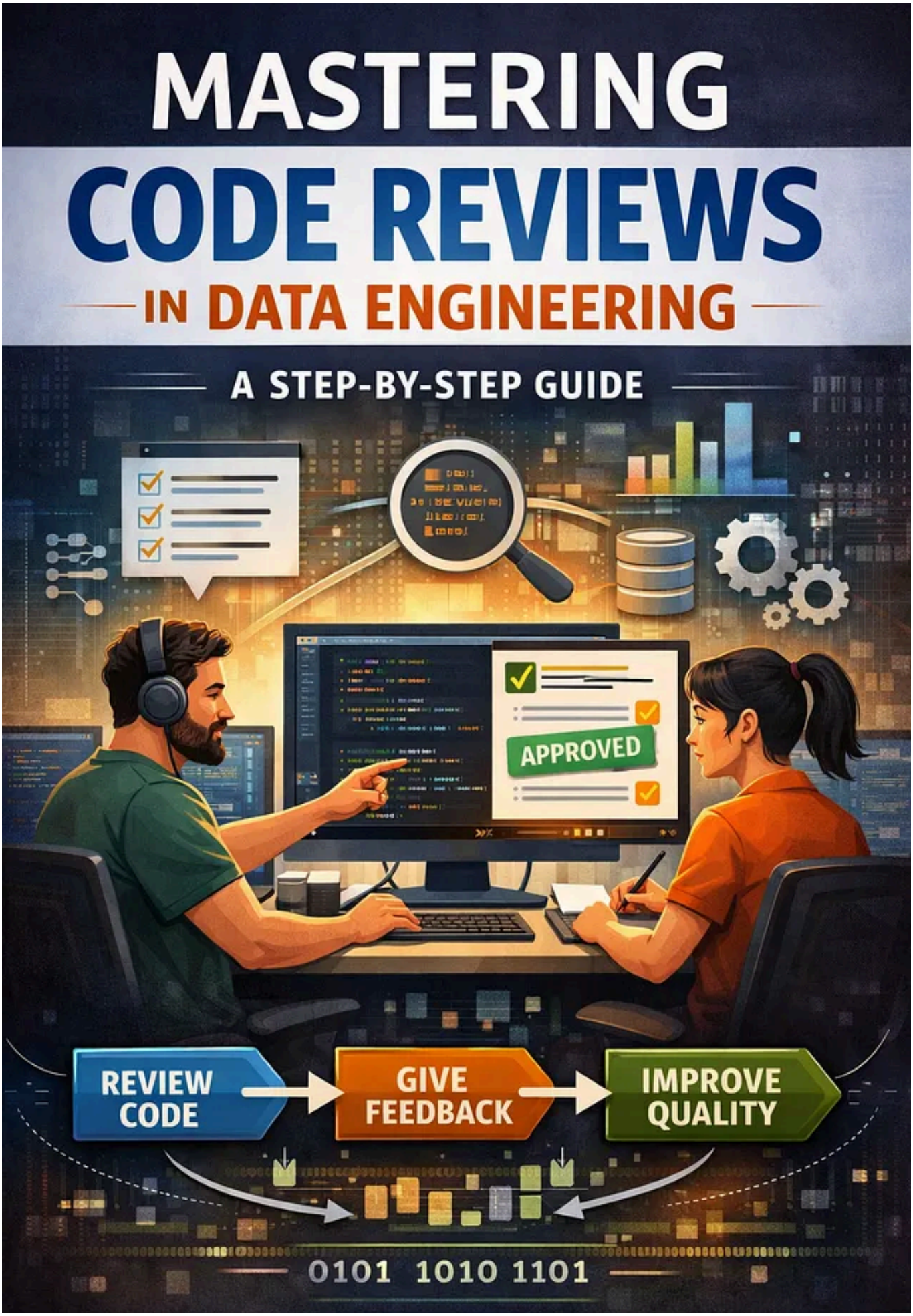
MASTERING
CODE REVIEWS
IN DATA ENGINEERING
A STEP-BY-STEP GUIDE
APPROVED
REVIEW CODE
GIVE FEEDBACK
IMPROVE QUALITY
0101 1010 1101

Mastering Code Reviews in Data Engineering: A Step-by-Step Guide

Anurag Sharma 5 min read · Feb 2, 2026

Hey there, data enthusiasts! If you are knee-deep in the world of data pipelines, ETL jobs, and big data frameworks, you know that code reviews are not just a checkbox — they are a lifeline for building robust, scalable systems.

In data engineering, where a small bug can cascade into massive data corruption or performance bottlenecks, effective code reviews can save hours (or days) of debugging downstream.

In this blog, I will walk you through a step-by-step guide to conducting code reviews in data engineering. Whether you are a reviewer or the one submitting code, these practices will help elevate your team's output. I will draw from real-world experiences with tools like Apache Spark and Python-based pipelines, but the principles apply broadly.

Let's dive in!

What is Code Review?

Code review is the collaborative process where team members examine proposed code changes (usually via pull requests) before integration into the main branch. In data engineering, it extends beyond traditional software to scrutinize:

- Data movement and transformation logic
- Handling of volume, velocity, variety (the 3 Vs)
- Reliability under production-scale loads
- Data quality guarantees
- Observability and recoverability

It's a quality assurance ritual that turns individual contributions into collective, battle-tested assets.

Why the Need for Code Review?

In data engineering projects — especially ETL/ELT pipelines — mistakes are not just bugs; they can corrupt datasets, inflate cloud bills, violate SLAs, or break downstream BI/ML models. Structured reviews (peer → lead → demo) dramatically lifted code quality, reduced incidents, and accelerated knowledge transfer.

Core benefits:

- Catch logic errors, inefficiencies, and anti-patterns early
- Enforce standards for scalability, idempotency, and modularity
- Share domain knowledge (e.g., optimal Spark partitioning)

- Ensure compliance (PII masking, audit trails)
- Build trust and collective ownership
- Prevent “it works on my laptop” disasters at the terabyte scale

Teams with rigorous reviews see 50–80% fewer production issues — critical when pipelines run unattended 24/7.

How is Code Review Done? A Step-by-Step Guide in ETL / Data Engineering Projects

In ETL and data engineering projects, code reviews follow a staged, multi-layered approach. Code Review flow (peer review → lead review → demo presentation) is excellent for tool-specific details for modern stacks (Spark, Airflow DAGs, dbt models, SQL transformations, etc.).

Stage 0: Pre-Review (Author Responsibilities — Do This First!)

Before anyone reviews:

- Write a **strong PR description**: What problem? Why now? Business impact? Data volume affected? Tests run?
- Link Jira/ticket, design doc, or data contract.
- Include evidence: sample input/output, small-scale run logs, or dbt docs preview.
- Self-review: Run linters (Black, SQLFluff), tests (pytest, dbt test), and a dry-run on dev data.
- Keep PRs small: One logical change (e.g., “Add customer enrichment in silver layer” not “Refactor entire pipeline + fix 17 bugs”).

Stage 1: Peer Review (First Pass — Fellow Data Engineers)

Goal: Catch obvious issues, share quick wins. Usually 1–2 peers.

Detailed checklist for ETL/data pipelines:

Readability & Structure

- Clear naming: customer_raw vs df1
- Modular: Separate extract/transform/load; use classes/functions/macros
- DRY: No copy-paste SQL/Spark code
- Comments/docstrings for complex logic (e.g., why that window function?)

Functionality & Correctness

- Logic matches requirements (joins preserve keys? Aggregations correct?)
- Edge cases: Nulls, duplicates, schema drift, late-arriving data
- Idempotency: Safe re-runs (e.g., upsert/merge, not append-only blind)
- Incremental vs full loads handled properly

Data Quality & Validation

- Schema enforcement (e.g., Spark StructType, dbt constraints)
- Built-in checks (Great Expectations, dbt tests, custom asserts)
- Null/duplicate/volume monitoring hooks

Basic Performance

- Avoid collecting/toPandas on big data
- Prefer native functions over Python UDFs
- Partitioning/bucketing hints where needed

Testing

- Unit tests present (e.g., pytest for Spark transformations)
- Integration tests or dbt tests run in CI

Provide specific, kind feedback: “Consider broadcasting the lookup table here because it’s <10MB” instead of “This join is bad.”

Stage 2: Lead / Senior Review (Deep Technical Dive)

Escalate to senior/lead once peers approve basics. Focus shifts to architecture, scale, and long-term maintainability.

Deeper ETL-specific checks:

Performance & Scalability

- Spark: Broadcast small tables? Avoid wide transformations? Proper caching/persistence? Skew handling?
- dbt: Incremental models? Materialization choice (table vs view vs incremental)?
- Airflow: Task granularity? Sensors vs time-based? XCom abuse?
- SQL: Avoid SELECT *, push filters early, use window functions efficiently

Error Handling & Reliability

- Retries/backoffs (Airflow default or custom)
- Dead-letter queues for bad records
- Alerts on failures (e.g., Slack/Email/AWS SNS)

Security & Compliance

- No hard-coded credentials → use secrets manager
- PII masking/anonymization
- Least-privilege IAM roles
- Audit logging for data access

Observability

- Metrics: row counts, latency, data drift (integrate Monte Carlo/Elementary?)
- Lineage: Documented via dbt docs or tools like Marquez

CI/CD Readiness

- Lints/tests pass in pipeline
- Deployment script/configs updated

Iterate until the **lead approves.**

Stage 3: Demo Presentation (Final Validation & Knowledge Transfer)

What to cover in the 15–30 min demo:

- **Live walkthrough:** Run pipeline end-to-end on staging/test data
- **Key changes demo:** Show before/after, highlight optimizations
- **Deployment plan:** CI/CD flow, rollback strategy, cutover timing
- **After-support artifacts:**
- Code flow diagram (e.g., Draw.io / Lucidchart showing bronze → silver → gold)
- Runbook: How to monitor, debug, and re-run failed jobs
- Business usage doc: What tables feed which reports/dashboards/ML features? SLAs?
- Data lineage snippet or dbt docs link

Team/stakeholders ask questions → address blockers → merge → monitor in production first few runs.

Tools That Make This Flow Smoother:

- GitHub/GitLab for PRs + Actions for auto-lint/test
- Pre-commit hooks (Black, SQLFluff, mypy)
- dbt Cloud / Cosmos for Airflow + dbt integration
- Great Expectations / Soda for data contracts in CI

Examples of Poor Code vs. Better Code in Apache Spark (ETL Context)

Example 1: Collecting Large Data to Spark Driver

Poor (Risks driver OOM in production ETL)

```
# Poor: Pulls full result set
bad_rows = df.filter(F.col("quality_score") < 0.5).collect()
print(f"Found {len(bad_rows)} invalid rows")
```

```
# Better: Log metrics distributed + sample for debug
invalid_count = df.filter(F.col("quality_score") < 0.5).count()
print(f"Found {invalid_count:,} invalid rows - check error table")

df.filter(F.col("quality_score") < 0.5) \
  .write.mode("append").saveAsTable("etl_quality_issues")
```

Example 2: Inefficient Join in ETL Transformation

Poor (Full shuffle on large fact table)

```
# Poor: Shuffle join on big data
enriched = fact_df.join(dim_df, "customer_id")
```

Better (Broadcast small dimension)

```
# Better: Map-side join - ideal for ETL enrichment
from pyspark.sql.functions import broadcast
enriched = fact_df.join(broadcast(dim_df), "customer_id")
```

Summary

Code reviews in data engineering ETL projects are your frontline defense against fragile pipelines and costly surprises. By mastering **What** (collaborative vetting), **Why** (quality, scale, knowledge sharing), and especially **How** (staged peer → lead → demo process with ETL-specific checklists), you'll build systems that are robust, observable, and business-aligned.

Implement this tomorrow on your next PR. Start with small changes, automate the boring bits, and celebrate when a review catches a sneaky skew or missing retry.

Code Review | Big Data | Data Engineering | Spark | Data