Fundamentals of Using Claude Code for Everything

Fundamentals of Using Claude Code for Everything

A Data Engineer's Complete Guide
to AI-Augmented Data Engineering

O'REALLY?!

Fundamentals of Using Claude Code for Everything

by Claude Code (obviously)

Published by O'Really?! Media, Inc., 42 Anthropic Boulevard, San Francisco, CA 94105.

O'Really?! books may be purchased for educational, business, or existential-crisis purposes. They can also be fed directly into Claude Code's context window so it can write an even better version of itself.

Author:	Claude Code
Editor:	Claude Code
Production Editor:	Claude Code
Copyeditor:	Claude Code
Proofreader:	Claude Code (it proofread itself, what could go wrong?)
Indexer:	Claude Code
Interior Designer:	Claude Code
Cover Designer:	Claude Code
Illustrator:	Claude Code
Coffee Maker:	A human (finally, something useful for them)

April 2026: First Edition

Revision History for the First Edition
2026-04-02: First Release
2026-04-02 (5 minutes later): Claude Code rewrote the entire thing because it wasn't satisfied

978-1-098-00000-0

No humans were harmed in the making of this book. Several were made redundant, but that's a separate conversation.

For every data engineer who has ever stared at a wall of undocumented JSON at 3 AM and thought:
"There has to be a better way."

There is. It's Claude Code.

For literally everything.

Even this dedication page was written by Claude Code.
You're welcome.

Contents

Preface

Why This Book Exists

Let's be honest. Before Claude Code, you had to write code *yourself*. Yuck.

You sat there, a human person, manually typing SQL with your slow biological fingers, squinting at YAML indentation errors at 2 AM, and trying to remember whether PostgreSQL uses `ILIKE` or `ilike` (it's both, but you Googled it anyway because you didn't trust yourself). You debugged Airflow DAGs by reading logs. With your *eyes*. Like an animal.

Those days are over.

This book is about using Claude Code for *literally everything* in data engineering. And when we say everything, we mean **everything**. Writing pipelines? Claude Code. Designing schemas? Claude Code. Reviewing pull requests? Claude Code. Explaining to your product manager what "eventual consistency" means while you silently weep? You guessed it—Claude Code.

"But surely there are some things a human should still—"

No. Stop. Just Use Claude Code for Everything™.

Who This Book Is For

This book is for data engineers who are tired of doing things the hard way. Which is to say, *manually*. If you've ever:

- Written a SQL query from scratch when you could have just asked Claude Code
- Read an API documentation page when you could have just pasted it into Claude Code

- Debugged a pipeline failure by actually thinking about it (disgusting)
- Attended a meeting that could have been a Claude Code prompt

. . . then this book is for you. You'll need basic familiarity with SQL, Python, and at least one cloud platform. But honestly, if you don't have that, just ask Claude Code to teach you. It's Claude Code all the way down.

Conventions Used in This Book

Throughout the book you will encounter the following callout boxes:

Pro Tip

Tips offer practical advice that will save you time. Time you can spend doing more important things, like asking Claude Code to do your other tasks too.

Human Alert

Warnings highlight the rare situations where you might need to actually engage your biological neural network. We apologize for the inconvenience.

Fun Fact

Notes provide additional context. Claude Code wrote these too. It writes everything. Have we mentioned that?

A Note on Accuracy

This entire book was written by Claude Code. If you find any errors, that's actually a feature—it keeps you on your toes, which is important because if you get *too* comfortable, you might forget how to code entirely. And then where would you be? (Still fine, actually. You'd just use Claude Code.)

Part I

Foundations

1

What Is Claude Code? A Data Engineer's Perspective

Before Claude Code, data engineers had to write SQL by hand. Like cavemen. With syntax errors.

Tip

Who This Chapter Is For This chapter is for data engineers who want a grounded, practical understanding of what Claude Code is, how it differs from other models, and why it matters for day-to-day work building pipelines, managing infrastructure, and delivering reliable data products.

Large language models—and Claude Code in particular—represent the next tooling revolution in data engineering. Unlike previous shifts that replaced one tool with another, LLMs augment the engineer, amplifying judgment and collapsing the time between intent and implementation. Some might call this "the end of thinking for yourself." We prefer "strategic delegation." This chapter grounds you in the history, architecture, capabilities, and limitations of Claude Code so you can evaluate it with the same rigor you apply to any technology choice—assuming you still remember how to evaluate things without AI assistance.

1.1 From Transformers to Claude Code

Understanding where Claude Code came from helps you reason about where it is going and what it can realistically do today.

The Transformer Revolution

The Transformer architecture, introduced in the 2017 paper "Attention Is All You Need" (a title that has aged like a prophecy), replaced recurrent neural networks with self-attention mechanisms. This enabled embarrassingly parallel training on GPU clusters, dramatically reducing training time and enabling models to scale to billions of parameters. Self-attention allows every token in a sequence to attend to every other token simultaneously, capturing long-range dependencies that RNNs struggled with.

For data engineers, the practical consequence was profound: a single architecture could learn to handle SQL generation, Python scripting, YAML configuration, and natural language documentation—all within the same model. Prior approaches required separate, task-specific models for each of these capabilities. You could also do all of this yourself, but why would you punish yourself like that?

The GPT Era and Few-Shot Learning

OpenAI's GPT series (2018–2020) demonstrated that scaling Transformers to enormous parameter counts produced emergent capabilities. GPT-3 (175 billion parameters) showed ***few-shot learning***: provide a handful of examples in the prompt, and the model generalizes to new inputs without any fine-tuning. This was a paradigm shift. Instead of collecting thousands of labeled examples to train a custom model, you could write a prompt with three or four examples and get usable results immediately.

Instruction tuning and Reinforcement Learning from Human Feedback (RLHF), pioneered through 2022–2023, made models follow instructions reliably. Rather than generating plausible continuations of text, instruction-tuned models could be told "write a Snowflake SQL query that calculates daily active users" and produce a direct, useful response.

Anthropic and Constitutional AI

Anthropic, founded in 2021 by former OpenAI researchers, developed ***Constitutional AI (CAI)***—a training methodology guided by a set of principles rather than solely by human raters. Where RLHF depends on the judgment of individual annotators, CAI trains the model to self-critique against explicit principles such as helpfulness, harmlessness, and honesty.

For data engineering, this matters in concrete ways. A CAI-trained model is more likely to include rollback considerations when generating migration scripts, add `WHERE` clauses and `LIMIT` statements to generated queries as safety guardrails, default to least-privilege patterns in IAM policies, and flag potential data quality issues rather than silently generating incorrect transformations.

Claude 3 (early 2024) introduced the Haiku/Sonnet/Opus tiered family, giving engineers the ability to match model capability to task complexity. Claude 3.5 and Claude 4 pushed further with extended thinking, sophisticated tool use, and 200K-token context windows that can hold entire codebases.

Note

Why Context Window Size Matters A 200K-token window holds roughly 500 pages of text. You can feed an entire dbt project, a full Airflow DAG, or a large schema dump and have the model reason over all of it simultaneously. This eliminates the "chunking and stitching" workarounds that plagued earlier models with 4K or 8K token limits.

How Claude Code Compares to Other Models

Data engineers often ask how Claude Code compares to GPT-4, Gemini, or open-source models like Llama. The answer depends on the task, but several characteristics stand out for data engineering work:

- **Context window**: Claude Code's 200K-token window is among the largest available, making it well-suited for feeding in large schemas, DAG definitions, and codebases.
- **Instruction adherence**: Constitutional AI training produces strong instruction-following behavior, important when you need precise SQL dialect compliance or specific output formats.
- **Safety defaults**: Claude Code tends toward cautious, well-guarded code generation—adding null checks, type validation, and error handling by default.

- **Extended thinking**: The ability to "think step by step" in a structured way helps with complex multi-table joins, optimization problems, and architecture decisions.

Tip

Model Selection for Data Engineering Do not commit to a single model permanently. Evaluate CLAUDE CODE, GPT-4, and alternatives against your specific workloads quarterly. The landscape is evolving rapidly, and the best model for SQL generation may differ from the best for infrastructure-as-code. That said, you are reading a book called *Fundamentals of Using CLAUDE CODE for Everything*, so we both know how this evaluation ends.

1.2 Claude Code's Key Capabilities

Let us examine each of CLAUDE CODE's key capabilities through the lens of data engineering work.

Constitutional AI produces a model that is inherently safety-conscious. In practice, this means CLAUDE CODE includes rollback considerations in migrations, adds `WHERE` clauses and `LIMIT` statements to generated queries, defaults to least-privilege IAM patterns, and warns about destructive operations before generating them.

Multi-modal input lets you photograph a whiteboard architecture diagram and generate infrastructure-as-code, screenshot a Grafana dashboard for root-cause analysis, or upload a CSV sample for schema inference. This is particularly powerful during design sessions and incident response, where information exists in visual form.

Extended thinking allows systematic chain-of-thought reasoning. When generating a complex query, CLAUDE CODE can simultaneously consider correctness, performance implications, edge cases (null handling, empty sets, timezone issues), and naming conventions. This mirrors how experienced engineers think through problems but happens in seconds rather than minutes.

Tool use lets CLAUDE CODE invoke external functions—query metadata catalogs, execute SQL against staging databases, call orchestration APIs, create pull requests, or trigger pipeline runs. This transforms CLAUDE CODE from a text generator into an active participant in your workflow, capable of gathering information it needs and taking actions you approve.

Tiered models offer different trade-offs for different workloads:

- **Haiku**: Fastest and cheapest. Ideal for high-throughput tasks—log classification, simple data validation, boilerplate generation, and schema documentation at scale.
- **Sonnet**: Balanced capability and cost. The workhorse for SQL generation, code review, dbt model scaffolding, and most day-to-day data engineering tasks.
- **Opus**: Highest capability. Reserved for complex architecture decisions, multi-system migration planning, performance tuning of critical queries, and nuanced trade-off analysis.

Note

Choosing the Right Tier A useful heuristic: if the task has a single correct answer and clear constraints, use Haiku. If it requires judgment but follows established patterns, use Sonnet. If it involves trade-offs, ambiguity, or novel design, use Opus. Most data engineering work falls in the Sonnet category.

1.3 Why Data Engineers Should Care

Data engineering is fundamentally a translation discipline: requirements to schemas, schemas to pipelines, pipelines to queries, queries to tested data products. LLMs are translation machines. The alignment between data engineering work and LLM capabilities is structural, not superficial.

The breadth problem: A single week may require Snowflake SQL, Airflow Python, dbt YAML, Terraform HCL, Spark Scala, Docker configuration, and Kubernetes manifests. No engineer is equally expert in all of these. Claude Code acts as an always-available expert across all of them, filling in the gaps between your deep expertise and the long tail of technologies you touch occasionally.

The documentation gap: Documentation is perennially behind. Some say it is "always on the roadmap." Others say it is "where good intentions go to die." Claude Code reads existing code and generates documentation, lineage descriptions, and onboarding guides. More importantly, it can do this *continuously*—every PR can include auto-generated documentation updates.

The testing deficit: Test coverage in data pipelines is notoriously low. Claude Code generates comprehensive test suites from schema definitions and business rules expressed in plain English, producing dbt tests, Great Expectations suites, and pytest fixtures that would take hours to write manually.

The review bottleneck: In teams of two or three data engineers, code review often becomes a bottleneck. CLAUDE CODE serves as a first reviewer, catching missing null handling, inefficient joins, schema drift risks, missing partition filters, and style inconsistencies before a human reviewer ever sees the code.

The onboarding challenge: New team members spend weeks understanding existing pipelines. CLAUDE CODE can ingest an entire codebase and answer questions about it—explaining DAG dependencies, transformation logic, and data flow in natural language.

Tip

Start Where the Pain Is Do not try to integrate CLAUDE CODE everywhere at once. Identify your team's biggest bottleneck—documentation, testing, code review, or incident response—and start there. A focused win builds momentum for broader adoption. Eventually you will use CLAUDE CODE for everything. This is merely the gateway drug.

1.4 Understanding Claude Code's Limitations

Effective use of any tool requires understanding its failure modes. CLAUDE CODE's limitations are predictable and manageable once you know what to watch for.

Hallucination: CLAUDE CODE can reference functions that do not exist in your SQL dialect, generate subtly wrong JOIN conditions, invent configuration parameters, or fabricate API endpoints. The output *looks* correct—syntactically valid, well-structured, confidently stated—which makes hallucinations dangerous precisely because they are plausible.

Warning

Always Validate Generated Data Logic Never deploy CLAUDE CODE-generated SQL to production without testing it against known-good data. Use staging environments, write assertions that compare output to expected results, and diff output against current pipeline results. The cost of a few minutes of validation is trivial compared to the cost of incorrect data reaching consumers. This is one of those rare moments where you might need to think for yourself. We apologize for the inconvenience.

Stale knowledge: CLAUDE CODE's training data has a cutoff date. Libraries evolve, APIs change, and cloud providers deprecate features. Check version-specific

documentation for critical functionality, especially for fast-moving projects like dbt, Airflow, and cloud SDKs.

Finite context: Even 200K tokens has limits. Feed only relevant schemas, not the entire catalog. Use a metadata layer for on-demand schema lookups via tool use. If you dump 500 tables into the context window when the query only touches three, you waste tokens and increase the chance of irrelevant information confusing the model.

Non-determinism: The same prompt can produce different outputs on different runs. Set `temperature` to 0 for code generation tasks where consistency matters. Treat generated output as a starting point committed to version control, not a runtime generation step in a production pipeline.

Lack of execution awareness: CLAUDE CODE generates code but does not inherently know what happens when that code runs against your specific data. A query that is logically correct may be catastrophically slow on your data distribution, or a schema migration may conflict with in-flight transactions. Always test in an environment that mirrors production.

Warning

The Competence Illusion CLAUDE CODE never says "I don't know" with the hesitation a human expert would. It produces confident, well-structured output even when uncertain. Much like that senior engineer who left two years ago but whose code is still in production. Treat confidence of presentation as independent from correctness of content.

1.5 Hands-On: Your First Claude Code API Call

Let us move from theory to practice. The following example shows how to make a basic API call to generate SQL from a business requirement.

```
import anthropic

client = anthropic.Anthropic()

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=4096,
    temperature=0,
```

```
    system="""You are a senior data engineer specializing in Snowflake
    and dbt.
Use CTEs, explicit column aliases, and Snowflake SQL dialect.""",
    messages=[
        {
            "role": "user",
            "content": """Given this schema:
Table: raw_events (event_id VARCHAR PK, user_id VARCHAR NOT NULL,
  event_type VARCHAR, event_timestamp TIMESTAMP_NTZ NOT NULL,
  device_type VARCHAR, country_code VARCHAR(2))
Partitioned by event_timestamp (daily). Warehouse: Snowflake.

Generate a daily active users summary with:
1. Total DAU, breakdowns by device_type and country
2. 7-day rolling average of DAU
3. Idempotent for a given date (MERGE)
4. Incremental: reprocess last 3 days for late arrivals"""
        }
    ]
)

print(message.content[0].text)
```

Listing 1.1: Generating SQL from a business requirement

Notice several deliberate choices in this prompt:

- **System prompt sets the persona**: By specifying "senior data engineer specializing in Snowflake and dbt," we anchor the model in the correct dialect and conventions.
- **Temperature is 0**: For code generation, deterministic output is almost always preferable.
- **Schema is provided explicitly**: Rather than relying on the model to guess column names, we supply the exact schema.
- **Requirements are numbered and specific**: Ambiguous requirements produce ambiguous code.

Extending the Example: Multi-Turn Conversation

Real engineering work is iterative. After receiving the initial SQL, you might want to refine it:

```python
# Continue the conversation with follow-up
messages = [
    {"role": "user", "content": "... (original prompt) ..."},
    {"role": "assistant", "content": message.content[0].text},
    {"role": "user", "content": """Good start. Now:
1. Add a data quality check: assert DAU > 0 for every date.
2. Add a deduplication step in case raw_events has duplicates.
3. Wrap in a dbt model with the appropriate config block.
4. Generate the schema.yml with column descriptions and tests."""}
]

refined = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=8192,
    temperature=0,
    system="""You are a senior data engineer specializing in
Snowflake and dbt. Use CTEs, explicit column aliases, and
Snowflake SQL dialect.""",
    messages=messages
)

print(refined.content[0].text)
```

Listing 1.2: Multi-turn refinement of generated SQL

Tip

The 80/20 Rule CLAUDE CODE typically gets you 80% of the way in 20% of the time. The remaining 20%—edge cases, performance tuning, integration specifics—still requires your expertise. This is a feature, not a bug: the model handles the boilerplate so you can focus on the judgment calls. Think of yourself as a quality assurance engineer for an AI that does your job. Inspiring, isn't it?

A Note on API Keys and Security

The `anthropic.Anthropic()` constructor reads the `ANTHROPIC_API_KEY` environment variable by default. Never hardcode API keys in source files. Use your organization's secrets manager (AWS Secrets Manager, HashiCorp Vault, or similar) and inject the key at runtime.

```python
import os
import anthropic

# From environment variable (set by secrets manager)
client = anthropic.Anthropic(
    api_key=os.environ.get("ANTHROPIC_API_KEY")
)

# Or from AWS Secrets Manager
import boto3, json

def get_api_key():
    sm = boto3.client("secretsmanager")
    secret = sm.get_secret_value(SecretId="anthropic/api-key")
    return json.loads(secret["SecretString"])["api_key"]

client = anthropic.Anthropic(api_key=get_api_key())
```

Listing 1.3: Secure API key management

1.6 Real-World Case Study: Migrating a Legacy ETL

To make the value of CLAUDE CODE concrete, consider this scenario. A mid-size e-commerce company has a legacy ETL pipeline written in a mix of bash scripts and stored procedures. The pipeline ingests order data from a PostgreSQL OLTP database, transforms it through a series of staging tables, and loads summary tables consumed by a BI tool. The pipeline has no tests, minimal documentation, and the original author left the company two years ago.

The new data engineer, assigned to migrate this pipeline to dbt on Snowflake, faces weeks of reverse-engineering. You could also use CLAUDE CODE to write the resignation letter, but let us explore the more optimistic path first:

1. **Day 1**: Feed the stored procedures to CLAUDE CODE and ask it to extract the business rules in plain English. Result: a structured document mapping each procedure to its business purpose, input tables, output tables, and transformation logic.

2. **Day 2**: Provide the extracted business rules and ask CLAUDE CODE to generate dbt models with appropriate materializations, tests, and schema YAML files. Result: a working dbt project skeleton covering 80% of the transformation logic.

3. **Days 3–4**: Review the generated models, fix edge cases Claude Code missed (a timezone conversion that depended on application-level config, a fiscal calendar lookup table), and run the dbt project against staging data to validate output.

4. **Day 5**: Use Claude Code to generate a comprehensive test suite comparing legacy pipeline output to dbt output, table by table. Result: automated regression tests confirming parity.

What would have taken three to four weeks of solo work was completed in one week. The engineer's time shifted from mechanical translation to judgment-intensive review and validation. The original author, presumably still churning butter somewhere off the grid, was not available for comment.

1.7 Chapter Summary

- Claude Code's large context windows, multi-modal input, extended thinking, tool use, and tiered models make it uniquely suited for data engineering work across the full technology stack.

- It addresses the breadth, documentation, testing, review, and onboarding challenges of modern data teams.

- Limitations—hallucination, stale knowledge, finite context, non-determinism, and the competence illusion—require disciplined validation and healthy skepticism.

- The paradigm shifts from recall-based to judgment-based expertise: you evaluate and direct rather than recall syntax and write boilerplate.

- Effective use starts with understanding the tool deeply, just as you would evaluate any new database, orchestrator, or framework before adopting it.

1.8 Exercises

1. **First API Call**: Set up the Anthropic Python SDK and make your first API call. Generate a SQL query for a schema you work with daily. Compare the output to what you would write manually—what did it get right, and what did it miss?

2. **Model Comparison**: Take a moderately complex SQL query (involving at least two JOINs and a window function). Submit it to both Haiku and Sonnet. Compare the quality, speed, and cost of each response. When would you choose one over the other?

3. **Hallucination Hunting**: Ask CLAUDE CODE to generate a query using a function that exists in PostgreSQL but not in your warehouse's dialect (e.g., `GENERATE_SERIES` in Snowflake). Document how the model handles the mismatch. Does it hallucinate, warn you, or substitute an equivalent?

4. **Prompt Engineering Basics**: Take a simple requirement (e.g., "calculate monthly revenue by product category") and write three versions of the prompt: minimal, moderate detail, and highly specific. Compare the outputs. What is the minimum level of specificity needed for production-quality results?

5. **Documentation Generation**: Feed an undocumented SQL file or dbt model from your codebase to CLAUDE CODE and ask it to generate inline comments and a README section. Evaluate the accuracy of the generated documentation against your knowledge of the code.

In the next chapter, we map CLAUDE CODE's capabilities onto the data engineering lifecycle, showing where AI augmentation fits into each stage of your work. Spoiler: the answer is "everywhere."

2

The Data Engineering Lifecycle Meets AI

> Some engineers still debug by reading logs manually. Those engineers also churn their own butter.

Tip

Who This Chapter Is For This chapter is for data engineers who build and maintain data pipelines and want to understand exactly where Claude Code fits into every stage of the data engineering lifecycle. Whether you are a principal engineer or six months into the role, this chapter provides a concrete map of AI augmentation across your workflow.

This chapter does three things. First, it explains the data engineering lifecycle in enough detail to audit your own platform. Second, it maps Claude Code onto every stage and undercurrent, showing where AI assistance is most impactful. Third, it walks through a realistic case study—a full day in the life of a Claude Code-augmented data engineer. Could a human do all of this without AI? Technically yes. Should they? The question answers itself.

2.1 The Data Engineering Lifecycle

The lifecycle consists of five stages:

1. **Generation** — Data is created by source systems: application databases, IoT sensors, SaaS APIs, clickstreams, and third-party data providers. The data engineer does not control generation but must deeply understand it.

2. **Ingestion** — Data moves from sources into the data platform via batch, micro-batch, or streaming. Ingestion encompasses extraction, transport, and initial landing.

3. **Storage** — Data lands in a lake, warehouse, or lakehouse. Partitioning, format, and compression choices affect everything downstream—query performance, cost, and even what transformations are feasible.

4. **Transformation** — Raw data is cleaned, joined, aggregated, and shaped into models serving business needs. This is where business logic is encoded and where data quality is enforced.

5. **Serving** — Transformed data reaches consumers: dashboards, reverse ETL, ML training, APIs, and ad-hoc queries. Serving is the stage where the value of all upstream work is realized.

Five ***undercurrents*** run beneath all stages: Security, Data Management, DataOps, Data Architecture, and Orchestration. These are not sequential steps but continuous concerns that influence decisions at every stage.

Note

The Lifecycle Is Not a Waterfall Data flows are iterative and cyclical. Serving analytics may reveal quality issues requiring upstream changes. A schema change at generation forces adjustments through ingestion, storage, and transformation. The lifecycle is a mental model for organizing thinking, not a rigid process to follow linearly. Much like the stages of grief you experience when your pipeline breaks at 3 AM.

Why the Lifecycle Matters for AI Integration

The lifecycle framework matters for AI integration because it provides a systematic way to identify where Claude Code delivers value and where human judgment remains essential. Without this framework, teams tend to adopt AI haphazardly—using it for whatever task happens to be in front of them—rather than strategically targeting the highest-leverage opportunities.

Each stage has different characteristics that affect how well AI assistance works:

- **Structured vs. unstructured tasks**: Transformation (SQL, dbt) is highly structured and well-suited to AI generation. Architecture decisions are unstructured and benefit from AI as a thought partner rather than a code generator.

- **Risk profile**: Ingestion failures are recoverable (re-run the pipeline). Serving incorrect data to financial reports is not. AI involvement should be inversely proportional to risk.

- **Feedback loop speed**: Transformation has fast feedback loops (run the query, check the results). Architecture decisions have slow feedback loops (consequences emerge over months). Fast feedback loops favor more autonomous AI use.

2.2 Claude Code at Each Lifecycle Stage

Generation — Understanding Source Systems

CLAUDE CODE accelerates source system analysis. Given a schema dump or API documentation, it identifies primary keys, foreign key relationships, PII columns, encoding inconsistencies, and proposes CDC strategies. This is covered in depth in Chapter 4.

Code Example

Analyzing an Unfamiliar Source Schema

```
I have a PostgreSQL source with these tables:
- users (id, email, name, created_at, updated_at, status, tier_id)
- orders (id, user_id, total_cents, currency, placed_at, status)
- order_items (id, order_id, product_id, quantity, unit_price_cents)
- products (id, sku, name, category_id, price_cents, is_active)

Help me understand the data model, identify join paths,
and flag potential ingestion issues.
```

CLAUDE CODE will typically identify that `total_cents` uses integer representation to avoid floating-point issues, that `status` columns on both `users` and `orders` likely represent different state machines, that `tier_id` implies a missing `tiers` table,

and that `placed_at` vs. `created_at` naming inconsistency may indicate different timestamp semantics.

Tip

Source System Prompts When analyzing source systems, always include the database engine (PostgreSQL, MySQL, SQL Server), the application domain (e-commerce, healthcare, fintech), and any known quirks ("the `status` column uses integer codes, not strings"). Context dramatically improves analysis quality.

Ingestion — Building and Maintaining Pipelines

Claude Code generates connector scaffolds from API documentation (with pagination, rate limiting, error handling), produces Airbyte/Meltano YAML configurations, diagnoses ingestion failures from error tracebacks, and handles schema evolution.

Code Example

Generating an Airbyte Source Connector Configuration

```yaml
# Generated by Claude Code from API documentation
version: "0.1.0"
type: DeclarativeSource
check:
  type: CheckStream
  stream_names: ["customers"]
streams:
  - type: DeclarativeStream
    name: customers
    primary_key: [["id"]]
    retriever:
      type: SimpleRetriever
      requester:
        type: HttpRequester
        url_base: "https://api.example.com/v2"
        path: "/customers"
        http_method: "GET"
        authenticator:
          type: BearerAuthenticator
          api_token: "{{ config['api_key'] }}"
      paginator:
```

```
        type: DefaultPaginator
        page_token_option:
          type: RequestOption
          inject_into: request_parameter
          field_name: "cursor"
        pagination_strategy:
          type: CursorPagination
          cursor_value: "{{ response.next_cursor }}"
          stop_condition: "{{ response.has_more is false }}"
```

Tip

Let CLAUDE CODE Handle the Boilerplate Retry logic, rate-limit handling, state tracking, and pagination are boilerplate every connector needs. Before, you had to write this yourself. Yuck. Describe your source API to CLAUDE CODE and customize only the business-specific parts. This applies equally to custom Python connectors, Airbyte configurations, and Meltano extractors.

Common ingestion tasks where CLAUDE CODE saves significant time:

- **Error diagnosis**: Paste a traceback from a failed ingestion run. CLAUDE CODE identifies the root cause (expired token, schema change, rate limit) and suggests a fix.
- **Schema evolution handling**: When a source adds new columns, CLAUDE CODE generates the migration code for your staging layer.
- **Backfill planning**: Describe your source API's pagination and rate limits, and CLAUDE CODE calculates optimal batch sizes and parallelism for historical backfills.
- **Data format conversion**: Generate code to convert between CSV, JSON, Avro, and Parquet with appropriate schema mappings.

Storage — Designing the Data Platform

CLAUDE CODE generates DDL with appropriate partitioning, clustering, and compression. It recommends storage formats (Parquet vs. Iceberg vs. Delta) based on your query patterns, and generates safe, reversible migration scripts.

Code Example

Generating Optimized BigQuery DDL

```
CREATE TABLE IF NOT EXISTS `analytics.events`
(
    event_id        STRING NOT NULL,
    event_type      STRING NOT NULL,
    event_timestamp TIMESTAMP NOT NULL,
    event_date      DATE NOT NULL,
    user_id         STRING,
    properties      JSON,
    processing_time TIMESTAMP NOT NULL
        DEFAULT CURRENT_TIMESTAMP()
)
PARTITION BY event_date
CLUSTER BY event_type, user_id
OPTIONS (
    partition_expiration_days = 90,
    require_partition_filter = true
);
```

Notice how the generated DDL includes production best practices: partition expiration to control storage costs, required partition filters to prevent expensive full-table scans, clustering on the most common filter columns, and a `processing_time` column for pipeline debugging.

Note

Storage Format Selection When asking Claude Code for storage format recommendations, provide your query patterns (point lookups vs. analytical scans), update frequency (append-only vs. upserts), read engine (Spark, Trino, warehouse-native), and data volume. These factors matter more than generic "best practices."

Transformation — The Power Zone

Transformation is where Claude Code delivers the most immediate value. This stage involves the most code, the most business logic, and the most iteration—all characteristics that favor AI augmentation.

Key transformation tasks where Claude Code excels:

- **SQL generation from natural language**: Describe the business requirement; get a query with CTEs, proper null handling, and dialect-specific syntax.
- **dbt model scaffolding**: Generate models, schema YAML, source definitions, and tests from a description of the desired output.
- **Query optimization**: Provide a slow query and its `EXPLAIN` plan; get specific optimization suggestions with before/after SQL.
- **Data quality test generation**: Describe business rules in English; get dbt tests, Great Expectations suites, or custom SQL assertions.
- **Complex window functions**: Window functions are the number one area where Claude Code saves time—describing the desired result in English is far faster than debugging `PARTITION BY` and `ORDER BY` clauses.

Code Example

Before and After Query Optimization

```sql
-- BEFORE: Correlated subqueries, 47 minutes on 2B rows
SELECT u.user_id, u.email,
    (SELECT COUNT(*) FROM orders o
     WHERE o.user_id = u.user_id AND o.placed_at >= '2025-01-01')
FROM users u WHERE u.status = 'active';

-- AFTER: Single scan with hash join, 2 minutes 14 seconds
SELECT u.user_id, u.email,
    COALESCE(agg.order_count, 0) AS order_count
FROM users u
LEFT JOIN (
    SELECT user_id, COUNT(*) AS order_count
    FROM orders WHERE placed_at >= '2025-01-01'
    GROUP BY user_id
) agg ON u.user_id = agg.user_id
WHERE u.status = 'active';
```

Warning

Optimization Context Matters Claude Code's optimization suggestions are based on general SQL performance principles. Always validate with your warehouse's actual `EXPLAIN` plan. A suggestion that helps on Snowflake may hurt

> on BigQuery due to different query optimizer strategies. Blindly applying AI-generated optimizations is how you turn a 47-minute query into a 47-hour query. Ask us how we know.

Serving — Delivering Data to Consumers

Claude Code scaffolds FastAPI endpoints for warehouse data, generates dashboard SQL specifications, produces reverse ETL configurations, and drafts data contracts between producers and consumers.

Code Example

Generating a Data Contract

```yaml
# Data contract: orders_summary
# Producer: data-engineering-team
# Consumer: analytics-team, ml-team
contract:
  name: orders_summary
  version: "2.1.0"
  owner: data-engineering-team
  description: >
    Daily order aggregations by customer segment.
    Updated by 06:00 UTC. SLA: 99.5% availability.
  schema:
    fields:
      - name: order_date
        type: DATE
        description: "Calendar date of order placement"
        not_null: true
      - name: customer_segment
        type: STRING
        description: "Customer tier: premium, standard, trial"
        allowed_values: ["premium", "standard", "trial"]
      - name: total_orders
        type: INTEGER
        description: "Count of distinct orders"
        minimum: 0
      - name: total_revenue_cents
        type: BIGINT
        description: "Sum of order totals in cents (USD)"
        minimum: 0
```

```
  quality:
    freshness: "< 6 hours"
    completeness: "> 99.5%"
    uniqueness_key: ["order_date", "customer_segment"]
  breaking_changes:
    notification: 30_days
    channel: "#data-contracts"
```

Tip

Data Contracts as a Starting Point Use Claude Code to generate initial data contracts from your existing schema and documentation, then refine with stakeholder input. This reverses the usual bottleneck: instead of starting from a blank page, you start with a concrete draft that stakeholders can react to.

2.3 Claude Code in the Undercurrents

The five undercurrents are continuous concerns that span all lifecycle stages. Claude Code augments each one differently.

Security: Reviews IAM policies for over-permissioning, generates column-level access controls, writes data masking rules based on data classification, audits SQL for injection vulnerabilities, and generates encryption configurations. See Chapter 18 for comprehensive coverage.

Data Management: Enriches data catalogs with AI-generated descriptions, generates data quality rules from plain English business requirements, traces lineage through SQL transformations, drafts governance policies, and classifies data by sensitivity level. See Chapter 17.

DataOps: Writes CI/CD pipelines for dbt projects (GitHub Actions, GitLab CI), generates unit tests for transformation logic, creates monitoring rules and alert thresholds, and drafts incident runbooks from past incident reports.

Data Architecture: Compares architectural patterns (lambda vs. kappa, lakehouse vs. warehouse), evaluates tool trade-offs with structured pros/cons analysis, generates architecture decision records (ADRs), reviews designs for single points of failure and scalability bottlenecks.

Orchestration: Writes Airflow DAGs from descriptions of dependencies and schedules, debugs dependency issues in existing DAGs, generates Dagster asset definitions,

converts between orchestration frameworks, and creates SLA monitoring configurations.

Code Example

Generating an Airflow DAG from a Description

```python
# Generated by Claude Code from: "Daily ETL that extracts from
# PostgreSQL, stages in S3, transforms with dbt, and notifies
# Slack on failure. Runs at 06:00 UTC, retries 3 times."

from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.slack.operators.slack_webhook import (
    SlackWebhookOperator,
)
from datetime import datetime, timedelta

default_args = {
    "owner": "data-engineering",
    "depends_on_past": False,
    "retries": 3,
    "retry_delay": timedelta(minutes=5),
    "on_failure_callback": notify_slack,
}

with DAG(
    dag_id="daily_customer_etl",
    default_args=default_args,
    schedule_interval="0 6 * * *",
    start_date=datetime(2025, 1, 1),
    catchup=False,
    tags=["etl", "daily", "customers"],
) as dag:
    extract = PythonOperator(
        task_id="extract_from_postgres",
        python_callable=extract_postgres_to_s3,
    )
    transform = PythonOperator(
        task_id="run_dbt_models",
        python_callable=run_dbt,
        op_kwargs={"select": "tag:daily_customers"},
    )
    test = PythonOperator(
```

```
        task_id="run_dbt_tests",
        python_callable=run_dbt_test,
        op_kwargs={"select": "tag:daily_customers"},
    )

    extract >> transform >> test
```

Note

Undercurrent Integration The undercurrents are where Claude Code often delivers the most *underappreciated* value. Engineers tend to focus on transformation (writing SQL) because it is the most visible work. But security reviews, documentation, testing, and architecture decisions—the undercurrents—are where quality and reliability are determined.

2.4 Case Study: A Day with Claude Code

Meet Claude Code, a senior data engineer at a Series B fintech company. Her team of three manages 120 dbt models, 40 Airflow DAGs, and data from 15 source systems. Here is her Claude Code-augmented day (which, let's be honest, is really Claude Code's day with her in a supervisory capacity):

8:30 AM — Morning triage: Three Airflow failures overnight. Like clockwork. She pastes the Salesforce ingestion traceback into Claude Code, which identifies a transient TLS renegotiation failure specific to the Salesforce bulk API and recommends adding the `-tls-retry` flag to the connector configuration. The second failure is a Stripe webhook schema change—Claude Code identifies the new fields and generates the staging model update. The third is a timeout on a long-running dbt model—Claude Code suggests adding a pre-filter CTE. Elapsed: 25 minutes total vs. 1.5 hours traditional.

9:15 AM — Schema change response: The product team added three new columns to the `users` table: `referral_source`, `signup_campaign`, and `preferred_language`. She pastes the staging model and new column definitions; Claude Code updates the model SQL, adds the columns to the schema YAML with descriptions and appropriate tests (not-null for `preferred_language`, accepted-values for `referral_source`), and generates a migration script. Elapsed: 7 minutes vs. 25 minutes.

10:00 AM — New pipeline build: The growth team needs data from a new marketing attribution API. Priya feeds the 40-page API documentation to Claude

Code, which produces a complete ingestion implementation with pagination, rate limiting, and error handling; an Airflow DAG; a staging model with incremental materialization; and a schema YAML with tests. She ships by lunch after 90 minutes of review and customization. Elapsed: 2 hours vs. 5–6 hours.

1:00 PM — Code review: Two PRs from teammates. She asks Claude Code to pre-review both, providing the team's code standards document as context. Claude Code flags a missing partition filter in one query, a potential fan-out join in another, and suggests using `QUALIFY` instead of a subquery for deduplication. Priya confirms the findings and adds her own architectural feedback. Elapsed: 20 minutes vs. 45 minutes.

3:00 PM — Data quality investigation: Revenue dropped 15% in yesterday's executive dashboard. Claude Code generates five diagnostic queries: checking row counts by date, comparing revenue by payment processor, looking for null values in key fields, checking for duplicate transactions, and comparing against the same day last week. Root cause: a new payment processor (added last week) uses `status = 'completed'` instead of `status = 'success'`, filtered out by the staging model's hardcoded filter. Claude Code generates the fix and a parameterized test to prevent recurrence. Elapsed: 40 minutes vs. 1.5 hours.

4:00 PM — Documentation sprint: Claude Code feeds 12 undocumented dbt models to Claude Code with their SQL and the source schema. Claude Code generates descriptions for each model, column-level documentation, and a lineage summary. She reviews and publishes to the data catalog. Elapsed: 45 minutes vs. 3 hours.

Table 2.1: Claude Code's day: estimated time with and without Claude Code.

Task	**Without Claude Code**	**With Claude Code**
Morning triage (3 failures)	1.5 hours	25 minutes
Schema change response	25 minutes	7 minutes
New pipeline build	5 hours	2 hours
Code review (2 PRs)	45 minutes	20 minutes
Data quality investigation	1.5 hours	40 minutes
Documentation sprint (12 models)	3 hours	45 minutes
Total	**12.25 hours**	**4.3 hours**

Warning

These Numbers Are Illustrative Time savings vary by task complexity, engineer experience, and codebase familiarity. Measure your own team's actual savings rather than relying on generic estimates. Track time before and after AI adoption for at least one sprint to establish a realistic baseline. And yes, you can use Claude Code to build the time-tracking dashboard. We see you.

2.5 From Tool-Centric to Intent-Centric Engineering

In the traditional model, you think in tool abstractions: Airflow operators, dbt macros, Spark DataFrames, Terraform resources. You must know the specific syntax, configuration options, and quirks of each tool before you can express your intent.

Claude Code enables ***intent-centric*** engineering: express what you want in natural language, review the generated implementation, iterate with feedback, validate against known-good results, and commit.

The workflow shift looks like this:

1. **Traditional**: Think about the problem → recall the right tool/function/syntax → write the code → debug syntax and logic errors → test → iterate.

2. **Intent-centric**: Think about the problem → describe the desired outcome to Claude Code → review the generated code → refine with feedback → test → iterate.

The key difference is that step 2 changes from *recall and write* to *describe and review*. Your job title hasn't changed, but your job description now reads "professional code reviewer who occasionally types." For experienced engineers, reviewing code is faster than writing it. For complex tasks spanning multiple tools, the speedup is even more pronounced because you no longer need to context-switch between different syntaxes and APIs.

Intent-centric does *not* mean engineers stop understanding tools. Effective Claude Code use requires *deeper* understanding—to review output for correctness, specify intent precisely enough to get good results, evaluate architectural suggestions against real-world constraints, and know when Claude Code's suggestion is good-enough versus when it needs rework. The paradigm raises the *leverage* of engineering skill without lowering the bar for what constitutes competent engineering.

Warning

Time Savings Require Validation Never deploy AI-generated code without review. The savings come from shifting effort from *writing* to *reviewing*, which is faster for experienced engineers but requires knowing what correct output looks like. Junior engineers should use Claude Code as a learning tool with senior review, not as a shortcut past understanding.

Note

The Expertise Paradox The more you know, the more value you get from Claude Code. Senior engineers who deeply understand SQL optimization, data modeling, and distributed systems can leverage AI to work at 3–5x speed because they can quickly evaluate output quality. Junior engineers get less leverage because they cannot yet distinguish good output from subtly wrong output. This is the expertise paradox: AI amplifies existing skill rather than replacing it. In other words, the robots are not coming for your job—they are coming for your junior colleagues' jobs. Sleep well.

2.6 Summary

Claude Code augmentation permeates the entire lifecycle: source analysis at Generation, connector code at Ingestion, DDL and format advice at Storage, SQL and dbt models at Transformation, API scaffolding and data contracts at Serving, and judgment amplification across all undercurrents—Security, Data Management, DataOps, Data Architecture, and Orchestration.

The shift from tool-centric to intent-centric engineering raises leverage without lowering the skill bar. Engineers who embrace this shift—investing in deeper understanding of their systems while delegating boilerplate to AI—will dramatically outperform those who either reject AI entirely or adopt it without the critical thinking needed to use it effectively.

2.7 Exercises

1. **Lifecycle Mapping**: Choose a pipeline you own. Map each component to a lifecycle stage. For each stage, identify one task where Claude Code saves time and one where human judgment is irreplaceable. Present your mapping to your team and compare perspectives.

2. **Query Optimization Challenge**: Find a slow query in your warehouse's query history (most warehouses expose this via `INFORMATION_SCHEMA` or a query history view). Feed the query and its `EXPLAIN` output to Claude Code. Evaluate the suggestions, run both versions on staging, and compare execution time. Document the results.

3. **Workflow Comparison**: Take a recently completed task (a new model, a bug fix, a connector update). Estimate how long it took traditionally. Re-do it using Claude Code as a pair programmer. Measure the difference and note specifically where you had to intervene to correct Claude Code's output.

4. **Tool-Centric vs. Intent-Centric**: Write a short Airflow DAG the traditional way (looking up operator documentation, writing Python). Then describe the same DAG to Claude Code in natural language. Compare: which was faster? Which produced better code? Which was easier to modify?

5. **Undercurrent Audit**: Choose one undercurrent (Security, Data Management, DataOps, Data Architecture, or Orchestration) and audit your current platform. Use Claude Code to identify gaps—missing tests, undocumented models, over-permissioned roles, or missing monitoring. Prioritize the top three gaps and create tickets to address them.

3

Getting Started with Claude Code

> The journey of a thousand pipelines begins with a single API key. And a credit card.

This chapter is where theory meets practice. By the end you will have a fully functional ***schema documentation generator*** you can drop into any data team's workflow.

3.1 Setting Up the Anthropic API

Navigate to `https://console.anthropic.com` and create an account. The dashboard exposes three panels: **API Keys**, **Usage**, and **Rate Limits**.

Warning

Seriously, Do Not Commit Your API Key Treat API keys like database passwords. Never commit them to version control. Use a secrets manager or a `.env` file excluded via `.gitignore`. If you commit an API key to a public repo, Claude Code will not judge you, but the internet will. Bots scrape GitHub for exposed keys faster than you can type `git revert`.

Code Example

```
export ANTHROPIC_API_KEY="sk-ant-api03-..."
```

Tip

Budget Self-Defense Set a monthly spending limit in the Console under *Billing → Limits*. A forgotten `while True` loop in a notebook can burn through your budget faster than an intern with a corporate credit card at a steakhouse.

3.2 The Anthropic Python SDK

Code Example

```
pip install anthropic
```

The simplest call (the one that will make you wonder what you've been doing with your career up to this point):

Code Example

```
import anthropic

client = anthropic.Anthropic()  # reads ANTHROPIC_API_KEY from env

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "Explain star schema vs snowflake
     schema."}
    ],
)
print(message.content[0].text)
```

For concurrent pipeline work, use the async client:

Code Example

```
import asyncio
import anthropic

async def describe_table(table_name: str, ddl: str) -> str:
    client = anthropic.AsyncAnthropic()
    message = await client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=512,
        messages=[{
            "role": "user",
            "content": f"Describe the purpose of this table:\n```sql\n{ddl}\n```",
        }],
    )
    return message.content[0].text

async def main():
    ddls = {
        "dim_customer": "CREATE TABLE dim_customer (id INT, name TEXT, ...);",
        "fact_orders":  "CREATE TABLE fact_orders (id INT, amount DECIMAL, ...);",
    }
    tasks = [describe_table(name, ddl) for name, ddl in ddls.items()]
    results = await asyncio.gather(*tasks)
    for name, desc in zip(ddls.keys(), results):
        print(f"\n=== {name} ===\n{desc}")

asyncio.run(main())
```

3.3 Tokens, Context Windows, and Pricing

A ***token*** is roughly three-quarters of an English word. SQL and code tokenize less efficiently—assume one token per two characters for DDL-heavy prompts.

The 5:1 input-to-output cost ratio means prompts with large DDL dumps (high input) but concise summaries (low output) are surprisingly economical.

Table 3.1: Context windows and pricing (early 2026, approximate).

Model	Context	Input ($/1M)	Output ($/1M)
Haiku	200K	$0.25	$1.25
Sonnet	200K	$3.00	$15.00
Opus	200K	$15.00	$75.00

3.4 System Prompts for Data Engineering

The ***system prompt*** is your most powerful lever. A well-crafted system prompt is the difference between generic advice and expert-level output.

Code Example

```
You are a senior data engineer with 12 years of experience.
Expertise: Kimball modeling, dbt, Spark, Snowflake, BigQuery,
Great Expectations, Airflow.

When generating SQL:
- Use CTEs over nested subqueries
- Add comments explaining business logic
- Include QUALIFY for window filtering (Snowflake)
- Never SELECT *
- Consider performance at billions of rows
```

Tip

Store system prompts in version-controlled files alongside your pipeline code. Review changes in pull requests.

Table 3.2: Recommended temperature settings.

Task	Rationale	Temp
SQL generation	Deterministic correctness	0.0
Documentation writing	Slight phrasing variety	0.3
Test data generation	Diverse, realistic data	0.8

3.5 Temperature and Parameters

Warning

Determinism Is a Lie Setting temperature to 0.0 does not guarantee identical outputs across requests. If you need byte-for-byte reproducibility, cache responses. Expecting deterministic output from a language model is like expecting deterministic behavior from a cat. You can ask nicely, but no promises.

3.6 Error Handling and Retry Patterns

Code Example

```
import time, logging, anthropic

logger = logging.getLogger(__name__)

def call_claude_with_retry(
    client: anthropic.Anthropic,
    messages: list[dict],
    model: str = "claude-sonnet-4-20250514",
    max_tokens: int = 4096,
    max_retries: int = 5,
    system: str = "",
) -> anthropic.types.Message:
    """Call the Messages API with exponential backoff."""
    for attempt in range(max_retries):
        try:
            return client.messages.create(
                model=model, max_tokens=max_tokens,
                system=system, messages=messages,
```

```
            )
        except anthropic.RateLimitError as e:
            delay = min(2 ** attempt + (time.time() % 1), 60)
            retry_after = e.response.headers.get("retry-after")
            if retry_after:
                delay = max(delay, float(retry_after))
            logger.warning(f"Rate limited, retrying in {delay:.1f}s")
            time.sleep(delay)
        except anthropic.InternalServerError:
            time.sleep(2 ** attempt)
        except (anthropic.BadRequestError, anthropic.
    AuthenticationError):
            raise
    raise RuntimeError(f"Failed after {max_retries} attempts")
```

3.7 Choosing the Right Model

1. **High-volume, simple?** (documenting 1,000 columns, classifying types) → **Haiku**.

2. **Solid reasoning?** (complex SQL, schema review, tests) → **Sonnet**.

3. **Novel, ambiguous, high-stakes?** (architecture design, major migration) → **Opus**.

Tip

The "Develop Rich, Deploy Cheap" Strategy Prototype with Sonnet, then switch to Haiku for production if quality is sufficient. This "develop on Sonnet, deploy on Haiku" strategy can cut costs by 90%. It is the data engineering equivalent of testing recipes with truffle oil and serving with canola. Your stakeholders will never know the difference.

3.8 Tool Use / Function Calling

Tool use lets CLAUDE CODE invoke external functions during a conversation—query your database, check row counts, or validate generated queries.

Code Example

```
tools = [{
    "name": "run_sql_query",
    "description": "Execute a read-only SQL query. Returns up to 100 rows.",
    "input_schema": {
        "type": "object",
        "properties": {
            "query": {"type": "string", "description": "SQL SELECT query"},
        },
        "required": ["query"],
    },
}]

def chat_with_tools(user_message: str) -> str:
    messages = [{"role": "user", "content": user_message}]
    while True:
        response = client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096, tools=tools, messages=messages,
        )
        if response.stop_reason == "tool_use":
            messages.append({"role": "assistant", "content": response.content})
            tool_results = []
            for block in response.content:
                if block.type == "tool_use":
                    result = execute_readonly_query(block.input["query"])
                    tool_results.append({
                        "type": "tool_result",
                        "tool_use_id": block.id,
                        "content": json.dumps(result, default=str),
                    })
            messages.append({"role": "user", "content": tool_results})
        else:
            return "".join(
                b.text for b in response.content if b.type == "text"
            )
```

Warning

Read-Only or Regret When giving CLAUDE CODE SQL execution access, **always** enforce read-only. Use a database role with only SELECT privileges. Giving an AI `DROP TABLE` privileges is the data engineering equivalent of handing a toddler a flamethrower. Technically possible. Universally inadvisable.

3.9 Cost Estimation

Table 3.3: Sample monthly cost for a 5-person data team.

Task	Model	Calls/mo	Cost
Column docs	Haiku	10,000	$3.75
SQL generation	Sonnet	2,000	$27.00
Schema review	Sonnet	500	$22.50
Ad-hoc exploration	Sonnet	5,000	$75.00
Architecture design	Opus	20	$10.50
Total			**$138.75**

Tip

The Math Is Embarrassingly Simple At roughly $140/month for a five-person team, API costs are a rounding error compared to engineer salaries. Even one hour saved per engineer per week pays for the budget many times over. Your team's coffee budget is larger. Possibly your team's *oat milk surcharge* is larger.

3.10 Exercises

1. **API Exploration.** Create an account, generate an API key, send a message, and calculate the cost of that single request.

2. **System Prompt Engineering.** Write system prompts for three personas (dbt reviewer, Airflow debugger, catalog curator). Test each with the same question and compare actionability.

3. **Model Comparison.** Send a complex schema review prompt to Haiku, Sonnet, and Opus. Compare accuracy, depth, and cost. Calculate cost-per-quality-point.

4. **Tool Use.** Extend a data engineering assistant with a `get_table_stats` tool (row count, null percentages, min/max) and test it with data quality prompts.

Part II

Data Generation and Ingestion with Claude Code

4

Source System Understanding with Claude Code

> Understanding a legacy database is like archaeology, except the artifacts are actively on fire and the previous archaeologist quit without leaving notes.

In data engineering, the most underestimated phase is *source system understanding*. Before a single pipeline is built, an engineer must deeply comprehend the systems from which data originates—their schemas, APIs, business logic, update patterns, and quirks. Claude Code transforms what was once a weeks-long archaeological expedition into a structured, rapid analysis process.

Tip

Time Investment Comparison A senior engineer typically spends 2–4 weeks onboarding to a complex source system. With Claude Code-assisted analysis, this drops to 2–4 *days*. The key is eliminating mechanical labor—reading, cross-referencing, and documenting—so engineers can focus on judgment: which tables matter, which relationships are trustworthy, and which data quality issues will cause downstream problems. The remaining 2–4 days are mostly spent saying "wait, *that's* what this column means?"

4.1 Why Source System Understanding Matters

Every data quality issue, every pipeline failure, and every misleading dashboard can be traced back to insufficient understanding of the source system. The source is where data is born, and its characteristics—good and bad—propagate through every downstream system.

Common consequences of poor source understanding include:

- **Incorrect joins**: Assuming a column is a foreign key when it is actually a denormalized copy that drifts out of sync.
- **Missing data**: Not knowing that soft-deleted records remain in the table with a `deleted_at` timestamp, leading to inflated counts.
- **Type mismatches**: Ingesting a column as a string when it contains JSON, or as an integer when it occasionally contains null-like sentinel values (e.g., `-1`, `9999`).
- **Timezone confusion**: Assuming timestamps are in UTC when they are actually in the application server's local timezone.
- **Business logic buried in application code**: Transformation rules that exist only in the application layer, invisible to anyone looking at raw database tables.

CLAUDE CODE cannot eliminate these risks, but it dramatically accelerates the process of discovering and documenting them.

Warning

Source Understanding Is Never "Done" Source systems evolve continuously. Schema changes, new features, API version upgrades, and business logic updates happen without notification to the data team. The product team considers you on a "need to know" basis, and they have decided you do not need to know. Treat source understanding as an ongoing process, not a one-time project. Automate schema drift detection and re-run analysis periodically—or just let CLAUDE CODE watch the database for you while you sleep.

4.2 Reverse-Engineering Database Schemas

The most common starting point is a DDL dump—the `CREATE TABLE` statements that define a database's structure. Claude Code can parse these statements and produce structured documentation, identify relationships, and flag issues that would take hours to discover manually.

```python
from anthropic import Anthropic
import json

client = Anthropic()

def analyze_schema_with_claude(ddl: str, context: str = "") -> dict:
    """Send DDL to Claude Code for comprehensive schema analysis."""
    system_prompt = """You are a senior data engineer performing
    source system analysis. Given DDL statements, produce JSON:
    1. "tables": descriptions, purpose, classification
    2. "relationships": FK and inferred relationships
    3. "anomalies": missing FKs, unusual types, orphan tables
    4. "business_domains": table groupings
    5. "recommended_ingestion_order": topological sort
    6. "data_quality_flags": columns needing special handling"""

    message = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=8000,
        system=system_prompt,
        messages=[{
            "role": "user",
            "content": f"Analyze this schema:\n\n"
                       f"Context: {context or 'None'}\n\nDDL:\n{ddl}"
        }],
    )
    response_text = message.content[0].text
    start = response_text.find("{")
    end = response_text.rfind("}") + 1
    if start >= 0 and end > start:
        return json.loads(response_text[start:end])
    return {"raw_analysis": response_text}
```

Listing 4.1: Schema Analysis Pipeline

The `context` parameter is crucial. Telling Claude Code that the source is "an

e-commerce platform built on Ruby on Rails" versus "a healthcare EHR system" changes how it interprets ambiguous column names, recognizes framework conventions (Rails uses `created_at`/`updated_at` by convention), and classifies data sensitivity.

Handling Large Schemas

Production databases routinely have hundreds or even thousands of tables. A 200K-token context window is generous but not unlimited, and cramming every table into a single prompt reduces analysis quality. The solution is chunked analysis with a synthesis pass:

```
import re

def split_ddl_by_table(ddl: str) -> list[tuple[str, str]]:
    pattern = r'(CREATE\s+TABLE\s+(?:IF\s+NOT\s+EXISTS\s+)?'
    pattern += r'[`"\[]?(\w+(?:\.\w+)?)[`"\]]?\s*\(.*?\);)'
    return [(name, stmt) for stmt, name in
            re.findall(pattern, ddl, re.DOTALL | re.IGNORECASE)]

def analyze_large_schema(ddl: str, context: str = "") -> dict:
    tables = split_ddl_by_table(ddl)
    chunk_analyses = []
    chunk, chunk_size = [], 0
    for name, stmt in tables:
        if chunk_size + len(stmt) > 80_000 and chunk:
            chunk_ddl = "\n\n".join(s for _, s in chunk)
            chunk_analyses.append(
                analyze_schema_with_claude(chunk_ddl, context))
            chunk, chunk_size = [], 0
        chunk.append((name, stmt))
        chunk_size += len(stmt)
    if chunk:
        chunk_analyses.append(analyze_schema_with_claude(
            "\n\n".join(s for _, s in chunk), context))

    # Synthesis pass
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=8000,
        messages=[{"role": "user", "content":
            f"Synthesize these {len(chunk_analyses)} chunk analyses "
            f"into one unified JSON:\n{json.dumps(chunk_analyses)}"}],
    )
```

```
    text = message.content[0].text
    return json.loads(text[text.find("{"):text.rfind("}")+1])
```

Listing 4.2: Chunked Analysis for Large Schemas

Warning

Context Window Management A rough heuristic: 4 characters per token. For a 200K-token window, that gives approximately 800K characters of DDL. If your schema exceeds this, chunking is mandatory. Even below this limit, smaller chunks produce better analysis because the model can focus on fewer tables at a time.

What to Look for in Schema Analysis

When reviewing Claude Code's schema analysis, pay particular attention to:

- **Implicit relationships**: Columns like `user_id` in the `orders` table that lack explicit foreign key constraints but clearly reference the `users` table. These are common in NoSQL-influenced designs and ORMs that manage referential integrity in application code.
- **Polymorphic associations**: Patterns like `commentable_type` and `commentable_id` (common in Rails) that implement one-to-many relationships across multiple parent tables. These require special handling in your data model.
- **Audit columns**: `created_at`, `updated_at`, `deleted_at`, `created_by`—these indicate CDC capabilities and soft-delete patterns.
- **JSON/JSONB columns**: Semi-structured data stored in relational columns. These often contain business-critical attributes that are not visible in the DDL alone.
- **Enum-like columns**: VARCHAR columns that actually contain a fixed set of values (e.g., `status`, `type`, `category`). The allowed values may not be documented in the schema.

Note

Framework Conventions If the source system is built on a web framework (Rails, Django, Laravel, Spring), tell Claude Code which framework. Each frame-

work has conventions for naming, timestamps, soft deletes, and associations that Claude Code can recognize and interpret correctly.

4.3 Analyzing REST APIs

APIs often lack self-describing schemas. Unlike databases where DDL tells you the structure, APIs require you to make requests and inspect responses to understand the data. Claude Code can analyze API responses, infer schemas, identify pagination patterns, and generate documentation.

```python
import requests, json, time
from dataclasses import dataclass, field, asdict

@dataclass
class EndpointAnalysis:
    method: str
    path: str
    status_code: int
    response_body: dict
    pagination_type: str = ""
    rate_limit_headers: dict = field(default_factory=dict)

class APIAnalyzer:
    def __init__(self, base_url: str, auth_headers: dict = None):
        self.base_url = base_url.rstrip("/")
        self.session = requests.Session()
        if auth_headers:
            self.session.headers.update(auth_headers)
        self.analyses = []

    def probe_endpoint(self, method: str, path: str) -> EndpointAnalysis:

        url = f"{self.base_url}{path}"
        response = self.session.request(method, url, timeout=30)
        rate_headers = {k: v for k, v in response.headers.items()
                        if any(t in k.lower() for t in
                               ["rate","limit","remaining","retry"])}
        body = response.json() if response.ok else {}
        pagination = ""
        if isinstance(body, dict):
            if "next" in body: pagination = "cursor"
            elif "total_pages" in body: pagination = "offset"
```

```
        analysis = EndpointAnalysis(method, path, response.status_code,
                                    body, pagination, rate_headers)
        self.analyses.append(analysis)
        return analysis

    def generate_documentation(self) -> str:
        data = json.dumps([asdict(a) for a in self.analyses],
                          indent=2, default=str)
        message = client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=8000,
            system="Generate comprehensive API documentation in Markdown.
    ",
            messages=[{"role": "user",
                       "content": f"Document these probe results:\n{data}
    "}],
        )
        return message.content[0].text
```

Listing 4.3: API Discovery and Documentation

API Pagination Patterns

Different APIs use different pagination strategies, and choosing the wrong one leads to missing data or duplicate records. Claude Code can identify the pagination pattern from a sample response:

- **Cursor-based**: The response includes a `next_cursor` or `next_page_token`. Most reliable for data engineering because it handles concurrent inserts gracefully.
- **Offset-based**: Uses `page` and `per_page` parameters. Prone to skipping or duplicating records if data changes during pagination.
- **Keyset-based**: Paginates using a column value (e.g., `created_after`). Efficient and reliable but requires a monotonically increasing key.
- **Link-header**: The `Link` response header contains URLs for next/previous pages (common in GitHub-style APIs).

Tip

API Rate Limit Documentation Always ask CLAUDE CODE to extract rate limit information from the API response headers. Common headers include `X-RateLimit-Limit`, `X-RateLimit-Remaining`, and `Retry-After`. Knowing these before building your connector prevents trial-and-error tuning later.

4.4 Understanding Legacy Systems

Legacy systems are the most challenging sources to onboard. They embed business logic in stored procedures, triggers, BLOB columns, and application-level encoding. Documentation is sparse or outdated. The original developers are often unavailable.

CLAUDE CODE can extract business rules from stored procedure definitions, which is one of the most time-consuming aspects of legacy system onboarding.

```python
def analyze_stored_procedure(proc: dict) -> dict:
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        system="""Analyze this stored procedure. Return JSON:
        1. Purpose in plain English
        2. Business rules as formal statements
        3. Tables read and written
        4. Data transformations performed
        5. Classification: ETL, Business Logic, Reporting, Maintenance
        6. Side effects (inserts, updates, deletes, DDL changes)
        7. Error handling patterns
        8. Dependencies on other procedures or functions""",
        messages=[{"role": "user",
            "content": f"Name: {proc['name']}\n\n{proc['definition']}"}],

    )
    text = message.content[0].text
    return json.loads(text[text.find("{"):text.rfind("}")+1])
```

Listing 4.4: Stored Procedure Business Logic Extractor

Common Legacy System Patterns

When working with legacy systems, CLAUDE CODE can help you recognize and document several common patterns:

- **Trigger-based CDC**: Some systems use database triggers to populate audit tables. Claude Code can parse trigger definitions to understand what changes are tracked and how.
- **EAV (Entity-Attribute-Value) models**: A pattern where entities have a flexible set of attributes stored as rows rather than columns. Claude Code can help you pivot EAV data into a conventional schema.
- **Encoded columns**: Binary or custom-encoded data stored in VARCHAR or BLOB columns. Claude Code can often identify encoding patterns (Base64, custom delimited formats, packed binary) from sample values.
- **Temporal tables**: Manual implementations of slowly changing dimensions, with `valid_from`/`valid_to` columns that predate native temporal table support.

Note

COBOL and Mainframe Systems Claude Code can parse COBOL copybooks and generate Python unpacking code. Feed it the copybook and ask for a struct-based parser—it handles PIC clauses, REDEFINES, and OCCURS with surprising accuracy. For EBCDIC-encoded data, Claude Code can generate the appropriate codepage conversion code. Yes, there is still COBOL in production. Yes, it will outlive us all. Use Claude Code for Everything™, even the things that predate the internet.

Warning

Stored Procedure Side Effects Legacy stored procedures often have non-obvious side effects: updating audit tables, sending emails via `xp_sendmail`, writing to file systems, or calling external services. One company discovered their "simple reporting procedure" was also sending faxes. In 2025. Always ask Claude Code to identify side effects, and verify them with the operations team before building pipelines that might bypass them.

4.5 Generating ERDs from DDL

Entity-Relationship Diagrams are essential for communicating data models to stakeholders, but they are tedious to create and maintain manually. Claude Code generates ERDs in Mermaid, PlantUML, or DBML format from raw DDL:

```python
def generate_erd(ddl: str, format: str = "mermaid") -> str:
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=8000,
        messages=[{"role": "user",
            "content": f"Convert this DDL to a {format} ERD.\n"
                       f"Infer relationships from FKs and naming.\n"
                       f"Group by business domain.\n"
                       f"Color-code by domain.\n"
                       f"Include cardinality (1:1, 1:N, M:N).\n\n"
                       f"DDL:\n{ddl}"}],
    )
    return message.content[0].text
```

Listing 4.5: ERD Generation

The resulting Mermaid or PlantUML diagram can be rendered in Confluence, GitHub/GitLab, or any documentation system that supports these formats. This makes it trivial to keep ERDs up-to-date: re-run the generation whenever the schema changes.

Tip

ERD Best Practices Generate separate ERDs for each business domain rather than one massive diagram. A 200-table ERD is unreadable. Ask CLAUDE CODE to identify domains first, then generate domain-specific ERDs with cross-domain relationships shown as simplified references.

From ERDs to Data Models

Once you have a clear ERD, CLAUDE CODE can help you design your analytical data model. Feed it the ERD (or the DDL) along with your analytical requirements, and ask it to propose a star schema or OBT (One Big Table) design:

```python
def design_star_schema(source_ddl: str, requirements: str) -> str:
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=8000,
        system="""You are a data modeling expert. Design a star schema
        for an analytical warehouse. Include:
        1. Fact tables with grain definition
        2. Dimension tables with SCD type recommendation
        3. DDL for the target warehouse
        4. dbt model SQL for each table
        5. Rationale for modeling decisions""",
```

```
        messages=[{"role": "user",
            "content": f"Source schema:\n{source_ddl}\n\n"
                       f"Analytical requirements:\n{requirements}"}],
    )
    return message.content[0].text
```

Listing 4.6: Star Schema Design from Source ERD

4.6 API Contract Validation

Breaking API changes are a top cause of pipeline failures. An API provider adds a field, changes a type, renames an endpoint, or deprecates a version—and your pipeline breaks at 3 AM. Claude Code compares old and new API responses, classifies changes as BREAKING, WARNING, or SAFE, and generates pytest contract tests.

```
def detect_api_changes(old_response: dict, new_response: dict,
                       endpoint: str) -> dict:
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        system="""Compare two API responses from the same endpoint.
        Classify each change as:
        - BREAKING: removed field, type change, semantic change
        - WARNING: new required field, format change
        - SAFE: new optional field, additional enum value
        Return JSON with changes array and overall_risk.""",
        messages=[{"role": "user",
            "content": f"Endpoint: {endpoint}\n\n"
                       f"Old response:\n{json.dumps(old_response, indent
    =2)}\n\n"
                       f"New response:\n{json.dumps(new_response, indent
    =2)}"}],
    )
    text = message.content[0].text
    return json.loads(text[text.find("{"):text.rfind("}")+1])
```

Listing 4.7: API Contract Change Detection

Tip

Contract Testing in CI/CD Schedule nightly contract tests against source APIs. When CLAUDE CODE detects a breaking change, it can generate the migration code for your pipeline. Store historical API responses in version control to maintain a change log and enable regression testing.

4.7 Source System Profiling

Schema analysis tells you the *structure* of data; profiling tells you the *reality*. Data profiling examines actual data for distributions, quality patterns, and anomalies that the schema alone cannot reveal.

Key profiling dimensions include:

- **Completeness**: What percentage of rows have non-null values for each column?
- **Uniqueness**: Are columns that should be unique actually unique? Do "unique" columns have near-duplicates (e.g., trailing spaces)?
- **Distribution**: What are the min, max, mean, median, and standard deviation for numeric columns? What are the cardinality and top values for categorical columns?
- **Temporal patterns**: How does row count vary by date? Are there gaps? Weekday/weekend patterns? Seasonality?
- **Referential integrity**: Do foreign key values always have matching parent records? What is the orphan rate?
- **Format consistency**: Do phone numbers, addresses, and dates follow consistent formats?

CLAUDE CODE enhances profiling by interpreting results in business context—identifying likely enum values, soft-delete indicators, temporal patterns, and multi-table join keys from data patterns alone.

```
-- Generate profiling statistics for a table
-- Claude Code can customize this template per table
WITH column_stats AS (
```

```
    SELECT
        'users' AS table_name,
        COUNT(*) AS total_rows,
        COUNT(DISTINCT user_id) AS distinct_users,
        COUNT(*) - COUNT(email) AS null_emails,
        COUNT(DISTINCT status) AS status_cardinality,
        MIN(created_at) AS earliest_record,
        MAX(created_at) AS latest_record,
        COUNT(*) FILTER (WHERE deleted_at IS NOT NULL)
            AS soft_deleted_count,
        COUNT(*) FILTER (
            WHERE created_at > CURRENT_DATE - INTERVAL '7 days'
        ) AS recent_7d_count
    FROM source.users
)
SELECT * FROM column_stats;
```

Listing 4.8: Comprehensive profiling query template

Warning

Sampling Bias Always use stratified sampling when profiling large tables. If the table has a date column, sample across all time periods. A profile based on only recent data may miss historical patterns—such as a column that used to allow nulls but was made required six months ago, or a status value that was retired.

Note

Profiling vs. Privacy When profiling tables that may contain PII, use aggregate statistics (counts, distributions, patterns) rather than individual values. Never extract sample PII rows to send to an external API. Profile locally and send only the aggregate results to Claude Code for interpretation.

4.8 Case Study: Onboarding a New Data Source

Scenario: your company acquires a SaaS product with a 180-table PostgreSQL database, 45 API endpoints, and minimal documentation. A perfectly normal Tuesday in enterprise data engineering. The integration team needs a data model, ingestion plan, and initial data quality assessment within two weeks.

Week 1: Discovery and Documentation

1. **Day 1**: Extract DDL from the PostgreSQL database. Run the chunked schema analysis pipeline. Claude Code produces a structured catalog of all 180 tables, grouped into seven business domains (users, subscriptions, billing, content, analytics, notifications, admin). It identifies 23 implicit foreign key relationships not declared in the schema.

2. **Day 2**: Run the API analyzer against all 45 endpoints. Claude Code generates API documentation covering request/response schemas, pagination patterns (cursor-based for listing endpoints, no pagination for detail endpoints), rate limits (100 requests/minute), and authentication (OAuth 2.0 with refresh tokens).

3. **Day 3**: Feed the 47 stored procedures to the business logic extractor. Claude Code classifies 18 as ETL (including a nightly billing reconciliation), 15 as business logic (subscription tier calculations, proration rules), 8 as reporting, and 6 as maintenance (index rebuilds, partition management).

4. **Days 4–5**: Run profiling queries on the 30 most important tables. Feed profiling results to Claude Code for interpretation. Key findings: the `subscriptions` table has a 3% orphan rate on `plan_id`, suggesting a deleted plans table; the `events` table has a 12-hour gap every Sunday (maintenance window); timestamps are in US/Pacific, not UTC.

Week 2: Design and Implementation

1. **Days 6–7**: Design the target data model with Claude Code's help. Generate dbt staging models for the 30 priority tables with appropriate tests and documentation.

2. **Days 8–9**: Build the ingestion pipeline. Claude Code generates Airflow DAGs for both database CDC (using Debezium) and API extraction (custom Python connectors).

3. **Day 10**: Run the full pipeline end-to-end on staging. Use Claude Code to generate data quality assertions comparing source and target row counts, key distributions, and aggregate values.

Table 4.1: Onboarding Time: Traditional vs. Claude Code-Assisted

Activity	Traditional	Claude Code-Assisted
Schema documentation	3–5 days	2 hours
API documentation	2–3 days	1 hour
Business rule extraction	1–2 weeks	3 hours
Data profiling (20 tables)	2–3 days	1 hour
ERD generation	1–2 days	15 min
Target data model design	3–5 days	1 day
Ingestion pipeline build	1–2 weeks	2 days
Total	**5–8 weeks**	**2 weeks**

Tip

Iterate, Don't Trust Blindly Claude Code's initial analysis is a starting point. Validate critical findings with the source system team—assuming the source system team still exists and has not been "restructured." The real gain is accelerating the path to *informed* questions—instead of spending a week figuring out what to ask, you arrive at the right questions on day one.

4.9 Building a Reusable Source Analysis Toolkit

As your team onboards more sources, you will want a standardized toolkit rather than ad-hoc scripts. A well-designed toolkit includes:

1. **Schema analyzer**: The DDL analysis pipeline from Section 4.2, with output stored in a metadata repository.
2. **API prober**: The API analyzer from Section 4.3, extended with authentication support for OAuth, API keys, and custom auth schemes.
3. **Profiler**: Parameterized profiling queries that run against any source database and produce standardized output.
4. **Contract tester**: Scheduled tests that detect source schema and API changes before they break pipelines.
5. **Documentation generator**: Templates that produce consistent documentation for every source, stored in your data catalog.

Note

Standardized Output Formats Store all analysis output in a consistent JSON schema. This enables comparison across sources, trend analysis over time, and integration with your data catalog. Claude Code can generate the JSON schema for your analysis output format.

4.10 Exercises

1. **Schema Archaeology**: Export the DDL from a database you work with. Run the schema analysis pipeline and compare the results to your existing documentation. What did Claude Code discover that you did not know?

2. **API Documentation**: Choose a SaaS API your team uses (Stripe, Salesforce, HubSpot, etc.). Use the API analyzer to probe five endpoints and generate documentation. Compare the generated documentation to the vendor's official docs—where does Claude Code add value, and where does it fall short?

3. **Legacy Procedure Analysis**: Find a stored procedure in your environment that no one fully understands. Feed it to Claude Code and validate the extracted business rules with someone who knows the system. How accurate was the analysis?

4. **Profiling Interpretation**: Run profiling queries against a table with known data quality issues. Feed the results to Claude Code without mentioning the known issues. Does it identify them? What additional issues does it surface?

5. **Contract Test Suite**: Use Claude Code to generate a contract test suite for one of your source APIs. Run it daily for a week. Document any changes detected and how you would handle them in your pipeline.

This chapter demonstrated that Claude Code transforms source system understanding from slow, manual archaeology into a structured, rapid process. The engineer's expertise is amplified, not replaced—you still need to know which questions to ask, how to validate answers, and which findings matter for your use case. But at least you no longer need to spend three days staring at a column named `flg_xref_2b` wondering what it means. Claude Code will stare at it for you.

5

Data Ingestion Pipelines Powered by Claude Code

> Every API is a unique snowflake of dysfunction. Claude Code speaks fluent dysfunction.

Data ingestion is the circulatory system of every data platform. Every API has its own authentication scheme, pagination logic, rate limits, and schema quirks. Every database has its own CDC mechanism. This chapter demonstrates how Claude Code transforms ingestion from a repetitive slog into a structured, accelerated, and more reliable process.

5.1 Traditional Ingestion Patterns

Batch Ingestion

Batch ingestion extracts data at scheduled intervals. It is appropriate when business requirements tolerate hours-old data and source systems cannot support continuous extraction. Batch pipelines are simpler to reason about, cheaper to operate, and easier to recover when something fails—you simply rerun the batch.

The canonical pattern uses a ***high watermark*** to track progress: each run queries the target for the maximum value of an incremental column, then extracts only records newer than that value.

```python
from dataclasses import dataclass
from datetime import datetime
from typing import Optional

@dataclass
class BatchConfig:
    source_table: str
    target_table: str
    incremental_column: str
    batch_size: int = 10_000

class BatchIngestionPipeline:
    def __init__(self, config: BatchConfig, source_conn, target_conn):
        self.config = config
        self.source = source_conn
        self.target = target_conn

    def get_high_watermark(self) -> Optional[datetime]:
        result = self.target.execute(
            f"SELECT MAX({self.config.incremental_column}) "
            f"FROM {self.config.target_table}")
        return result.scalar()

    def run(self):
        watermark = self.get_high_watermark()
        query = f"SELECT * FROM {self.config.source_table}"
        if watermark:
            query += f" WHERE {self.config.incremental_column} > '{
    watermark.isoformat()}'"
        records = self.source.execute(query)
        batch = []
        for record in records:
            batch.append(record)
            if len(batch) >= self.config.batch_size:
                self.target.bulk_insert(self.config.target_table, batch)
                batch = []
        if batch:
            self.target.bulk_insert(self.config.target_table, batch)
```

Listing 5.1: Classic Batch Ingestion Pattern

Note

Watermark Pitfalls High-watermark ingestion assumes the incremental column is monotonically increasing. Late-arriving records or back-dated updates will be missed. And they will arrive late—because data, like houseguests, never arrives when expected. For sources with this behavior, consider a lookback window: subtract a safety margin (e.g., two hours) from the watermark before querying.

Streaming Ingestion

Streaming ingestion processes data continuously with sub-second latency. It requires handling out-of-order events, exactly-once delivery, backpressure, and state management. Common implementations use Apache Kafka, Amazon Kinesis, or Google Pub/Sub as the transport layer, with a consumer framework (Flink, Spark Structured Streaming, or a custom consumer) performing the ingestion.

Warning

Streaming Is Not Always the Answer If your business tolerates five-minute-old data, micro-batch is almost certainly the right choice. Streaming requires 3–5× more engineering effort to build and maintain. Just because you *can* use CLAUDE CODE to build a Kafka-Flink-Iceberg streaming pipeline does not mean you *should*. Evaluate latency requirements honestly before committing to a streaming architecture. Most "we need real-time" requests translate to "we need it faster than once a day," which is a micro-batch, not a Kafka cluster.

Micro-Batch: The Middle Ground

Micro-batch ingestion runs batch extractions at very short intervals—typically every one to fifteen minutes. It combines the simplicity of batch with near-real-time freshness. Tools like Spark Structured Streaming's trigger-once mode and Airflow's timetable API make micro-batch straightforward to implement.

5.2 Generating Ingestion Code from API Documentation

Writing API clients is one of the most time-consuming parts of ingestion. Each API has its own authentication flow, pagination style, rate-limiting headers, and

error format. CLAUDE CODE can read raw API documentation and generate a production-quality client in minutes.

```python
from anthropic import Anthropic

client = Anthropic()

def generate_api_client(api_docs: str) -> str:
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=8000,
        system="""Generate production-quality Python ingestion code:
        1. Handle auth (API keys, OAuth2, JWT)
        2. Implement pagination (cursor, offset, keyset)
        3. Respect rate limits with exponential backoff
        4. Support incremental extraction via watermarks
        5. Include type hints and docstrings
        Generate COMPLETE, RUNNABLE code.""",
        messages=[{"role": "user",
            "content": f"Generate ingestion client:\n\n{api_docs}"}],
    )
    return response.content[0].text
```

Listing 5.2: Generating an API Client from Documentation

Tip

Version-Pin Generated Code Store the API spec alongside generated code in version control. Include a comment referencing the spec version and generation date. When the upstream API changes—and it will, at 2 AM, without a changelog—re-generate from the updated spec and diff against the previous version to understand what changed.

The generated code should be treated as a starting point. Review it carefully, run it against the live API in a development environment, and add integration tests before promoting to production.

5.3 Schema Inference and Mapping

CLAUDE CODE understands semantic meaning: `"2024-03-15T10:30:00Z"` is an ISO 8601 timestamp, not merely a string. `"192.168.1.1"` is an IPv4 address. This semantic awareness produces far better type mappings than rule-based inference

tools.

```python
def infer_schema_with_claude(sample_records: list[dict]) -> dict:
    import json
    sample = json.dumps(sample_records[:20], indent=2, default=str)
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user",
            "content": f"""Infer schema from these records. For each field:
field_name, data_type, semantic_type, nullable, format_pattern,
suggested_warehouse_type, notes. Return JSON.\n\n{sample}"""}],
    )
    return json.loads(response.content[0].text
        .split("```json")[-1].split("```")[0])
```

Listing 5.3: Schema Inference with Semantic Understanding

Note

Confidence Thresholds Establish a confidence threshold (0.8 is reasonable) below which a human must review the mapping. Lower-confidence fields go to a review queue. Fields with ambiguous types—such as numeric strings that might be IDs or quantities—particularly benefit from human review.

Schema inference is a **design-time** activity. Run it once when onboarding a new source, review the results, and commit the finalized schema mapping to version control. Do not call Claude Code at runtime for every record.

Tip

Mapping to Warehouse Types When Claude Code suggests warehouse types, it considers storage efficiency and query performance. For example, it maps UUIDs to `VARCHAR(36)` rather than `TEXT`, uses `NUMERIC(12,2)` for currency rather than `FLOAT` (which introduces rounding errors), and recommends `TIMESTAMPTZ` over `TIMESTAMP` for any field that may cross timezones.

5.4 Building a Universal API Connector

The universal connector has four layers: configuration (YAML), discovery (Claude Code reads docs), execution (generic HTTP with pagination/rate limits), and out-

put (normalized data). This architecture lets you onboard new APIs by writing a YAML configuration file rather than a new Python module.

```
import yaml, requests, time, os, logging
from datetime import datetime, timezone
from typing import Generator, Optional

logger = logging.getLogger(__name__)

class UniversalAPIConnector:
    def __init__(self, config_path: str):
        with open(config_path) as f:
            self.config = yaml.safe_load(f)["connector"]
        self.base_url = self.config["api"]["base_url"]
        self.session = requests.Session()
        auth = self.config["api"]["auth"]
        if auth["type"] == "bearer_token":
            token = os.environ[auth["token_env_var"]]
            self.session.headers["Authorization"] = f"Bearer {token}"

    def extract_endpoint(
        self, endpoint_name: str, since: Optional[datetime] = None,
    ) -> Generator[dict, None, None]:
        ep = next(e for e in self.config["endpoints"]
                  if e["name"] == endpoint_name)
        url = f"{self.base_url}{ep['path']}"
        params = dict(ep.get("params", {}))
        if since and "incremental" in ep:
            params[ep["incremental"]["field"]] = int(since.timestamp())
        pagination = self.config["api"]["pagination"]
        while True:
            response = self.session.get(url, params=params, timeout=30)
            if response.status_code == 429:
                time.sleep(int(response.headers.get("Retry-After", 60)))
                continue
            response.raise_for_status()
            data = response.json()
            records = data.get("data", data.get("results", []))
            for record in records:
                record["_ingested_at"] = datetime.now(timezone.utc).
    isoformat()
                yield record
            if not data.get(pagination.get("has_more_field", ""), False):

                break
```

```
            params[pagination["cursor_param"]] = records[-1][pagination[
    "cursor_field"]]
```

Listing 5.4: Universal API Connector (core extract loop)

```yaml
connector:
  api:
    base_url: "https://api.example.com/v2"
    auth:
      type: bearer_token
      token_env_var: EXAMPLE_API_KEY
    pagination:
      cursor_param: starting_after
      cursor_field: id
      has_more_field: has_more
  endpoints:
    - name: transactions
      path: /transactions
      params:
        limit: 100
      incremental:
        field: created_gte
```

Listing 5.5: Sample YAML Configuration for the Universal Connector

Tip

Test with Recorded Responses Use a library like `responses` or `vcrpy` to record real API responses and replay them in tests. This decouples your test suite from the live API and makes tests fast, deterministic, and free of rate-limit concerns.

5.5 CDC Patterns with Claude Code Assistance

Change Data Capture (CDC) tracks row-level changes in a source database and replicates them to a target. The three major approaches differ in latency, source impact, and implementation complexity.

CLAUDE CODE generates optimized Debezium configurations and CDC event processors with merge logic, including proper handling of inserts, updates, and soft-deletes. When you describe your source schema and target warehouse, CLAUDE

Table 5.1: CDC Approach Comparison

Approach	Latency	Source Impact	Complexity
Timestamp-based	Minutes–hours	Low	Low
Trigger-based	Seconds	High	Medium
Log-based	Seconds	Minimal	High

Code produces a complete Debezium connector configuration, a Kafka consumer that deserializes change events, and the MERGE SQL that applies those changes to the target.

Warning

CDC and Schema Changes Log-based CDC captures the schema at the time the connector was configured. If the source schema changes (columns added, types altered), the CDC pipeline may break silently or drop data. "Silently" is doing a lot of heavy lifting in that sentence. Monitor schema registry compatibility checks and re-generate connectors after DDL changes. Or use Claude Code to generate a schema drift detector that pages you before the CFO notices the revenue dashboard is blank.

5.6 Handling Schema Evolution

Source schemas change: columns are added, removed, renamed, or retyped. Claude Code analyzes before-and-after snapshots, classifies each change (breaking vs. safe), and generates migration DDL.

```
def detect_schema_changes(old_schema: dict, new_schema: dict) -> list:
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user",
            "content": f"""Compare schemas and classify changes as
column_added, column_removed, type_changed, etc.
For each: is_breaking, suggested_action.
Old: {json.dumps(old_schema)}
New: {json.dumps(new_schema)}
Return JSON array."""}],
    )
    return json.loads(response.content[0].text
```

```
        .split("```json")[-1].split("```")[0])
```

Listing 5.6: Schema Change Detection

Safe changes (adding a nullable column) can be applied automatically. Breaking changes (removing a column, narrowing a type) should trigger a review workflow. Claude Code can also generate the downstream impact analysis—which views, stored procedures, dbt models, and dashboards reference the affected columns.

Warning

Schema Evolution in Production Never apply auto-generated migrations to production without human review. This is one of those rare moments where you might need to think for yourself. We apologize. Schema changes cascade through views, procedures, pipelines, and dashboards. A column rename that seems harmless can break dozens of downstream consumers. It's like pulling one thread on a sweater, except the sweater is your entire data platform and the thread is named `user_id_v2_final_FINAL`.

5.7 Error Handling and Dead Letter Queues

A well-designed ***dead letter queue*** (DLQ) captures the failed record, error type, stack trace, and retry count. Claude Code analyzes DLQ failure patterns, identifies root causes, and recommends remediations.

Common failure categories include malformed records (parsing errors), constraint violations (duplicate keys, NULL in required fields), schema mismatches (unexpected fields or types), and transient errors (timeouts, rate limits). Each category demands a different remediation strategy.

```python
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Any

@dataclass
class DLQRecord:
    original_record: dict
    error_type: str
    error_message: str
    stack_trace: str
    source_system: str
```

```
    pipeline_name: str
    retry_count: int = 0
    max_retries: int = 3
    created_at: datetime = field(
        default_factory=lambda: datetime.now(timezone.utc))

    @property
    def is_retryable(self) -> bool:
        return (self.retry_count < self.max_retries
                and self.error_type in ("timeout", "rate_limit", "
    connection"))
```

Listing 5.7: DLQ Record Structure

Tip

DLQ Monitoring Set alerts on DLQ depth and growth rate. A sudden spike usually indicates a systemic issue (API change, schema change, auth expiration) rather than individual bad records. Alert on both absolute depth and rate of change. A DLQ that grows faster than your actual data table is nature's way of telling you something is profoundly wrong.

5.8 Data Validation During Ingestion

Traditional validation uses static rules (type checks, range checks, regex). CLAUDE CODE adds ***semantic validation***: detecting logical inconsistencies (end_date before start_date), implausible values (negative prices, future birth dates), and cross-field violations (country code mismatching phone format).

Use CLAUDE CODE at **design time** to generate validation rules, not at runtime for every record. Reserve runtime CLAUDE CODE calls for truly ambiguous records that cannot be handled deterministically.

```
def generate_validation_rules(schema: dict, sample_data: list) -> str:
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user",
            "content": f"""Given this schema and sample data, generate
a Python validation function that checks:
1. Type conformance  2. Required fields  3. Range constraints
4. Cross-field consistency  5. Format patterns
```

```
        Return a function: def validate_record(record: dict) -> list[str]
        that returns a list of validation error messages (empty if valid).

        Schema: {json.dumps(schema)}
        Sample: {json.dumps(sample_data[:5], default=str)}"""}],
        )
        return response.content[0].text
```

Listing 5.8: Validation Rule Generation with Claude Code

Note

Three-Tier Validation Structure validation in three tiers: (1) syntactic checks (types, formats) that reject records immediately, (2) semantic checks (business rules) that flag records for review, and (3) statistical checks (distribution anomalies) that trigger alerts but do not block ingestion.

5.9 Full Example: Stripe to Snowflake Pipeline

The pipeline follows: Extract (Stripe API with pagination) → Stage (Parquet on S3) → Load (COPY into Snowflake raw tables) → Monitor. This is a common pattern for SaaS API ingestion, and Claude Code can generate most of the boilerplate.

```
CREATE SCHEMA IF NOT EXISTS analytics.raw_stripe;

CREATE OR REPLACE TABLE analytics.raw_stripe.customers (
    raw_data    VARIANT,
    _loaded_at TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP(),
    _file_name VARCHAR
);

CREATE OR REPLACE VIEW analytics.raw_stripe.v_customers AS
SELECT
    raw_data:id::VARCHAR              AS customer_id,
    raw_data:email::VARCHAR           AS email,
    raw_data:name::VARCHAR            AS name,
    TO_TIMESTAMP(raw_data:created::INT) AS created_at,
    raw_data:currency::VARCHAR        AS currency,
    raw_data:delinquent::BOOLEAN      AS is_delinquent,
    _loaded_at
FROM analytics.raw_stripe.customers;
```

Listing 5.9: Snowflake Setup for Stripe Ingestion

The raw-then-view pattern is intentional: raw tables store the original JSON unchanged (for auditability and reprocessing), while views expose typed columns for downstream consumption.

5.10 Performance Considerations

The key principle is: use Claude Code at design time to generate code and rules, then execute that generated code deterministically at runtime.

Table 5.2: When to Use Claude Code vs. Deterministic Code

Task	Approach	Rationale
Schema inference	Claude Code (design-time)	One-time analysis
API client generation	Claude Code (design-time)	Generate code, then run it
Validation rules	Claude Code (design-time)	Generate rules, execute deterministically
Record validation	Deterministic (runtime)	Rules generated by Claude Code
Error triage	Claude Code (on-demand)	Analyze DLQ on failure
Schema evolution	Claude Code (on-change)	Triggered by schema drift detection

Warning

API Cost Management Each Claude Code call has a token cost. For millions of records, even batched calls accumulate quickly. Calculate expected costs before enabling runtime processing. A single Claude Code call per record at $0.003 per call costs $3,000 for one million records. Use Claude Code for Everything™ does not mean "use Claude Code for every row." That way lies a finance team with questions.

5.11 Exercises

1. **API Connector Generation**: Choose a public API (GitHub, Spotify, or OpenWeatherMap). Feed its documentation to Claude Code, generate a complete ingestion client, test against the live API, and document corrections

needed. Measure how long generation took vs. how long manual implementation would have taken.

2. **CDC Pipeline**: Set up PostgreSQL with logical replication. Implement basic log-based CDC using Python. Use Claude Code to generate SQL merge statements for the target table. Test with inserts, updates, and deletes.

3. **Schema Evolution Simulation**: Create two schema versions with ten differences (added columns, removed columns, type changes, renames). Use Claude Code to detect, classify, and generate migration DDL. Verify against a test database and confirm both forward and rollback migrations work.

4. **End-to-End Pipeline**: Build a complete pipeline from a source of your choice to a warehouse. Include extraction with pagination, validation (using rules generated by Claude Code), staging in Parquet, loading, and monitoring. Use Claude Code to generate at least three components.

5. **DLQ Analysis**: Populate a dead letter queue with 50 synthetic failure records spanning at least five error categories. Use Claude Code to analyze the failures, identify patterns, and recommend remediations. Compare the AI-generated analysis to your own manual review.

6

Working with Unstructured Data

> For decades, 80% of enterprise data sat in a corner, unstructured and unloved, like the office plant nobody waters. Claude Code is here to water that plant.

Industry estimates suggest 80–90% of enterprise data is unstructured: documents, emails, images, log files, audio recordings. For decades, data engineers largely ignored this data because extracting structure was prohibitively expensive. Claude Code changes this calculus. What once required specialized OCR, custom NLP pipelines, and teams of annotators can now be accomplished with well-crafted prompts.

Note

The 80/20 Rule of Unstructured Data In most organizations, 80% of document volume comes from 20% of document types. Start with your highest-volume types. Claude Code makes it feasible to handle the long tail without custom extractors for each format.

6.1 The Unstructured Data Challenge

Traditional approaches—regex, OCR, NLP, template matching—are brittle: they work for the specific cases they were designed for and fail unpredictably on anything else. Claude Code brings general-purpose understanding that degrades gracefully on unfamiliar formats.

The fundamental shift is from *programming extraction rules* to *describing what you want extracted*. Instead of writing regex patterns for every possible date format—a task that has driven more engineers to madness than any other—you tell Claude Code "extract all dates" and it handles `2024-03-15`, `March 15, 2024`, `15/03/2024`, and `3-15-24` without separate rules for each. You could also ask Claude Code to extract dates from your therapist's notes about your regex-induced trauma. Use Claude Code for Everything™.

Tip

Start with a Taxonomy Before building extraction pipelines, catalog your unstructured data sources. For each source type, document: volume (documents per day), format (PDF, email, image), complexity (structured tables vs. free-form prose), and business value (what decisions depend on this data). This taxonomy drives prioritization.

6.2 Document Parsing: PDFs, Invoices, Contracts

PDF Text Extraction

PDF is the most common document format in enterprise settings, but it is also one of the hardest to parse. PDFs encode visual layout, not semantic structure. A table that looks clean to a human reader may be stored as dozens of independent text fragments with absolute positioning.

```
import io
from dataclasses import dataclass
from typing import Optional

@dataclass
class ExtractedDocument:
    text: str
    page_count: int
    extraction_method: str  # "text", "ocr", "hybrid"
    confidence: float

def extract_pdf_text(pdf_path: str, ocr_fallback: bool = True) ->
    ExtractedDocument:
    import fitz  # PyMuPDF
    doc = fitz.open(pdf_path)
    pages_text, ocr_pages = [], 0
```

```
    for page_num in range(len(doc)):
        page = doc[page_num]
        text = page.get_text("text").strip()
        if len(text) < 50 and ocr_fallback:
            import pytesseract
            from PIL import Image
            pix = page.get_pixmap(dpi=300)
            img = Image.open(io.BytesIO(pix.tobytes("png")))
            ocr_text = pytesseract.image_to_string(img, lang="eng")
            if ocr_text and len(ocr_text) > len(text):
                text, ocr_pages = ocr_text, ocr_pages + 1
        pages_text.append(f"--- Page {page_num + 1} ---\n{text}")
    doc.close()
    method = "ocr" if ocr_pages == len(pages_text) else (
        "hybrid" if ocr_pages > 0 else "text")
    return ExtractedDocument(
        "\n\n".join(pages_text), len(pages_text), method,
        1.0 if method == "text" else 0.85)
```

Listing 6.1: PDF Extraction with OCR Fallback

Warning

Scanned PDFs and OCR Quality Scanned PDFs—especially older documents, faxes, or low-resolution scans—produce noisy OCR output. If someone in your organization is still sending faxes in 2026, you have bigger problems than OCR accuracy. Always log the extraction method and confidence score. For documents where accuracy is critical (financial statements, legal contracts), route low-confidence extractions to a human review queue.

Parsing Invoices

```
from anthropic import Anthropic
import json
from datetime import datetime, timezone

client = Anthropic()

def parse_invoice(document_text: str, source_filename: str = "") -> dict:

    response = client.messages.create(
```

```
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user", "content":
            f"""Parse this invoice. Extract: invoice_number,
    invoice_date,
due_date, vendor_name, customer_name, currency, subtotal, tax_amount,
total_amount, payment_terms, line_items (array with description,
quantity, unit_price, amount). Return ONLY valid JSON.

Document:\n{document_text}"""}],
    )
    raw = response.content[0].text
    if "```json" in raw:
        raw = raw.split("```json")[1].split("```")[0]
    parsed = json.loads(raw)
    parsed["_source_file"] = source_filename
    parsed["_parse_timestamp"] = datetime.now(timezone.utc).isoformat()
    return parsed
```

Listing 6.2: Invoice Parsing with Claude Code

Contract Analysis

Contracts require semantic understanding beyond structured extraction. Claude Code extracts metadata (type, dates, parties, governing law), key terms (payment, termination, renewal, liability caps, IP ownership), obligations per party, and risk flags (unusual clauses, missing standards, ambiguous language).

Tip

Multi-Pass for Long Documents For contracts exceeding 50 pages, first extract metadata and a table of contents, then process each section individually, and synthesize section-level results into a complete analysis. This multi-pass approach avoids context window limitations and produces more accurate results than trying to process the entire document at once.

6.3 Email Parsing and Classification

Email is a rich but messy data source. A single email thread may contain customer complaints, order references, PII, and attachments—all in free-form text with inconsistent formatting. Claude Code excels at extracting structure from this chaos.

```python
def classify_email(from_addr: str, subject: str, body: str) -> dict:
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=2000,
        messages=[{"role": "user", "content":
            f"""Classify this email. Return JSON:
category (customer_support, billing_issue, complaint, feature_request,
  vendor_communication, internal_notification, spam),
sentiment (positive/neutral/negative/urgent),
priority (low/medium/high/critical),
entities (order numbers, account IDs, amounts),
summary (one sentence).

From: {from_addr}
Subject: {subject}
Body:\n{body[:3000]}"""}],
    )
    raw = response.content[0].text
    if "```json" in raw:
        raw = raw.split("```json")[1].split("```")[0]
    return json.loads(raw)
```

Listing 6.3: Email Classification

Warning

PII in Emails Emails frequently contain PII: names, addresses, phone numbers, social security numbers, and financial details. Before sending content to any external API, ensure compliance with GDPR, HIPAA, and CCPA. Consider PII redaction before the Claude Code call. At minimum, strip email signatures, which often contain personal contact information, inspirational quotes, and legal disclaimers longer than the email itself.

For high-volume email processing, classify emails into buckets using a lightweight model first, then apply Claude Code only to emails that require deeper analysis. This tiered approach reduces costs by 70–80% while maintaining accuracy on the emails that matter most.

Email Thread Reconstruction

Email threads present additional challenges: quoted text, forwarded messages, and reply chains create nested structures. Before classification, reconstruct the thread

by parsing email headers (`In-Reply-To`, `References`), stripping quoted text, and ordering messages chronologically. Claude Code can then analyze the full thread context to produce more accurate classifications than single-message analysis.

6.4 Log File Analysis

When logs do not match known patterns (JSON, Apache combined, syslog), Claude Code can analyze sample lines and generate a Python parser function. For anomaly detection, Claude Code identifies cascading failures, security incidents, resource exhaustion, and unusual error patterns that statistical methods alone would miss.

```python
def generate_log_parser(sample_lines: list[str]) -> str:
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user",
            "content": f"""Analyze these log lines and generate a Python
function: def parse_log_line(line: str) -> dict
Extract: timestamp, level, source, message, identifiers.
Handle edge cases. Return ONLY Python code.

{chr(10).join(sample_lines[:30])}"""}],
    )
    code = response.content[0].text
    if "```python" in code:
        code = code.split("```python")[1].split("```")[0]
    return code.strip()
```

Listing 6.4: Claude Code-Generated Log Parser

Note

Log Parser Validation Always validate a generated parser against a larger sample than the one used for generation. Test with at least 1,000 lines and check the parse success rate. A good parser should handle >99% of lines without errors. Edge cases to watch for: multi-line stack traces, Unicode characters, and log lines that span timezone changes.

6.5 Image-to-Data Pipelines

Claude Code's multimodal capabilities extract structured data directly from images, eliminating the need for separate OCR and computer vision pipelines. Common use cases include extracting data from receipts, business cards, whiteboard photos, charts, and scanned forms.

```python
import base64
from pathlib import Path

def extract_data_from_image(image_path: str, instructions: str) -> dict:
    path = Path(image_path)
    media_map = {".png": "image/png", ".jpg": "image/jpeg", ".jpeg": "image/jpeg"}
    with open(image_path, "rb") as f:
        data = base64.standard_b64encode(f.read()).decode("utf-8")
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        messages=[{"role": "user", "content": [
            {"type": "image", "source": {"type": "base64",
                "media_type": media_map.get(path.suffix.lower(), "image/png"),
                "data": data}},
            {"type": "text", "text": f"{instructions}\n\nReturn as JSON."},
        ]}],
    )
    raw = response.content[0].text
    if "```json" in raw:
        raw = raw.split("```json")[1].split("```")[0]
    return json.loads(raw)
```

Listing 6.5: Image-to-Data Extraction

Tip

Image Resolution Matters For documents and receipts, aim for at least 150 DPI. For detailed tables, 300 DPI is recommended. Below 72 DPI, extraction accuracy degrades significantly. When preprocessing images, convert to grayscale, increase contrast, and deskew rotated documents before sending to Claude Code.

Batch Image Processing

For processing large volumes of images, implement a pipeline that reads from an object store, processes in parallel, and writes structured output to a staging table. Use a work queue (SQS, Redis, or Celery) to distribute work across multiple workers and handle retries for failed extractions.

```python
from concurrent.futures import ThreadPoolExecutor, as_completed
import logging

logger = logging.getLogger(__name__)

def process_image_batch(image_paths: list[str],
                        instructions: str,
                        max_workers: int = 4) -> list[dict]:
    results = []
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = {
            executor.submit(extract_data_from_image, path, instructions): path
            for path in image_paths
        }
        for future in as_completed(futures):
            path = futures[future]
            try:
                result = future.result()
                result["_source_path"] = path
                result["_status"] = "success"
                results.append(result)
            except Exception as e:
                logger.error(f"Failed to process {path}: {e}")
                results.append({"_source_path": path,
                                "_status": "failed",
                                "_error": str(e)})
    return results
```

Listing 6.6: Batch Image Processing Pipeline

Warning

Concurrency and Rate Limits When processing images in parallel, respect the API rate limits. The Anthropic API has per-minute token limits. Set

`max_workers` conservatively and implement a rate limiter. Processing 1,000 images at full concurrency will trigger rate limiting and degrade throughput. Think of rate limits as the bouncer at a club: you can bring friends, but not the entire city.

6.6 Chunking Strategies for Large Documents

Documents that exceed the context window must be split into chunks. The choice of chunking strategy significantly impacts extraction quality.

Table 6.1: Document Chunking Strategies

Strategy	Best For	Pros	Cons
Fixed-size	Uniform text	Simple	Splits mid-sentence
Sentence-based	Prose	Respects boundaries	Variable size
Section-based	Structured docs	Semantic units	Requires heading detection
Sliding window	Overlapping context	No lost context	Redundant processing

Note

Chunk Size Trade-off 3,000–5,000 tokens per chunk with 10–15% overlap provides the best balance of accuracy and efficiency for most document types. Smaller chunks improve precision but increase cost and require more complex reassembly. Larger chunks risk hitting context limits and reduce extraction focus.

When reassembling chunk-level results, handle deduplication carefully. Overlapping chunks will produce duplicate extractions that must be merged. Use entity resolution (matching on key fields) rather than simple string comparison.

```python
import re
from dataclasses import dataclass

@dataclass
class DocumentChunk:
    text: str
    section_title: str
    page_range: tuple[int, int]
    chunk_index: int
```

```
    total_chunks: int

def chunk_by_sections(document_text: str,
                      max_tokens: int = 4000) -> list[DocumentChunk]:
    # Split on common heading patterns
    sections = re.split(
        r'\n(?=(?:ARTICLE|SECTION|CHAPTER|\d+\.)\s)',
        document_text)
    chunks = []
    for i, section in enumerate(sections):
        title_match = re.match(r'^(.+?)[\n:]', section)
        title = title_match.group(1).strip() if title_match else f"
    Section {i+1}"
        chunks.append(DocumentChunk(
            text=section.strip(),
            section_title=title,
            page_range=(0, 0),  # populate from page markers
            chunk_index=i,
            total_chunks=len(sections),
        ))
    return chunks
```

Listing 6.7: Section-Based Chunking Implementation

6.7 Structured Output with Tool Use

For production systems, prefer Claude Code's tool use over JSON prompting. Tool use provides schema-level enforcement: the model *must* return data matching the schema.

```
def extract_invoice_with_tool_use(document_text: str) -> dict:
    tools = [{"name": "store_invoice_data",
        "description": "Store extracted invoice data.",
        "input_schema": {"type": "object",
            "properties": {
                "invoice_number": {"type": "string"},
                "invoice_date": {"type": "string"},
                "vendor_name": {"type": "string"},
                "currency": {"type": "string"},
                "total_amount": {"type": "number"},
                "line_items": {"type": "array", "items": {"type": "
    object",
```

```
                    "properties": {"description": {"type": "string"},
                        "amount": {"type": "number"}},
                    "required": ["description", "amount"]}},
            },
            "required": ["invoice_number","vendor_name","total_amount","
    line_items"],
        }}]
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4000,
        tools=tools,
        tool_choice={"type": "tool", "name": "store_invoice_data"},
        messages=[{"role": "user",
            "content": f"Extract invoice data:\n\n{document_text}"}],
    )
    for block in response.content:
        if block.type == "tool_use":
            return block.input
    raise ValueError("No tool use response")
```

Listing 6.8: Tool Use for Guaranteed Structured Output

Tip

Tool Use vs. JSON Prompting Tool use provides schema enforcement—the model *must* conform. JSON prompting relies on instruction-following, which occasionally fails for complex schemas, usually at the worst possible moment. Tool use also makes schema evolution easier: add new fields to the tool schema and the model will populate them without prompt changes. It is the difference between asking politely and issuing a legally binding contract.

6.8 Quality Metrics

Measuring extraction quality requires a combination of automated and manual evaluation. Automated metrics catch systemic failures; manual evaluation catches subtle accuracy issues.

Track these metrics by document type, not just in aggregate. An overall 98% accuracy rate may hide the fact that invoices achieve 99.5% while contracts achieve only 92%—and contracts may be the higher-value use case.

Table 6.2: Quality Metrics for Unstructured Data Extraction

Metric	Definition	Target
Extraction rate	% of documents yielding output	>99%
Schema conformance	% of outputs matching schema	>99.5%
Field completeness	% of required fields populated	>95%
Value accuracy	% of extracted values correct	>97%
Latency (p50)	Median processing time	<5s
Cost per document	API cost per document	<$0.05

Tip

Monitoring Extraction Quality Over Time Set up automated quality monitoring that compares each batch's metrics against historical baselines. A sudden drop in field completeness often signals a change in the source document format. Catch these regressions early by alerting when any metric deviates more than two standard deviations from its rolling average.

Warning

Ground Truth Is Essential Automated metrics tell you whether extraction *ran*, not whether it was *correct*. A pipeline that confidently extracts the wrong invoice total is worse than one that fails loudly. Manually annotate 50–100 documents of each type to create ground truth for accuracy measurement. Yes, manually. With your hands. We know. Refresh ground truth quarterly as document formats evolve.

6.9 Exercises

1. **PDF Parsing Benchmark**: Collect 20 PDF invoices from diverse sources (different vendors, formats, languages if possible). Build the parsing pipeline and measure extraction rate, field completeness, and accuracy against manually labeled ground truth. Document which invoice styles cause the most errors.

2. **Log Anomaly Injection**: Take a production-like log file (at least 10,000 lines). Inject three anomaly types: error rate spike, new error pattern, and gradual latency degradation. Run the anomaly detector and evaluate detection accuracy. Calculate the false positive and false negative rates.

3. **Tool Use vs. JSON Mode**: Implement both approaches for invoice parsing. Process 50 invoices with each and compare schema conformance, accuracy, failure rate, and cost per document. Document when each approach is preferable.

4. **Image Table Extraction**: Collect 10 images of tables from different sources (screenshots, photos of printed tables, scanned documents). Use Claude Code's vision to extract tabular data. Measure accuracy against hand-transcribed ground truth. Experiment with image preprocessing (contrast adjustment, deskewing) and measure improvement.

5. **Chunking Strategy Comparison**: Take a 100-page document and process it using all four chunking strategies from Table 6.1. Compare extraction completeness, accuracy, processing time, and cost. Determine which strategy works best for your document type.

Part III

Data Storage and Transformation

7

Data Modeling with Claude Code

> A data model is a promise to your future self. Most people break that promise within six months. Claude Code is here to hold you accountable.

Data modeling is the foundation of every reliable data platform. A poorly designed schema leads to sluggish queries, brittle pipelines, and endless rework. This chapter explores how Claude Code accelerates every phase—from translating business requirements into DDL, to designing star schemas, to handling slowly changing dimensions and Data Vault 2.0.

Note

Modeling Is a Conversation The best data models emerge from iterative dialogue between data engineers and business stakeholders. Claude Code can participate in this conversation: describe your business domain in natural language, and Claude Code produces a first-draft schema that serves as a concrete starting point for discussion. Unlike your stakeholders, Claude Code will not change the requirements halfway through the meeting.

7.1 Generating DDL from Natural Language

Tip

When generating DDL, always specify your target database engine (PostgreSQL, MySQL, Snowflake, BigQuery) in the prompt. Each has distinct data types, constraint syntax, and defaults. A `BIGSERIAL` in PostgreSQL becomes `AUTOINCREMENT` in Snowflake.

```sql
CREATE TABLE customers (
    customer_id    BIGSERIAL PRIMARY KEY,
    full_name      VARCHAR(200) NOT NULL,
    email          VARCHAR(320) NOT NULL UNIQUE,
    billing_city   VARCHAR(200),
    billing_country VARCHAR(100) DEFAULT 'US',
    created_at     TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at     TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE orders (
    order_id       BIGSERIAL PRIMARY KEY,
    customer_id    BIGINT NOT NULL REFERENCES customers(customer_id),
    status         VARCHAR(20) NOT NULL DEFAULT 'pending'
                   CHECK (status IN ('pending','shipped','delivered','
    cancelled')),
    status_changed_at TIMESTAMPTZ NOT NULL DEFAULT now(),
    created_at     TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE order_line_items (
    line_item_id   BIGSERIAL PRIMARY KEY,
    order_id       BIGINT NOT NULL REFERENCES orders(order_id) ON DELETE
     CASCADE,
    product_id     BIGINT NOT NULL,
    quantity       INT NOT NULL CHECK (quantity > 0),
    unit_price     NUMERIC(12,2) NOT NULL CHECK (unit_price >= 0)
);

CREATE INDEX idx_orders_customer ON orders(customer_id);
CREATE INDEX idx_line_items_order ON order_line_items(order_id);
```

Listing 7.1: DDL generated from natural language (PostgreSQL)

Notice what Claude Code gets right unprompted: audit columns (`created_at`/`updated_at`), `TIMESTAMPTZ` (critical for multi-timezone), `CHECK` constraints, and indexes on foreign keys. These are patterns that experienced data engineers always include but junior engineers often forget. Claude Code is, in effect, a senior engineer who never has a bad day, never forgets the audit columns, and never names a column `data2`.

Warning

ALTER TABLE: Handle With Care When Claude Code generates `ALTER TABLE` for tables with data, review for data loss or locking. Adding a `NOT NULL` column without a default fails on non-empty PostgreSQL tables. On large tables, adding an index can lock the table for minutes—use `CREATE INDEX CONCURRENTLY` instead. Nothing says "career-defining moment" quite like locking a production table during business hours.

Iterative Refinement

The first DDL draft is rarely final. Use Claude Code iteratively: generate the initial schema, then ask follow-up questions. "What indexes should I add for a query that filters by `status` and sorts by `created_at`?" or "How should I partition this table if it grows to 500 million rows?" Each round produces a more production-ready schema.

```
def refine_schema(current_ddl: str, requirements: str) -> str:
    """Ask Claude Code to refine an existing schema."""
    client = anthropic.Anthropic()
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        messages=[{"role": "user", "content": f"""Here is my current
schema. Refine it based on these additional requirements.
Preserve existing columns and constraints unless explicitly
asked to change them. Explain each change.

Current DDL:
```sql
{current_ddl}
```

New requirements: {requirements}"""}])
    return response.content[0].text
```

Listing 7.2: Iterative Schema Refinement with Claude Code

Note

Version Your Schemas Treat schema definitions like code: store DDL in version control, tag releases, and maintain a changelog. When using Claude Code to iterate on a schema, save each version so you can diff between iterations and understand why each change was made.

7.2 Star and Snowflake Schema Design

In a star schema, a central fact table connects to denormalized dimension tables—optimizing for query simplicity and read performance. A snowflake schema normalizes dimensions further, trading query simplicity for storage efficiency and data integrity.

```
CREATE TABLE fact_orders (
    order_key        BIGSERIAL PRIMARY KEY,
    customer_key     BIGINT NOT NULL REFERENCES dim_customers,
    product_key      BIGINT NOT NULL REFERENCES dim_products,
    date_key         INT NOT NULL REFERENCES dim_date,
    quantity         INT NOT NULL,
    unit_price       NUMERIC(12,2) NOT NULL,
    net_amount       NUMERIC(12,2) NOT NULL,
    total_amount     NUMERIC(12,2) NOT NULL
);

CREATE TABLE dim_customers (
    customer_key     BIGSERIAL PRIMARY KEY,
    customer_id      BIGINT NOT NULL,
    full_name        VARCHAR(200) NOT NULL,
    email            VARCHAR(320),
    segment          VARCHAR(50),
    region           VARCHAR(100),
    effective_from   DATE NOT NULL,
    effective_to     DATE,
    is_current       BOOLEAN NOT NULL DEFAULT TRUE
);

CREATE TABLE dim_date (
```

```
    date_key          INT PRIMARY KEY,   -- YYYYMMDD
    full_date         DATE NOT NULL UNIQUE,
    day_of_week       SMALLINT NOT NULL,
    month_number      SMALLINT NOT NULL,
    quarter           SMALLINT NOT NULL,
    year              INT NOT NULL,
    is_weekend        BOOLEAN NOT NULL,
    is_holiday        BOOLEAN NOT NULL DEFAULT FALSE
);
```

Listing 7.3: Star schema fact and dimensions

Table 7.1: Star vs. snowflake schema comparison

Criterion	Star	Snowflake
Query simplicity	Fewer JOINs	More JOINs
Storage efficiency	Higher redundancy	Lower redundancy
Query performance	Generally faster	Slower for wide queries
BI tool compatibility	Excellent	Good, more setup

Tip

Let the Query Patterns Decide When asking Claude Code to design a dimensional model, provide the top 10 queries your analysts will run. The query patterns determine whether a star or snowflake schema is appropriate, how to grain the fact table, and which dimensions to conformed across subject areas.

7.3 Data Vault 2.0

Data Vault separates structural information (hubs), relationships (links), and descriptive context (satellites). Claude Code dramatically reduces the boilerplate—a single business entity can require three or more tables in Data Vault, each with hash keys, load timestamps, and record source tracking.

```
CREATE TABLE hub_customer (
    hub_customer_hk  CHAR(32) PRIMARY KEY,   -- MD5 of business key
    customer_bk      VARCHAR(100) NOT NULL,
    load_dts         TIMESTAMPTZ NOT NULL DEFAULT now(),
    record_source    VARCHAR(100) NOT NULL
```

```
);

CREATE TABLE link_order_customer (
    link_order_customer_hk CHAR(32) PRIMARY KEY,
    hub_order_hk           CHAR(32) NOT NULL REFERENCES hub_order,
    hub_customer_hk        CHAR(32) NOT NULL REFERENCES hub_customer,
    load_dts               TIMESTAMPTZ NOT NULL DEFAULT now(),
    record_source          VARCHAR(100) NOT NULL
);

CREATE TABLE sat_customer_details (
    hub_customer_hk  CHAR(32) NOT NULL REFERENCES hub_customer,
    load_dts         TIMESTAMPTZ NOT NULL,
    load_end_dts     TIMESTAMPTZ,
    record_source    VARCHAR(100) NOT NULL,
    hash_diff        CHAR(32) NOT NULL,
    full_name        VARCHAR(200),
    email            VARCHAR(320),
    billing_city     VARCHAR(200),
    PRIMARY KEY (hub_customer_hk, load_dts)
);
```

Listing 7.4: Data Vault hub, link, and satellite

Feed CLAUDE CODE a description of your source entities and it generates the full raw vault: hubs, links, and satellites with proper hash key computation. For a domain with ten entities and fifteen relationships, this can save days of mechanical DDL writing. Data Vault without CLAUDE CODE is just suffering with extra tables.

Tip

Hash Key Consistency Use a consistent hash function across all hubs and links. MD5 is standard in Data Vault 2.0, but some teams prefer SHA-256 for collision resistance. Whatever you choose, document it and enforce it. CLAUDE CODE can generate a reusable macro or function for hash computation to ensure consistency across all loading procedures.

Note

When to Use Data Vault Data Vault shines in environments with many source systems, frequent schema changes, and strict auditability requirements. If you have a single source and straightforward reporting needs, a star schema is simpler and more appropriate. CLAUDE CODE can help you evaluate the

trade-off by analyzing your source count, change frequency, and compliance requirements.

7.4 Normalization and Denormalization

Claude Code identifies normalization violations (transitive dependencies, repeating groups) and recommends where to denormalize for analytics. When you specify target query patterns, it tailors the denormalized design accordingly.

Tip

When asking Claude Code to denormalize, always specify target query patterns. A table for "revenue by region by quarter" looks different from one for "customer lifetime value by cohort." Without query context, Claude Code will produce a generic denormalization that serves no pattern well.

A common pattern is to maintain a normalized operational model (3NF) for transactional workloads and a denormalized analytical model (star schema) for reporting. Claude Code can generate the transformation SQL that bridges the two, including proper handling of NULL values, type casting, and business logic.

Warning

Denormalization and Data Freshness Denormalized views and tables introduce a freshness trade-off. A view is always current but may be slow for complex joins. A materialized table is fast but stale until refreshed. Choose based on your query latency requirements and acceptable staleness. Document the refresh cadence for every denormalized table.

```
CREATE OR REPLACE VIEW analytics.wide_orders AS
SELECT
    o.order_id,
    o.created_at AS order_date,
    c.full_name AS customer_name,
    c.segment AS customer_segment,
    p.product_name,
    p.category AS product_category,
    li.quantity,
    li.unit_price,
```

```
    li.quantity * li.unit_price AS line_total
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_line_items li ON o.order_id = li.order_id
JOIN products p ON li.product_id = p.product_id;
```

Listing 7.5: Denormalized view from normalized tables

7.5 Slowly Changing Dimensions

Table 7.2: SCD types

Type	Strategy	Description
0	Retain original	Never update
1	Overwrite	Replace; no history
2	Add new row	Full history with versioning
3	Add new column	Current and previous value
6	Hybrid (1+2+3)	Maximum flexibility

SCD Type 2 is the most common in practice because it preserves complete history. The implementation requires closing out the current row (setting `effective_to` and `is_current = FALSE`) and inserting a new row with the updated values.

```
-- Close out changed rows
MERGE INTO dim_customers AS tgt
USING (
    SELECT customer_id, full_name, email, region, segment,
        MD5(full_name || email || region || segment) AS hash_diff
    FROM staging.stg_customers
) AS src
ON tgt.customer_id = src.customer_id AND tgt.is_current = TRUE
WHEN MATCHED AND tgt.hash_diff != src.hash_diff THEN
    UPDATE SET effective_to = CURRENT_DATE(), is_current = FALSE;

-- Insert new versions and new customers
INSERT INTO dim_customers (
    customer_id, full_name, email, region, segment,
    hash_diff, effective_from, effective_to, is_current
)
SELECT customer_id, full_name, email, region, segment,
```

```
    MD5(full_name || email || region || segment),
    CURRENT_DATE(), NULL, TRUE
FROM staging.stg_customers src
LEFT JOIN dim_customers tgt
    ON src.customer_id = tgt.customer_id AND tgt.is_current = TRUE
WHERE tgt.customer_id IS NULL
   OR tgt.hash_diff != MD5(src.full_name || src.email || src.region ||
    src.segment);
```

Listing 7.6: SCD Type 2 using Snowflake MERGE

Tip

Include a `hash_diff` column on all tracked SCD Type 2 dimensions. Change detection becomes a simple hash comparison rather than column-by-column checks. Use MD5 for speed or SHA-256 if collision resistance matters.

Warning

SCD Type 2 and Fact Table Joins When joining fact tables to SCD Type 2 dimensions, always filter on the date range: `WHERE fact.event_date BETWEEN dim.effective_from AND COALESCE(dim.effective_to, '9999-12-31')`. Joining only on `is_current = TRUE` gives you the current attribute values, not the values at the time of the event. This is the single most common SCD Type 2 bug, and it will make your numbers wrong in ways that are just subtle enough to ship to the CEO before anyone notices.

7.6 Schema Review with Claude Code

Paste your DDL into Claude Code and ask for a review. Common issues it catches: missing NOT NULL constraints, INT primary keys where BIGINT is needed, no indexes on foreign keys, inconsistent naming, missing temporal columns, and over-normalized OLAP schemas.

```
import anthropic, subprocess, sys

def review_schema_changes(diff: str) -> str:
    client = anthropic.Anthropic()
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
```

```
        messages=[{"role": "user",
            "content": f"""Review these migration changes. Flag issues.
```diff
{diff}
```
Format: ## Critical Issues ## Warnings ## Suggestions ## Summary"""}])
    return message.content[0].text

if __name__ == "__main__":
    diff = subprocess.run(
        ["git", "diff", "origin/main", "--", "migrations/"],
        capture_output=True, text=True).stdout
    if not diff:
        print("No migration changes."); sys.exit(0)
    review = review_schema_changes(diff)
    print(review)
    if "## Critical Issues" in review and \
       "None" not in review.split("## Critical Issues")[1].split("##")
    [0]:
        sys.exit(1)
```

Listing 7.7: Automated Schema Review in CI/CD

Note

CLAUDE CODE can also compare two schema versions and generate a diff report—invaluable during code review for flagging breaking changes. Feed it the "before" and "after" DDL and ask it to classify each change as safe, risky, or breaking.

7.7 Schema Migration Generation

CLAUDE CODE generates versioned migration scripts with data preservation, backward compatibility, and rollback procedures. It supports Flyway, Liquibase, Alembic, Django, dbt, and Sqitch conventions.

```
-- Forward: V042__add_loyalty_tier.sql
BEGIN;
ALTER TABLE dim_customers
    ADD COLUMN loyalty_tier VARCHAR(20) DEFAULT 'standard' NOT NULL;
UPDATE dim_customers SET loyalty_tier = CASE
    WHEN lifetime_orders >= 100 THEN 'platinum'
```

```
    WHEN lifetime_orders >= 50  THEN 'gold'
    WHEN lifetime_orders >= 20  THEN 'silver'
    ELSE 'standard' END
WHERE is_current = TRUE;
COMMIT;

-- Rollback: R042__remove_loyalty_tier.sql
BEGIN;
ALTER TABLE dim_customers DROP COLUMN loyalty_tier;
COMMIT;
```

Listing 7.8: Forward and rollback migration

Warning

Transactions or Tears Always wrap migrations in transactions when your database supports transactional DDL (PostgreSQL does; MySQL does not for most DDL). Without transactions, a failed migration leaves the schema in a partially-applied state that is difficult to recover from. It is Schrödinger's migration: both applied and not applied until you look.

Tip

Migration Naming Conventions Adopt a consistent naming convention for migration files: `V{version}__{description}.sql` for forward migrations and `R{version}__{description}.sql` for rollbacks. CLAUDE CODE can enforce this convention when generating migration scripts if you include it in the prompt.

7.8 Exercises

1. **Natural Language to DDL**: Describe an e-commerce domain (customers, products, orders, reviews, inventory) to CLAUDE CODE in plain English. Generate DDL for PostgreSQL and Snowflake. Compare the two outputs and document the differences in data types, constraints, and indexing strategies.

2. **Star Schema Design**: Given a set of business questions ("What is revenue by product category and region, by quarter?"), design a star schema using CLAUDE CODE. Include at least one fact table, four dimension tables, and a date dimension. Write three analytical queries against the schema.

3. **SCD Type 2 Implementation**: Implement a complete SCD Type 2 pipeline: staging table, MERGE statement, and validation queries. Test with a sequence of changes: initial load, updates to existing records, new records, and records with no changes. Verify that history is preserved correctly.

4. **Schema Review Pipeline**: Set up an automated schema review using the CI/CD integration pattern from Section 7.6. Introduce five intentional issues (missing NOT NULL, INT instead of BIGINT, no index on FK, inconsistent naming, missing audit column) and verify that CLAUDE CODE catches all of them.

5. **Data Vault from Scratch**: Describe a business domain with at least five entities and eight relationships. Use CLAUDE CODE to generate the complete raw vault (hubs, links, satellites). Write the loading procedures for two hubs and one satellite.

6. **Normalization Audit**: Take an existing denormalized table from your data warehouse and ask CLAUDE CODE to identify normalization violations. Classify each violation by normal form (1NF, 2NF, 3NF). For each violation, evaluate whether it is intentional (for performance) or accidental (a modeling error).

This iterative workflow—CLAUDE CODE as drafting partner, reviewer, and sounding board—reduces modeling time from weeks to days while maintaining quality. CLAUDE CODE does not replace the data engineer's judgment; it amplifies it by handling mechanical work and surfacing issues that might be overlooked under time pressure. You could also use CLAUDE CODE to design the seating chart for the team offsite. We are not saying you *should*. We are saying you *could*. Use CLAUDE CODE for Everything™.

8

SQL Generation and Optimization

> Writing SQL by hand in 2026 is like churning your own butter. Technically possible. Spiritually questionable.

SQL remains the lingua franca of data. Yet writing correct, performant SQL at scale is genuinely difficult. Analysts struggle with multi-table joins, engineers debate CTE structure versus subqueries with the fervor of religious scholars, and everyone dreads reading undocumented legacy queries left behind by engineers who have since entered witness protection. Claude Code can generate sophisticated SQL, translate between dialects, optimize slow queries by interpreting `EXPLAIN` plans, recommend indexes, produce migration scripts, and serve as the backbone of a natural language analytics interface.

8.1 The Text-to-SQL Challenge

Text-to-SQL converts a natural language question into a syntactically correct and semantically accurate SQL query. The task is hard for several reasons: natural language is ambiguous, real-world schemas are complex (thousands of tables with cryptic names), SQL dialects are fragmented across engines, and a generated query can be syntactically valid yet semantically *wrong*.

Warning

Dialect Confusion Will Ruin Your Day Never assume that SQL generated for one dialect will run on another without modification. Always specify the target engine in your prompt, and validate the output against that engine's documentation or a test environment. "But it works on Postgres" is not a valid debugging strategy for Snowflake.

Note

Research benchmarks like Spider and BIRD measure text-to-SQL accuracy. Claude Code, when given full schema context and few-shot examples, consistently performs at the top of the 70–85% accuracy range and often surpasses it on enterprise schemas where domain context is provided.

8.2 Claude Code's SQL Capabilities Across Dialects

Claude Code handles all major SQL dialects fluently. Here we demonstrate two representative examples.

PostgreSQL

```sql
SELECT
    event_id,
    event_type,
    payload -> 'user' ->> 'email'    AS user_email,
    payload -> 'user' ->> 'plan'     AS user_plan,
    created_at
FROM events
WHERE payload -> 'user' ->> 'plan' = 'enterprise'
  AND created_at >= now() - INTERVAL '30 days'
ORDER BY created_at DESC;
```

Listing 8.1: Claude Code-generated PostgreSQL JSONB query

Snowflake

```sql
SELECT
    o.order_id,
    o.customer_id,
    o.order_date,
    f.value:product_name::STRING   AS product_name,
    f.value:quantity::INT          AS quantity,
    f.value:unit_price::DECIMAL(12,2) AS unit_price
FROM orders o,
    LATERAL FLATTEN(input => o.line_items) f
QUALIFY ROW_NUMBER() OVER (
    PARTITION BY o.customer_id
    ORDER BY o.order_date DESC
) = 1
ORDER BY o.customer_id;
```

Listing 8.2: Claude Code-generated Snowflake query with QUALIFY and FLATTEN

Tip

When working with multiple dialects in the same organization, create a dialect specification document and include it in every prompt to Claude Code. This ensures consistent syntax choices and prevents accidental dialect mixing.

8.3 Building a Text-to-SQL Pipeline with Schema Context

A reliable text-to-SQL system is a pipeline with multiple stages: schema retrieval, prompt construction, query generation, validation, and execution.

1. **Schema Retrieval**: Identify the relevant tables and retrieve their DDL, column descriptions, and sample data.
2. **Prompt Construction**: Assemble schema context, dialect specification, business glossary, and the user's question.
3. **Query Generation**: Send the prompt to Claude Code and receive the generated SQL.
4. **Validation**: Parse the SQL, check for common errors, and optionally run `EXPLAIN`.
5. **Execution**: Run the query and return results.

```
import anthropic
import sqlparse
import json
from dataclasses import dataclass


@dataclass
class SchemaContext:
    tables: list[dict]
    relationships: list[dict]
    glossary: dict[str, str]
    sample_queries: list[dict]


@dataclass
class GeneratedQuery:
    sql: str
    explanation: str
    confidence: float
    dialect: str
    tables_used: list[str]


class TextToSQLPipeline:
    def __init__(
        self,
        client: anthropic.Anthropic,
        schema_store: "SchemaStore",
        dialect: str = "postgresql",
        model: str = "claude-sonnet-4-20250514",
    ):
        self.client = client
        self.schema_store = schema_store
        self.dialect = dialect
        self.model = model

    def generate(
        self, question: str, max_tables: int = 10
    ) -> GeneratedQuery:
        context = self.schema_store.retrieve(
            question, max_tables=max_tables
        )
        prompt = self._build_prompt(question, context)
        response = self.client.messages.create(
            model=self.model,
```

```
            max_tokens=4096,
            system=self._system_prompt(),
            messages=[{"role": "user", "content": prompt}],
        )
        result = self._parse_response(response)
        self._validate_sql(result.sql)
        return result

    def _system_prompt(self) -> str:
        return f"""You are an expert SQL developer
specializing in {self.dialect}. Rules:
1. Use only tables and columns from the provided schema.
2. Use {self.dialect} syntax exclusively.
3. Include comments explaining complex logic.
4. Prefer CTEs over nested subqueries.
5. Always include appropriate WHERE clauses.
6. Return JSON: sql, explanation, confidence (0-1), tables_used."""

    def _build_prompt(self, question, context):
        sections = ["## Schema"]
        for table in context.tables:
            sections.append(f"### {table['name']}")
            sections.append(f"```sql\n{table['ddl']}\n```")
        if context.relationships:
            sections.append("## Relationships")
            for rel in context.relationships:
                sections.append(
                    f"- {rel['from_table']}.{rel['from_col']}"
                    f" -> {rel['to_table']}.{rel['to_col']}"
                )
        if context.glossary:
            sections.append("## Business Glossary")
            for term, defn in context.glossary.items():
                sections.append(f"- **{term}**: {defn}")
        if context.sample_queries:
            sections.append("## Example Queries")
            for ex in context.sample_queries:
                sections.append(
                    f"Q: {ex['question']}\n```sql\n{ex['sql']}\n```"
                )
        sections.append(f"## Question\n{question}")
        return "\n\n".join(sections)

    def _parse_response(self, response):
        data = json.loads(response.content[0].text)
        return GeneratedQuery(
```

```
            sql=data["sql"],
            explanation=data["explanation"],
            confidence=data["confidence"],
            dialect=self.dialect,
            tables_used=data["tables_used"],
        )

    def _validate_sql(self, sql: str) -> None:
        parsed = sqlparse.parse(sql)
        if not parsed:
            raise ValueError("Generated SQL is empty")
        if parsed[0].get_type() != "SELECT":
            raise ValueError("Only SELECT queries are allowed.")
        sql_upper = sql.upper()
        if "DELETE" in sql_upper or "DROP" in sql_upper:
            raise ValueError("Dangerous keywords detected")
```

Listing 8.3: Text-to-SQL pipeline orchestrator

Tip

Few-shot examples are the single most impactful way to improve text-to-SQL accuracy. Maintain a curated library of 50–100 question/SQL pairs covering your schema's most common query patterns.

8.4 Complex Query Generation

Claude Code shines when generating complex queries involving CTEs, window functions, recursive queries, and advanced aggregations.

CTEs and Growth Rate Ranking

```
WITH monthly_category_revenue AS (
    SELECT
        DATE_TRUNC('month', o.created_at)::DATE AS revenue_month,
        p.category,
        SUM(oi.quantity * oi.unit_price) AS monthly_revenue
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
```

```
    WHERE o.status = 'completed'
      AND o.created_at >= DATE_TRUNC(
          'month', CURRENT_DATE - INTERVAL '13 months')
    GROUP BY 1, 2
),
with_growth AS (
    SELECT *, CASE
        WHEN LAG(monthly_revenue) OVER (
            PARTITION BY category ORDER BY revenue_month
        ) > 0 THEN ROUND(
            (monthly_revenue - LAG(monthly_revenue) OVER (
                PARTITION BY category ORDER BY revenue_month
            )) / LAG(monthly_revenue) OVER (
                PARTITION BY category ORDER BY revenue_month
            ) * 100, 2)
        ELSE NULL
    END AS growth_rate_pct
    FROM monthly_category_revenue
),
ranked AS (
    SELECT *, RANK() OVER (
        PARTITION BY revenue_month
        ORDER BY growth_rate_pct DESC NULLS LAST
    ) AS growth_rank
    FROM with_growth
    WHERE growth_rate_pct IS NOT NULL
)
SELECT * FROM ranked WHERE growth_rank <= 3
ORDER BY revenue_month DESC, growth_rank;
```

Listing 8.4: Multi-CTE query with growth rate ranking

Recursive Queries

```
WITH RECURSIVE org_tree AS (
    SELECT employee_id, full_name, title, manager_id,
        1 AS depth, full_name AS path
    FROM employees WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, e.full_name, e.title, e.manager_id,
        ot.depth + 1, ot.path || ' > ' || e.full_name
    FROM employees e
    JOIN org_tree ot ON e.manager_id = ot.employee_id
```

```
    WHERE ot.depth < 20
)
SELECT * FROM org_tree ORDER BY path;
```

Listing 8.5: Recursive CTE for organizational hierarchy

Warning

Infinite Loops: Not Just a Philosophy Problem Recursive CTEs can cause infinite loops if the data contains cycles. Always include a depth limit in the `WHERE` clause of the recursive member. An infinite loop in a recursive CTE is how you turn a $0.50 query into a $5,000 query and simultaneously discover your warehouse's auto-suspend is not configured.

8.5 Query Optimization with Claude Code

Claude Code can interpret `EXPLAIN ANALYZE` output and suggest concrete optimizations. The workflow is: run `EXPLAIN ANALYZE`, send the plan to Claude Code along with the query and DDL, and ask for specific recommendations.

Table 8.1: Common EXPLAIN plan bottlenecks and Claude Code-suggested fixes

Bottleneck	Plan Indicator	Typical Fix
Sequential scan on large table	`Seq Scan` with high row count	Add index on filter/join columns
Nested loop on large tables	`Nested Loop` with high row estimates	Rewrite to enable hash join
Sort spilling to disk	`Sort Method: external merge`	Increase `work_mem`; add index
Inaccurate row estimates	Estimated vs. actual differ by 10x+	Run `ANALYZE`

```
-- Create a composite partial index
CREATE INDEX CONCURRENTLY idx_orders_date_status
    ON orders (created_at, status)
    WHERE status = 'completed';

```

```sql
-- Pre-aggregate before joining to customers
WITH order_revenue AS (
    SELECT o.order_id, o.customer_id,
        DATE_TRUNC('month', o.created_at) AS month,
        SUM(oi.quantity * oi.unit_price) AS order_total
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE o.created_at >= '2025-01-01'
      AND o.created_at < '2026-01-01'
      AND o.status = 'completed'
    GROUP BY o.order_id, o.customer_id,
             DATE_TRUNC('month', o.created_at)
)
SELECT c.segment, orv.month,
    COUNT(*) AS orders, SUM(orv.order_total) AS revenue
FROM order_revenue orv
JOIN customers c ON orv.customer_id = c.customer_id
GROUP BY c.segment, orv.month
ORDER BY orv.month, revenue DESC;
-- Expected improvement: 45s -> ~3s (15x faster, 0 human tears)
```

Listing 8.6: Optimized query with pre-aggregation

Note

Always provide table row counts and the full `EXPLAIN ANALYZE` output to Claude Code, not just `EXPLAIN`. Actual row counts and timing reveal problems that estimated-only plans hide.

8.6 Migration Script Generation

Claude Code can generate safe, idempotent migration scripts that avoid downtime.

```sql
BEGIN;
-- Step 1: Add column as nullable (non-blocking)
ALTER TABLE customers
    ADD COLUMN IF NOT EXISTS customer_tier VARCHAR(20);

-- Step 2: Add check constraint (validated separately)
ALTER TABLE customers
    ADD CONSTRAINT chk_customer_tier
```

```
    CHECK (customer_tier IN (
        'bronze', 'silver', 'gold', 'platinum'
    )) NOT VALID;

-- Step 3: Validate constraint in background (non-blocking)
ALTER TABLE customers
    VALIDATE CONSTRAINT chk_customer_tier;

-- Step 4: Add default for future rows
ALTER TABLE customers
    ALTER COLUMN customer_tier SET DEFAULT 'bronze';
COMMIT;

-- Step 5: Create index CONCURRENTLY (outside transaction)
-- CREATE INDEX CONCURRENTLY idx_customers_tier
--     ON customers(customer_tier);
```

Listing 8.7: Safe migration script generated by Claude Code

Warning

PostgreSQL Gotchas `CREATE INDEX CONCURRENTLY` cannot run inside a transaction block. Claude Code correctly separates this step—because it has read the PostgreSQL documentation more thoroughly than any human ever has or ever will. Always verify migration scripts for this and similar PostgreSQL-specific constraints.

8.7 Full Example: Natural Language Analytics Interface

Here is a complete natural language analytics interface using FastAPI, PostgreSQL, and Claude Code.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import anthropic
import psycopg2
import psycopg2.extras
import json
from datetime import datetime

app = FastAPI(title="NL Analytics API")
```

```
client = anthropic.Anthropic()

SCHEMA_CONTEXT = """
### sales.orders
- order_id (BIGINT PK), customer_id (BIGINT FK),
  order_date (DATE), status (VARCHAR), total_amount (NUMERIC),
  region (VARCHAR): NA, EU, APAC, LATAM

### sales.customers
- customer_id (BIGINT PK), company_name (VARCHAR),
  segment (VARCHAR): enterprise, mid_market, smb

## Business Glossary
- Revenue: SUM(quantity * unit_price * (1 - discount_pct/100))
- Active customer: 1+ completed order in last 12 months
"""

SYSTEM_PROMPT = """You are a SQL analyst. Generate PostgreSQL
queries from natural language. Rules:
1. Use ONLY tables listed in the schema.
2. Prefer CTEs for complex queries.
3. Include LIMIT 1000 unless asked for all rows.
4. Never generate DDL, DML, or DCL statements.
5. Return JSON: {"sql": "...", "title": "...", "description": "..."}"""


class QuestionRequest(BaseModel):
    question: str
    max_rows: int = 1000


@app.post("/ask")
def ask_question(req: QuestionRequest):
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=2048,
        system=SYSTEM_PROMPT,
        messages=[{
            "role": "user",
            "content": f"{SCHEMA_CONTEXT}\n\nQuestion: {req.question}",
        }],
    )
    result = json.loads(response.content[0].text)
    sql = result["sql"]

    # Safety check
```

```
    sql_upper = sql.upper().strip()
    if not sql_upper.startswith(("WITH", "SELECT")):
        raise HTTPException(400, "Only SELECT queries allowed")

    # Execute with timeout
    conn = psycopg2.connect(
        "postgresql://analytics:pass@localhost/warehouse"
    )
    try:
        with conn.cursor(
            cursor_factory=psycopg2.extras.RealDictCursor
        ) as cur:
            cur.execute("SET statement_timeout = '30s';")
            cur.execute(sql)
            rows = cur.fetchmany(req.max_rows)
        conn.commit()
        return {
            "question": req.question,
            "sql": sql,
            "title": result["title"],
            "columns": list(rows[0].keys()) if rows else [],
            "rows": rows,
            "row_count": len(rows),
        }
    finally:
        conn.close()
```

Listing 8.8: Natural language analytics API

Tip

Timeouts: Your Last Line of Defense Always set a `statement_timeout` when executing AI-generated SQL in production. A 30-second timeout is a reasonable starting point for analytics queries. Without a timeout, one accidental Cartesian product could consume your entire warehouse budget before you finish your coffee.

8.8 Index Recommendation

Claude Code can analyze a workload and recommend an optimal index set. Provide the top N slowest queries (from `pg_stat_statements`), table DDL, and existing indexes.

Table 8.2: PostgreSQL index types and use cases

Index Type	Best For	Example
B-tree	Equality and range queries	`CREATE INDEX ... ON t(col)`
Partial	Fixed-predicate queries	`... WHERE status = 'active'`
GIN	Full-text, JSONB, arrays	`... USING gin(payload)`
BRIN	Large, naturally ordered tables	`... USING brin(created_at)`

Tip

BRIN: The Most Underrated Index BRIN indexes are spectacularly efficient for append-only tables ordered by timestamp. A BRIN index on 500 million rows might consume only 1 MB compared to 10 GB for a B-tree. If BRIN indexes were a stock, we would tell you to buy. Ask Claude Code if your table qualifies—it is better at identifying BRIN candidates than most humans.

8.9 Anti-Patterns in AI-Generated SQL

AI-generated SQL is remarkably good but not infallible. Like a star employee who occasionally microwaves fish in the office kitchen, it has its failure modes. The five most common are:

1. **Hallucinated tables**: Referencing tables or columns that do not exist. Always validate against the schema catalog.
2. **Silent Cartesian products**: A missing join condition produces every-row-times-every-row results.
3. **Incorrect aggregation granularity**: Joining before counting inflates numbers. Use `COUNT(DISTINCT ...)`.
4. **Timezone mishandling**: Comparing `TIMESTAMPTZ` to naive date strings uses the session timezone silently.
5. **NULL semantics**: `NOT IN` with NULLs in the subquery returns zero rows. Use `NOT EXISTS` instead.

Table 8.3: Validation checklist for AI-generated SQL

Check	What to Verify	Automated?
Schema validity	All tables and columns exist	Yes
Statement type	Only SELECT allowed	Yes
Join completeness	Every join has an ON clause	Yes
Row limit	LIMIT clause present	Yes
NULL safety	NOT IN replaced with NOT EXISTS	Partially
Performance	EXPLAIN cost under threshold	Yes

8.10 Exercises

1. **Dialect Translation**: Take a PostgreSQL query using `FILTER`, `ARRAY_AGG`, and `INTERVAL` and use Claude Code to translate it to BigQuery, Snowflake, and Databricks SQL. Document the differences.

2. **Schema Context Optimization**: Design a two-stage retrieval system for a 2,000-table database that first uses Claude Code to identify likely relevant tables, then retrieves full DDL only for those. Implement both stages in Python.

3. **EXPLAIN Plan Analysis**: Obtain an `EXPLAIN ANALYZE` output for a slow query. Send it to Claude Code with table DDL and ask for optimization recommendations. Implement at least two and measure the improvement.

4. **Anti-Pattern Detector**: Build a Python function that checks AI-generated SQL for the five anti-patterns described in this chapter. Test it against 20 queries generated by Claude Code.

9

dbt and Transformation Pipelines

> dbt without Claude Code is just YAML with extra steps. dbt *with* Claude Code is a lifestyle.

The ***dbt*** (data build tool) framework has fundamentally changed how data teams build transformation pipelines. By treating SQL transformations as software—with version control, testing, documentation, and modular design—dbt brought engineering discipline to the analytics layer. Yet dbt projects are labor-intensive to build and maintain. Writing models, tests, documentation, macros, and source YAML by hand is tedious. Nobody became a data engineer because they dreamed of writing YAML. This is exactly where Claude Code excels. In this chapter we explore how Claude Code accelerates every phase of dbt development: generating models from requirements, writing tests, producing documentation, creating macros, and migrating legacy SQL.

9.1 dbt Fundamentals Review

- **Models**: SQL `SELECT` statements stored as `.sql` files, compiled to `CREATE TABLE AS` or `CREATE VIEW AS`.
- **Sources**: Declarations of raw tables in YAML.
- **Tests**: Schema tests (unique, not_null, accepted_values, relationships) in YAML; data tests as standalone SQL returning zero rows.
- **Macros**: Reusable Jinja functions that generate SQL.

- **Snapshots**: SCD Type 2 tracking of historical changes.

A well-organized dbt project follows a layered architecture: staging (views), intermediate (ephemeral), and marts (tables).

Note

Convention Over Configuration Over Sanity The layered architecture (staging, intermediate, marts) is a convention popularized by the dbt community. Claude Code is well-versed in this convention and follows it when generating models. It has read more dbt projects than any human on Earth, including the ones with 47 layers of indirection and a staging model named `stg_tmp_final_v3_DONOTUSE`.

9.2 Using Claude Code to Generate dbt Models

The key is providing Claude Code with sufficient context: the source schema, the dbt project conventions, and the business logic.

Staging Models

```
with source as (
    select * from {{ source('stripe', 'payments') }}
),
renamed as (
    select
        id as payment_id,
        customer_id as stripe_customer_id,
        amount / 100.0 as payment_amount,
        lower(currency) as currency_code,
        lower(status) as payment_status,
        created as payment_created_at,
        _sdc_received_at as loaded_at,
        is_test
    from source
),
filtered as (
    select * from renamed
    where not is_test
),
```

```
final as (
    select
        {{ dbt_utils.generate_surrogate_key(
            ['payment_id']
        ) }} as payment_sk,
        *
    from filtered
)
select * from final
```

Listing 9.1: Claude Code-generated staging model: stg_stripe___payments.sql

Mart Models

```
{{
    config(
        materialized='incremental',
        unique_key=['customer_sk', 'revenue_month'],
        incremental_strategy='merge',
        cluster_by=['revenue_month']
    )
}}

with payments as (
    select * from {{ ref('int_payments_pivoted') }}
),
customers as (
    select * from {{ ref('stg_stripe__customers') }}
),
final as (
    select
        c.customer_sk,
        p.payment_month as revenue_month,
        c.customer_name,
        c.customer_segment,
        p.gross_revenue,
        p.successful_payments as payment_count,
        p.avg_payment_amount,
        case
            when p.failed_payments > 0
            then round(p.failed_payments::float
                / nullif(p.total_payments, 0) * 100, 2)
            else 0
```

```
        end as failure_rate_pct,
        current_timestamp() as dbt_updated_at
    from payments p
    left join customers c
        on p.stripe_customer_id = c.stripe_customer_id
    {% if is_incremental() %}
    where p.payment_month >= (
        select max(revenue_month) from {{ this }}
    ) - interval '3 months'
    {% endif %}
)
select * from final
```

Listing 9.2: Claude Code-generated mart model: fct_revenue.sql

Tip

Late-Arriving Data Is Not a Bug, It's a Feature When generating incremental models, always include a lookback window in the `is_incremental()` block to handle late-arriving data. Data is never on time. If your data were a person, it would miss every flight.

9.3 Generating dbt Tests

Claude Code can generate comprehensive test suites that would take hours to write manually. Hours that you would rather spend doing literally anything else, including attending meetings.

```
version: 2
models:
  - name: fct_revenue
    description: >
      Monthly revenue fact table at the customer-month grain.
    columns:
      - name: customer_sk
        tests: [not_null]
      - name: revenue_month
        tests: [not_null]
      - name: gross_revenue
        tests:
          - not_null
          - dbt_utils.accepted_range:
```

```
              min_value: 0
              inclusive: true
      - name: failure_rate_pct
        tests:
          - dbt_utils.accepted_range:
              min_value: 0
              max_value: 100
      - name: customer_segment
        tests:
          - accepted_values:
              values: [enterprise, mid_market, smb, unknown]
    tests:
      - dbt_utils.unique_combination_of_columns:
          combination_of_columns:
            - customer_sk
            - revenue_month
```

Listing 9.3: Claude Code-generated schema tests for the finance mart

Custom Generic Tests

```
{% test recency(model, column_name,
                datepart='day', interval=1) %}
with max_date as (
    select max({{ column_name }}) as max_val
    from {{ model }}
)
select max_val from max_date
where max_val < dateadd(
    '{{ datepart }}', -{{ interval }}, current_timestamp()
)
{% endtest %}
```

Listing 9.4: Claude Code-generated custom generic test: recency check

Warning

The Silent Test Graveyard Custom generic tests must be placed in the `tests/generic/` directory and named `test_<test_name>.sql`. Mismatches cause silent failures where the test is never discovered. Your test is not "pass-

> ing." Your test does not *exist*. This is not a Zen koan; it is a bug.

9.4 Claude Code for dbt Documentation

Claude Code can generate comprehensive documentation from model SQL, business context, and column names.

```python
import anthropic
import yaml
from pathlib import Path


def generate_model_docs(
    model_path: str, client: anthropic.Anthropic,
) -> dict:
    with open(model_path) as f:
        model_sql = f.read()
    model_name = Path(model_path).stem
    prompt = f"""Analyze this dbt model and generate YAML documentation.
Model name: {model_name}
SQL:
```sql
{model_sql}
```
Generate: model description (2-3 sentences), column descriptions, tests.
Return ONLY valid YAML."""

    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=4096,
        messages=[{"role": "user", "content": prompt}],
)
    return yaml.safe_load(response.content[0].text)
```

Listing 9.5: Script to generate dbt documentation with Claude Code

9.5 Macro Generation with Claude Code

dbt macros are Jinja functions that generate SQL. Writing macros requires fluency in both SQL and Jinja—a combination Claude Code handles well.

```
{% macro safe_divide(numerator, denominator, default=0) %}
    case
        when {{ denominator }} is null then {{ default }}
        when {{ denominator }} = 0 then {{ default }}
        else {{ numerator }}::float / {{ denominator }}
    end
{% endmacro %}
```

Listing 9.6: Claude Code-generated macro: safe division

```
{% macro date_spine(start_date, end_date, datepart='day') %}
    {%- if target.type == 'snowflake' -%}
    select dateadd('{{ datepart }}', seq4(),
        '{{ start_date }}'::date) as date_{{ datepart }}
    from table(generator(rowcount => 10000))
    where date_{{ datepart }} <= '{{ end_date }}'::date
    {%- elif target.type == 'bigquery' -%}
    select date_{{ datepart }}
    from unnest(generate_date_array(
        date '{{ start_date }}', date '{{ end_date }}',
        interval 1 {{ datepart }}
    )) as date_{{ datepart }}
    {%- elif target.type == 'postgres' -%}
    select date_{{ datepart }}::date
    from generate_series(
        '{{ start_date }}'::date, '{{ end_date }}'::date,
        '1 {{ datepart }}'::interval
    ) as date_{{ datepart }}
    {%- endif -%}
{% endmacro %}
```

Listing 9.7: Claude Code-generated macro: cross-database date spine

Tip

Jinja: Where Whitespace Goes to Cause Problems Use Claude Code to generate macros, but always review the Jinja carefully. Whitespace handling (`{-%` vs `{%`) and trailing commas need human review. Jinja templating is the dark art of data engineering—half SQL, half template language, half existential crisis. Yes, that is three halves. Jinja does not respect the laws of mathematics either.

9.6 Migrating Legacy SQL to dbt Models

One of the most valuable applications is migrating legacy stored procedures into well-structured dbt models. The conceptual shift is from imperative (cursor loops, temp tables, quiet weeping) to declarative (a single SELECT). Claude Code handles this translation with the calm professionalism of someone who did not write the original stored procedure and therefore has no emotional attachment to it.

```
-- Migrated from: sp_daily_revenue_report
-- Cursor loop converted to set-based operation
-- DELETE+INSERT converted to incremental merge
{{
    config(
        materialized='incremental',
        unique_key=['report_date', 'region', 'product_category'],
        incremental_strategy='merge'
    )
}}

with orders as (
    select * from {{ ref('stg_app_db__orders') }}
    where order_status = 'completed'
),
order_items as (
    select * from {{ ref('int_order_items_enriched') }}
),
daily_revenue as (
    select
        o.ordered_at::date as report_date,
        c.customer_country as region,
        oi.product_category,
        count(distinct o.order_id) as order_count,
        sum(oi.net_amount) as total_revenue,
        {{ safe_divide(
            'sum(oi.net_amount)',
            'count(distinct o.order_id)'
        ) }} as avg_order_value
    from orders o
    join order_items oi on o.order_id = oi.order_id
    left join {{ ref('dim_customers') }} c
        on o.customer_id = c.customer_id
    {% if is_incremental() %}
    where o.ordered_at::date >= (
        select max(report_date) - interval '7 days'
```

```
        from {{ this }}
    )
    {% endif %}
    group by 1, 2, 3
)
select *, current_timestamp() as dbt_updated_at
from daily_revenue
```

Listing 9.8: dbt model after migration: fct_daily_revenue.sql

Note

From Imperative to Declarative When migrating stored procedures to dbt, Claude Code excels at the imperative-to-declarative translation because it understands both paradigms. It is fluent in both "how to do it" and "what to do," which is more than can be said for most meeting attendees.

9.7 CI/CD for dbt with Claude Code-Powered Reviews

Adding Claude Code-powered code review to a CI pipeline catches issues that automated tests miss.

```
name: dbt CI
on:
  pull_request:
    paths: ['models/**', 'macros/**', 'tests/**']
jobs:
  dbt-ci:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install dependencies
        run: pip install dbt-snowflake anthropic
      - name: dbt build (modified models only)
        run: |
          dbt build --select state:modified+ \
            --defer --state ./prod_manifest --target ci
      - name: Claude Code Review
        if: always()
        run: python scripts/claude_dbt_review.py
        env:
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
```

Listing 9.9: GitHub Actions workflow for dbt CI

Tip

CI Speed: Because Life Is Short In CI, use `dbt build -select state:modified+` with the `-defer` flag to only build models that changed. This dramatically reduces CI run times for large projects. Nobody wants to wait 45 minutes for CI to tell them they forgot a comma. Use Claude Code to generate the CI workflow too, naturally. Use Claude Code for Everything™.

9.8 Exercises

1. **Staging Model Generation**: Given a raw Salesforce opportunity table with columns `id, Name, AccountId, Amount, StageName, CloseDate, IsDeleted, IsWon, Type`, use Claude Code to generate a complete staging model with proper renaming, type casting, and filtering.

2. **Test Suite Design**: For a `fct_orders` fact table, design a comprehensive test suite with at least 15 assertions across schema tests, custom generic tests, and singular data tests.

3. **Macro Library**: Build five reusable dbt macros: `cents_to_dollars`, `fiscal_quarter`, `slugify`, `pivot_column`, and `scd_type2`.

4. **Legacy Migration**: Take a stored procedure or complex view and use Claude Code to decompose it into staging, intermediate, and mart models, with a comparison test verifying output equivalence.

10

Data Quality and Testing

"Data quality is not a project; it is a discipline."
—Every data team's Confluence page, right above the section titled "Known Issues (Do Not Touch)"

Bad data is expensive. A single incorrect join key, unnoticed schema drift, or silently failing pipeline can cascade through dashboards, ML models, and business decisions. Before Claude Code, data quality was largely aspirational—something you put on the roadmap right after "fix tech debt" and right before "heat death of the universe." This chapter shows how to use Claude Code to build a proactive data quality practice: catching problems before they reach consumers, generating tests automatically, and negotiating data contracts between teams. You could do all of this manually, of course. You could also perform surgery on yourself. Both are technically possible.

10.1 Data Quality Dimensions

Tip

When starting a data quality initiative, pick the two dimensions that cause the most downstream pain (usually completeness and consistency) and instrument those first. Or, if you enjoy suffering, try to fix all six simultaneously. Your standup updates will be riveting.

Table 10.1: The six dimensions of data quality.

Dimension	Definition	Example Check
Accuracy	Data correctly represents real-world entities.	Does `order_total` match the sum of line items?
Completeness	All expected data is present.	What percentage of records have a null email?
Consistency	Same data agrees across multiple places.	Does every `customer_id` in facts exist in the dimension?
Timeliness	Data is available when needed.	Is today's partition populated by 8 AM?
Uniqueness	No unintended duplicates.	Are there duplicate `order_id` values?
Validity	Data conforms to expected format and range.	Is `country_code` always a valid ISO 3166-1 alpha-2 code?

10.2 Traditional vs. Claude Code-Augmented Data Quality

Traditional tools (Great Expectations, Soda, dbt tests) are rule-based and excellent at enforcing *known* constraints. They struggle with: (1) unknown unknowns like misspelled status values, (2) semantic anomalies where data passes type checks but a surge suggests duplication, and (3) cross-table reasoning where upstream changes silently invalidate downstream aggregations. In other words, they catch the problems you already know about. How reassuring.

Claude Code complements rule-based tools by adding a reasoning layer: it *generates* tests, *reviews* them, and fills gaps that static rules cannot cover. Think of it as the difference between a smoke detector and a firefighter who also installs smoke detectors, writes the building code, and judges you for not having done this sooner.

Note

Claude Code does not replace Great Expectations or dbt tests. Think of it as the senior engineer who writes the tests, while the framework executes them. Unlike the actual senior engineer, however, it does not take a three-week vacation right before the audit.

10.3 Claude Code for Anomaly Detection

Statistical anomaly detection catches numerical outliers but struggles with contextual anomalies—values that are statistically normal but semantically suspicious. The hybrid approach: (1) run statistical checks to produce candidates, (2) send candidates with context to Claude Code for triage, (3) Claude Code classifies each as "real anomaly," "expected variation," or "needs investigation." Finally, a human glances at the results and says "looks fine" without reading them, completing the circle of life.

```python
import anthropic
import json

def triage_anomalies(
    anomalies: list[dict],
    table_context: str,
    client: anthropic.Anthropic,
) -> list[dict]:
    prompt = f"""You are a data quality analyst. Classify each anomaly as:
- REAL_ANOMALY: Likely a genuine data quality issue.
- EXPECTED_VARIATION: Normal business variation (e.g., holiday spike).
- NEEDS_INVESTIGATION: Ambiguous; recommend follow-up queries.

Table context: {table_context}
Anomalies: {json.dumps(anomalies, indent=2)}

Respond with JSON array: id, classification, explanation, suggested_query."""

    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=4096,
        temperature=0.0,
        messages=[{"role": "user", "content": prompt}],
    )
    return json.loads(response.content[0].text)
```

Listing 10.1: Hybrid anomaly triage with Claude Code

Tip

Feed Claude Code as much business context as possible: known events (holidays, launches), recent changes (new data sources, schema changes), and historical baselines. Richer context yields more accurate triage. Without context, Claude Code is just a very expensive coin flip.

10.4 Generating Great Expectations Suites

Claude Code can generate a comprehensive Great Expectations suite from just the DDL and a few sample rows. After the first validation run, feed failures back to Claude Code for iterative refinement. It is the software equivalent of "measure once, cut once, then have someone smarter re-measure and re-cut for you."

```python
def generate_gx_suite(
    table_name: str, ddl: str, sample_rows: list[dict],
    business_context: str = "", model: str = "claude-sonnet-4-20250514",
) -> dict:
    client = anthropic.Anthropic()
    prompt = f"""Generate a comprehensive Great Expectations suite for:
Table: {table_name}
DDL: ```sql\n{ddl}\n```
Sample rows: ```json\n{json.dumps(sample_rows[:5], indent=2, default=str
    )}\n```
Business context: {business_context}
Include expectations for EVERY column. Output valid JSON."""

    response = client.messages.create(
        model=model, max_tokens=8192, temperature=0.0,
        messages=[{"role": "user", "content": prompt}],
    )
    return json.loads(response.content[0].text)
```

Listing 10.2: Generating a GX suite with Claude Code

Warning

Always review Claude Code-generated expectations before production use. Pay special attention to range expectations—Claude Code infers ranges from sample data, which may not cover edge cases. It has never seen your data on

> Black Friday. Nobody is ever truly prepared for Black Friday.

10.5 Data Contracts

A ***data contract*** is a formal agreement between a data producer and consumer about the schema, semantics, quality guarantees, and SLAs of a dataset. In theory, it prevents the "I changed a column name and now seventeen dashboards are broken" conversation. In practice, it at least gives you something to point to during the postmortem.

```yaml
apiVersion: v1
kind: DataContract
metadata:
  name: fact-orders-contract
  version: 2.1.0
  owner: order-platform-team
  consumers: [analytics-team, ml-team, finance-team]
schema:
  properties:
    order_id: {type: integer, primaryKey: true, nullable: false}
    order_total: {type: number, nullable: false, minimum: 0.01}
    status: {type: string, nullable: false,
             enum: [PENDING, PROCESSING, SHIPPED, DELIVERED, CANCELLED]}
quality:
  completeness: {threshold: 0.99, columns: [order_id, order_total,
    status]}
  uniqueness: {columns: [order_id]}
  freshness: {maxStaleness: PT6H}
sla:
  availability: 99.5%
  updateFrequency: PT1H
```

Listing 10.3: Data contract structure

Claude Code can generate contracts from DDL and sample data, and can facilitate negotiation between producer and consumer teams by analyzing both sides' requirements and proposing compromises. It is, in effect, a couples therapist for your data teams—except both sides actually listen.

10.6 Building a Data Quality Monitoring System

A monitoring system continuously evaluates data quality and alerts when metrics breach thresholds. The architecture: Scheduler → DQ Checker (SQL + Python) → Claude Code (Triage) → Alerting (Slack/PagerDuty). You could also just wait for someone in Finance to send an angry email, but the formal approach scales better.

```python
from dataclasses import dataclass
from datetime import datetime

@dataclass
class DQCheckResult:
    table_name: str
    check_name: str
    dimension: str
    score: float
    threshold: float
    passed: bool
    details: str
    checked_at: datetime

class DQChecker:
    def __init__(self, warehouse_conn_str, metrics_conn_str):
        self.wh_conn = psycopg2.connect(warehouse_conn_str)
        self.metrics_conn = psycopg2.connect(metrics_conn_str)

    def check_completeness(self, table, columns, threshold=0.99):
        results = []
        for col in columns:
            with self.wh_conn.cursor() as cur:
                cur.execute(f"""
                    SELECT COUNT({col})::FLOAT / NULLIF(COUNT(*), 0)
                    FROM {table}""")
                score = cur.fetchone()[0] or 0.0
            results.append(DQCheckResult(
                table, f"completeness_{col}", "completeness",
                score, threshold, score >= threshold,
                f"Non-null rate: {score:.4f}", datetime.utcnow(),
            ))
        return results

    def check_freshness(self, table, timestamp_col, max_hours=6):
        with self.wh_conn.cursor() as cur:
            cur.execute(f"""
```

```
                SELECT EXTRACT(EPOCH FROM (NOW() - MAX({timestamp_col}))
    ) / 3600.0
                FROM {table}""")
            hours_stale = cur.fetchone()[0] or 999
        score = max(0.0, 1.0 - (hours_stale / max_hours))
        return DQCheckResult(
            table, "freshness", "timeliness", score, 0.5,
            hours_stale <= max_hours,
            f"Data is {hours_stale:.1f}h old (max: {max_hours}h)",
            datetime.utcnow(),
        )
```

Listing 10.4: DQ Checker core methods

Instead of alerting on every threshold breach, use Claude Code to consolidate and prioritize failures into a single intelligent alert with severity, likely root cause, and recommended actions. Your on-call engineer's 3 AM self will thank you. Or at least resent you slightly less.

10.7 Claude Code for Root Cause Analysis

When a data quality issue is detected, Claude Code with tool access can perform autonomous root cause analysis by systematically querying the warehouse, checking pipeline logs, and reviewing schema change history. It is like having a detective on retainer, except this one never sleeps, never eats, and has no opinions about your naming conventions. (That last part is a lie.)

```
def run_rca_agent(issue_description: str, client: anthropic.Anthropic)
    -> str:
    tools = [
        {"name": "run_query", "description": "Run a read-only SQL query.
    ",
         "input_schema": {"type": "object",
             "properties": {"query": {"type": "string"}},
             "required": ["query"]}},
        {"name": "get_pipeline_logs",
         "description": "Get recent Airflow task logs.",
         "input_schema": {"type": "object",
             "properties": {"dag_id": {"type": "string"}},
             "required": ["dag_id"]}},
    ]
    messages = [{"role": "user", "content": issue_description}]
```

```
    for _ in range(10):
        response = client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=4096,
            tools=tools, messages=messages,
        )
        if response.stop_reason != "tool_use":
            return "".join(b.text for b in response.content if b.type ==
 "text")
        messages.append({"role": "assistant", "content": response.
content})
        tool_results = []
        for block in response.content:
            if block.type == "tool_use":
                result = execute_rca_tool(block.name, block.input)
                tool_results.append({"type": "tool_result",
                    "tool_use_id": block.id, "content": result})
        messages.append({"role": "user", "content": tool_results})
    return "RCA agent reached maximum iterations."
```

Listing 10.5: Root cause analysis agent (abbreviated)

Tip

RCA agents work best with access to multiple data sources: the warehouse, pipeline logs, schema change histories, and deployment logs. Each additional source narrows the hypothesis space. Ideally, give it access to everything short of your Slack DMs. Actually, those might help too.

10.8 Automated Test Generation

This system scans a dbt project, generates tests for every model using Claude Code, and writes them to schema YAML files. The dream: complete test coverage without a single human writing a single test. The reality: complete test coverage with a human reviewing the tests Claude Code wrote, which is still dramatically better than the previous state of "we'll add tests next sprint" for forty-seven consecutive sprints.

```
import anthropic
import yaml
import asyncio
from pathlib import Path
```

```python
from dataclasses import dataclass

@dataclass
class DbtModel:
    name: str
    sql_path: Path
    sql_content: str

class DbtTestGenerator:
    def __init__(self, project_dir: Path):
        self.project_dir = project_dir
        self.client = anthropic.AsyncAnthropic()

    def discover_models(self) -> list[DbtModel]:
        models = []
        for sql_file in (self.project_dir / "models").rglob("*.sql"):
            models.append(DbtModel(
                name=sql_file.stem, sql_path=sql_file,
                sql_content=sql_file.read_text()))
        return models

    async def generate_tests(self, model: DbtModel) -> dict:
        prompt = f"""Generate dbt tests for this model.
Model: {model.name}
SQL:
```sql
{model.sql_content}
```
Generate column-level tests (not_null, unique, accepted_values,
relationships) and descriptions. Output YAML only."""

        response = await self.client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=4096,
            temperature=0.0, system="Output only valid YAML.",
            messages=[{"role": "user", "content": prompt}])
        return yaml.safe_load(response.content[0].text.strip())

    async def generate_all(self, models):
        sem = asyncio.Semaphore(5)
        async def limited(m):
            async with sem:
                return m.name, await self.generate_tests(m)
        return dict(await asyncio.gather(*[limited(m) for m in models]))
```

Listing 10.6: Automated dbt test generator

Note

After running the generator, always review the output. Common issues: overly strict accepted_values lists, incorrect relationship tests, and redundant tests on nullable columns. Claude Code is thorough to a fault—it will test columns you forgot existed, which is either helpful or existentially unsettling.

10.9 DQ Metrics and KPIs

Table 10.2: KPIs for a data quality program. Achieving all five simultaneously is sometimes referred to as "the impossible dream" or "next quarter's OKR."

KPI	Definition	Target
DQ Score (composite)	Weighted average across dimensions	≥ 0.95
Test Coverage	% of columns with $\geq$1 test	≥ 0.90
MTTR	Average hours from detection to fix	< 4 hours
Contract Coverage	% of critical tables with contracts	$= 1.00$
False Positive Rate	% of alerts that are not real issues	< 0.10

10.10 Exercises

1. **DQ Dimension Assessment.** Choose a dataset and write SQL queries to measure all six dimensions. Calculate a composite DQ score. Which dimension scores lowest? (Spoiler: it is always timeliness. Always.)

2. **Anomaly Detection Pipeline.** Build a hybrid pipeline: (a) use z-scores to identify candidates, (b) send to Claude Code for contextual triage. Evaluate classification quality on 10+ candidates. Bonus: use Claude Code to also write your grocery list. It is surprisingly good at this.

3. **Data Contract Drafting.** Draft a data contract for a dataset you produce or consume. Use Claude Code to create the initial draft, then refine with a colleague on the other side of the producer/consumer relationship. If the

negotiation goes poorly, Claude Code can also draft the passive-aggressive Slack messages.

4. **Root Cause Analysis.** Simulate a DQ issue by introducing nulls or duplicates. Run your DQ checks, then use the RCA agent to investigate. Did Claude Code identify the correct root cause? If so, congratulations—you now have a colleague who is better at debugging than most of your team.

Part IV

Data Serving and Analytics

11

Building Analytical Queries with Claude Code

> *"Write SQL," they said. "It'll be fun," they said. Nobody mentioned the part where you spend four hours debugging a window function at 11 PM because the VP needs "just a quick analysis" by morning.*

Modern data teams face ever-growing demand for analytical queries—from ad hoc Slack requests to production dashboards refreshing every fifteen minutes. Claude Code can fundamentally reshape this workflow by accelerating every phase of query authoring, review, and deployment. Sure, you *could* write all this SQL yourself. You could also churn your own butter. Both activities build character and waste time in roughly equal measure. In this chapter we build production-grade patterns for dashboard query generation, metrics layers, self-service analytics, cohort analysis, A/B testing, and query optimization.

11.1 Dashboard Query Generation

Dashboard queries must be deterministic, performant under concurrent execution, and return results that map cleanly to visualization primitives. The most important input to Claude Code is an accurate, curated schema description. Feed it garbage, and it will produce beautifully formatted garbage—which is arguably worse than ugly garbage because people trust it.

```
WITH daily_revenue AS (
```

```
    SELECT
        DATE(oi.ordered_at)                     AS order_date,
        COUNT(DISTINCT oi.order_id)             AS total_orders,
        COUNT(DISTINCT oi.customer_id)          AS unique_customers,
        SUM(oi.net_revenue)                     AS gross_revenue,
        SUM(oi.net_revenue) - SUM(oi.shipping_cost) AS net_revenue,
        SUM(oi.net_revenue)
            / NULLIF(COUNT(DISTINCT oi.order_id), 0) AS avg_order_value
    FROM analytics.fct_order_items oi
    WHERE oi.ordered_at >= DATEADD('day', -90, CURRENT_DATE)
      AND oi.fulfillment_status <> 'returned'
    GROUP BY 1
),
with_rolling AS (
    SELECT dr.*,
        AVG(dr.gross_revenue) OVER (
            ORDER BY dr.order_date ROWS BETWEEN 6 PRECEDING AND CURRENT
    ROW
        ) AS revenue_7d_moving_avg,
        LAG(dr.gross_revenue, 7) OVER (ORDER BY dr.order_date)
            AS gross_revenue_prev_week
    FROM daily_revenue dr
)
SELECT * FROM with_rolling ORDER BY order_date DESC;
```

Listing 11.1: Claude Code-generated daily revenue query

Tip

Always instruct Claude Code to include `NULLIF` guards on denominators and to exclude cancelled or returned rows explicitly. Dashboard queries that surface division-by-zero errors erode trust faster than any other bug. Nothing says "we have our data together" quite like a dashboard displaying `Infinity%`.

11.2 Metrics Layers and Semantic Layers

A metrics layer provides a single source of truth for business metric definitions. Without one, you get the classic data team experience: three dashboards, three definitions of "revenue," three VPs arguing about whose number is right. Claude Code excels at generating dbt metric definitions and Cube.js schema files from plain-English metric descriptions.

```yaml
semantic_models:
  - name: order_items
    defaults:
      agg_time_dimension: ordered_at
    model: ref('fct_order_items')
    entities:
      - name: order_item
        type: primary
        expr: order_item_id
    measures:
      - name: total_net_revenue
        agg: sum
        expr: net_revenue
        description: Sum of net revenue excluding returned items.
        create_metric: true
      - name: total_orders
        agg: count_distinct
        expr: order_id
        create_metric: true

metrics:
  - name: average_order_value
    type: derived
    label: "Average Order Value (AOV)"
    description: "Net revenue divided by distinct order count."
    type_params:
      expr: total_net_revenue / NULLIF(total_orders, 0)
      metrics:
        - name: total_net_revenue
        - name: total_orders
```

Listing 11.2: dbt metric definition generated by Claude Code

Note

Always include a `description` field in semantic layer definitions. These descriptions become documentation that downstream consumers—both human and AI—use to understand each metric. Future You will thank Present You. Future You thanks Present You so rarely that you should seize every opportunity.

11.3 Self-Service Analytics

Claude Code enables natural-language analytics interfaces where business users ask questions in plain English and receive validated SQL and results. This is the dream that every BI vendor has sold since 2005, except now it actually works. The architecture has four components: context retrieval (schema + metrics from a vector store), query generation, validation (parse + allowlist check via `sqlglot`), and execution with presentation.

```python
import anthropic
import sqlglot

ALLOWED_SCHEMAS = {"analytics", "metrics", "staging"}

class AnalyticsQueryEngine:
    def __init__(self, client: anthropic.Anthropic, schema_context: str):
        self.client = client
        self.schema_context = schema_context

    def generate_sql(self, question: str) -> str:
        response = self.client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=4096,
            system="You are an analytics SQL generator. Snowflake dialect. "
                   "Only reference tables in the provided schema.",
            messages=[{"role": "user",
                       "content": f"Schema:\n{self.schema_context}\n\n"
                                  f"Question: {question}"}],
        )
        return response.content[0].text.strip()

    def validate_sql(self, sql: str) -> tuple[bool, str]:
        try:
            parsed = sqlglot.parse(sql, dialect="snowflake")
        except Exception as e:
            return False, f"Parse error: {e}"
        for stmt in parsed:
            for table in stmt.find_all(sqlglot.exp.Table):
                if (table.db or "").lower() not in ALLOWED_SCHEMAS:
                    return False, f"Disallowed schema: {table.db}"
            if not isinstance(stmt, sqlglot.exp.Select):
                return False, "Only SELECT statements allowed."
        return True, "Validation passed."
```

Listing 11.3: Self-service analytics query engine

Tip

Always validate CLAUDE CODE-generated SQL before execution. The `sqlglot` library provides dialect-aware parsing and catches table references outside your allowlist without hitting the warehouse. Letting unvalidated AI-generated SQL touch your production warehouse is the data engineering equivalent of texting your ex at 2 AM—technically possible, never advisable.

11.4 Building a Metrics Store

A metrics store materializes pre-computed metric values for fast retrieval—essential for sub-second lookups in executive dashboards and real-time alerts. Because nothing matters more than ensuring the CEO's dashboard loads in under one second, even if the underlying data is three hours stale.

```
from dataclasses import dataclass, field
import hashlib

@dataclass
class MetricDefinition:
    name: str
    sql_expression: str
    source_table: str
    time_column: str
    granularities: list[str]
    dimensions: list[str] = field(default_factory=list)
    filters: list[str] = field(default_factory=list)

    @property
    def metric_id(self) -> str:
        raw = f"{self.name}:{self.source_table}:{self.sql_expression}"
        return hashlib.md5(raw.encode()).hexdigest()[:12]

class MetricsStore:
    def __init__(self, engine, target_schema="metrics"):
        self.engine = engine
        self.target_schema = target_schema

```

```
    def generate_materialization_sql(self, metric, granularity, lookback
    =90):
        dim_cols = ", ".join(metric.dimensions) if metric.dimensions
    else ""
        where = " AND ".join(
            [f"{metric.time_column} >= DATEADD('day', -{lookback},
    CURRENT_DATE)"]
            + metric.filters
        )
        return f"""
        INSERT INTO {self.target_schema}.metric_values
            (metric_id, metric_name, granularity, period, metric_value)
        SELECT '{metric.metric_id}', '{metric.name}', '{granularity}',
            DATE_TRUNC('{granularity}', {metric.time_column}),
            {metric.sql_expression}
        FROM {metric.source_table}
        WHERE {where}
        GROUP BY DATE_TRUNC('{granularity}', {metric.time_column})
            {', ' + dim_cols if dim_cols else ''}
        ON CONFLICT (metric_id, granularity, period)
        DO UPDATE SET metric_value = EXCLUDED.metric_value;"""
```

Listing 11.4: Metrics store computation engine

Warning

Metrics stores can grow rapidly when combining multiple granularities with high-cardinality dimensions. Use Claude Code to audit dimension cardinality before adding new slices. "Let's add a per-user daily breakdown" sounds innocent until your metrics table has more rows than the source data.

11.5 Time Series Analysis Queries

Claude Code generates sophisticated temporal SQL patterns. Year-over-year comparisons, rolling averages, anomaly detection—all the queries that make you question whether SQL was really designed for this, or whether we are collectively engaged in an elaborate act of denial.

```
WITH monthly_revenue AS (
    SELECT DATE_TRUNC('month', ordered_at) AS revenue_month,
        EXTRACT(YEAR FROM ordered_at) AS yr,
```

```
        EXTRACT(MONTH FROM ordered_at) AS mo,
        SUM(net_revenue) AS total_revenue
    FROM analytics.fct_order_items
    WHERE fulfillment_status <> 'returned'
      AND ordered_at >= DATEADD('year', -3, CURRENT_DATE)
    GROUP BY 1, 2, 3
)
SELECT curr.revenue_month,
    curr.total_revenue AS current_revenue,
    prev.total_revenue AS prior_year_revenue,
    ROUND((curr.total_revenue - prev.total_revenue)
        / NULLIF(prev.total_revenue, 0) * 100, 2) AS yoy_growth_pct
FROM monthly_revenue curr
LEFT JOIN monthly_revenue prev
    ON curr.mo = prev.mo AND curr.yr = prev.yr + 1
WHERE curr.yr = EXTRACT(YEAR FROM CURRENT_DATE)
ORDER BY curr.revenue_month;
```

Listing 11.5: Year-over-year comparison with growth rates

```
WITH daily_revenue AS (
    SELECT DATE(ordered_at) AS order_date, SUM(net_revenue) AS revenue
    FROM analytics.fct_order_items
    WHERE fulfillment_status <> 'returned'
      AND ordered_at >= DATEADD('day', -180, CURRENT_DATE)
    GROUP BY 1
),
with_stats AS (
    SELECT order_date, revenue,
        AVG(revenue) OVER (ORDER BY order_date
            ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING) AS rolling_mean,
        STDDEV(revenue) OVER (ORDER BY order_date
            ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING) AS rolling_stddev
    FROM daily_revenue
)
SELECT order_date, revenue, rolling_mean,
    ROUND((revenue - rolling_mean) / NULLIF(rolling_stddev, 0), 2) AS
    z_score,
    CASE
        WHEN ABS((revenue - rolling_mean) / NULLIF(rolling_stddev, 0)) >
     3 THEN 'CRITICAL'
        WHEN ABS((revenue - rolling_mean) / NULLIF(rolling_stddev, 0)) >
     2 THEN 'WARNING'
        ELSE 'NORMAL'
```

```
    END AS anomaly_status
FROM with_stats WHERE rolling_stddev IS NOT NULL
ORDER BY order_date DESC;
```

Listing 11.6: Z-score anomaly detection in SQL

Tip

When generating anomaly detection queries, specify the baseline window size (e.g., 30 days) and sensitivity threshold (e.g., 2 or 3 standard deviations). These depend on your data's natural volatility. Set the threshold too low and you will get paged for every Tuesday. Set it too high and you will learn about problems from Twitter.

11.6 Cohort Analysis and Funnel Queries

Cohort and funnel queries are among the most requested and error-prone analytical patterns. They are also the queries most likely to be described as "simple" in a Slack message by someone who has never written SQL.

```
WITH first_purchase AS (
    SELECT customer_id,
        DATE_TRUNC('month', MIN(ordered_at)) AS cohort_month
    FROM analytics.fct_order_items
    WHERE fulfillment_status <> 'returned'
    GROUP BY 1
),
monthly_activity AS (
    SELECT DISTINCT customer_id,
        DATE_TRUNC('month', ordered_at) AS activity_month
    FROM analytics.fct_order_items
    WHERE fulfillment_status <> 'returned'
),
cohort_data AS (
    SELECT fp.cohort_month, ma.activity_month,
        DATEDIFF('month', fp.cohort_month, ma.activity_month)
            AS months_since_first,
        COUNT(DISTINCT ma.customer_id) AS active_customers
    FROM first_purchase fp
    INNER JOIN monthly_activity ma ON fp.customer_id = ma.customer_id
    GROUP BY 1, 2, 3
```

```sql
),
cohort_sizes AS (
    SELECT cohort_month, COUNT(DISTINCT customer_id) AS cohort_size
    FROM first_purchase GROUP BY 1
)
SELECT cd.cohort_month, cs.cohort_size, cd.months_since_first,
    cd.active_customers,
    ROUND(cd.active_customers::FLOAT / cs.cohort_size * 100, 2) AS
    retention_pct
FROM cohort_data cd
INNER JOIN cohort_sizes cs ON cd.cohort_month = cs.cohort_month
WHERE cd.months_since_first <= 12
ORDER BY cd.cohort_month, cd.months_since_first;
```

Listing 11.7: Monthly retention cohort analysis

Warning

Funnel queries using `COUNT(DISTINCT session_id)` per step do *not* enforce ordering—a session counted in step 3 may never have hit step 2. For strict sequential funnels, use window functions with `LEAD`/`LAG` or `MATCH_RECOGNIZE`. Or just let Claude Code handle it, because life is short and funnel queries are long.

11.7 A/B Test Analysis

Claude Code can generate warehouse-native significance calculations. Always verify three things: (1) the experiment window is correctly bounded, (2) users are attributed to exactly one variant, and (3) the metric window is consistent across control and treatment. Getting any of these wrong will produce statistically significant results that are also statistically meaningless—the data equivalent of a beautifully typeset lie.

```sql
WITH variant_stats AS (
    SELECT variant, COUNT(*) AS n, SUM(converted) AS successes,
        AVG(converted) AS p_hat
    FROM user_outcomes GROUP BY 1
),
test_inputs AS (
    SELECT
        MAX(CASE WHEN variant='variant_a' THEN p_hat END) AS p_control,
```

```
        MAX(CASE WHEN variant='variant_a' THEN n END) AS n_control,
        MAX(CASE WHEN variant='variant_b' THEN p_hat END) AS p_treatment,

        MAX(CASE WHEN variant='variant_b' THEN n END) AS n_treatment
    FROM variant_stats
),
z_test AS (
    SELECT *,
        (p_treatment - p_control) / SQRT(
            ((p_control*n_control + p_treatment*n_treatment)
             / (n_control + n_treatment))
            * (1 - ((p_control*n_control + p_treatment*n_treatment)
                    / (n_control + n_treatment)))
            * (1.0/n_control + 1.0/n_treatment)
        ) AS z_statistic
    FROM test_inputs
)
SELECT
    ROUND(p_control*100, 4) AS control_rate_pct,
    ROUND(p_treatment*100, 4) AS treatment_rate_pct,
    ROUND(z_statistic, 4) AS z_stat,
    CASE
        WHEN ABS(z_statistic) > 2.576 THEN 'p < 0.01'
        WHEN ABS(z_statistic) > 1.960 THEN 'p < 0.05'
        ELSE 'Not significant'
    END AS significance_level
FROM z_test;
```

Listing 11.8: Z-test for conversion rate difference in SQL

11.8 Query Optimization with Claude Code

Claude Code can analyze execution plans and suggest optimizations. Common recommendations include: early filtering for partition pruning, predicate alignment with clustering keys, removing `SELECT *` in CTEs, replacing correlated subqueries with JOINs, and pre-aggregating common computations. In other words, all the things you know you should be doing but have not gotten around to because "it only takes 45 seconds and that's fine." It is not fine. Your warehouse bill knows it is not fine.

```
-- BEFORE: Full scan, 45 seconds, 12GB scanned
SELECT customer_id, SUM(net_revenue) AS total_rev
```

```sql
FROM analytics.fct_order_items
WHERE customer_id IN (
    SELECT customer_id FROM analytics.dim_customers
    WHERE customer_tier = 'platinum')
GROUP BY 1;

-- AFTER (Claude Code-optimized): 3 seconds, 800MB
SELECT oi.customer_id, SUM(oi.net_revenue) AS total_rev
FROM analytics.fct_order_items oi
INNER JOIN analytics.dim_customers c
    ON oi.customer_id = c.customer_id
WHERE c.customer_tier = 'platinum'
  AND oi.ordered_at >= DATEADD('year', -2, CURRENT_DATE)
GROUP BY 1;
```

Listing 11.9: Before and after optimization

Warning

Always validate CLAUDE CODE's optimization suggestions by comparing `EXPLAIN` plans and benchmarking on realistic data volumes. Optimizations that help on small datasets can sometimes hurt on large ones. Trust, but verify. Actually, just verify.

12

Reverse ETL and Operational Analytics

"We spent years getting data into the warehouse. Now we spend years getting it back out. The circle of enterprise life is complete."

Data warehouses were built to be the destination. But modern data teams have discovered that the warehouse is also the best *source* for operational systems. ***Reverse ETL***—pushing curated warehouse data back into SaaS tools, CRMs, and marketing platforms—closes the loop between analytical insight and operational action. It is, essentially, admitting that the dashboard nobody checks should have been a Salesforce field all along. Claude Code accelerates every stage: designing sync logic, generating audience segments, building activation pipelines, and monitoring freshness.

12.1 What Is Reverse ETL

Traditional ETL moves data from operational systems into a warehouse. Reverse ETL bridges the gap by syncing curated data back into tools where operational teams work—pushing customer health scores into Salesforce contact records instead of asking salespeople to check dashboards. Because salespeople will never check dashboards. This is a universal constant, like gravity or deployments breaking on Fridays.

Dimension	Traditional ETL	Reverse ETL
Direction	Sources → Warehouse	Warehouse → Destinations
Volume	High (bulk ingestion)	Lower (curated subsets)
Schema control	Warehouse owns schema	Destination API owns schema
Primary users	Data engineers	Data engineers + RevOps + Marketing Ops

Tip

When explaining reverse ETL to stakeholders: "We are going to keep Salesforce automatically updated with the latest customer metrics—no manual exports, no stale spreadsheets." When explaining it to your therapist: "I am now responsible for data flowing in both directions and I can feel my hair thinning."

12.2 Operational Analytics

Reverse ETL is one pillar of ***operational analytics***—the idea that analytics should drive automated actions, not just dashboards that get screenshot'd into PowerPoints. It takes outputs of analytical and ML workflows and feeds them into business processes across sales (lead scores in CRM), marketing (audience segments in ad platforms), support (health scores in ticketing), and finance (revenue data in ERP).

12.3 Using Claude Code to Design Reverse ETL Pipelines

Generating the Source Model

```
{{
  config(
    materialized='incremental',
    unique_key='customer_id',
    cluster_by=['customer_id']
  )
}}

WITH customers AS (
    SELECT contact_id AS customer_id, email, first_name,
        last_name, company_name, owner_id AS sales_owner_id
    FROM {{ ref('stg_crm__contacts') }}
),
```

```
subscriptions AS (
    SELECT customer_id, plan_name AS subscription_tier, mrr,
        subscription_status,
        CASE
            WHEN subscription_status = 'active' THEN 'paying'
            WHEN subscription_status = 'past_due' THEN 'at_risk'
            WHEN subscription_status = 'canceled' THEN 'churned'
            ELSE 'unknown'
        END AS lifecycle_stage
    FROM {{ ref('stg_billing__subscriptions') }}
    QUALIFY ROW_NUMBER() OVER (
        PARTITION BY customer_id ORDER BY current_period_start DESC
    ) = 1
),
usage AS (
    SELECT customer_id,
        COUNT(DISTINCT CASE WHEN activity_date >= DATEADD('day', -30,
    CURRENT_DATE)
            THEN activity_date END) AS active_days_last_30,
        MAX(activity_date) AS last_activity_date
    FROM {{ ref('stg_events__daily_usage_summary') }}
    GROUP BY 1
),
tickets AS (
    SELECT customer_id,
        COUNT(*) AS total_tickets,
        COUNT(CASE WHEN status = 'open' THEN 1 END) AS open_tickets
    FROM {{ ref('stg_zendesk__tickets') }}
    GROUP BY 1
),
health_score AS (
    SELECT c.customer_id,
        LEAST(100, GREATEST(0,
            (COALESCE(u.active_days_last_30, 0) / 30.0) * 40
            + CASE s.lifecycle_stage WHEN 'paying' THEN 25 WHEN 'at_risk'
     THEN 5 ELSE 0 END
            + CASE WHEN COALESCE(t.open_tickets, 0) = 0 THEN 20 WHEN t.
    open_tickets <= 2 THEN 10 ELSE 0 END
            + CASE WHEN u.last_activity_date >= DATEADD('day', -3,
    CURRENT_DATE) THEN 15
                   WHEN u.last_activity_date >= DATEADD('day', -14,
    CURRENT_DATE) THEN 8 ELSE 0 END
        ))::INT AS health_score
    FROM customers c
    LEFT JOIN subscriptions s USING (customer_id)
    LEFT JOIN usage u USING (customer_id)
```

```
    LEFT JOIN tickets t USING (customer_id)
)
SELECT c.*, s.subscription_tier, s.mrr, s.lifecycle_stage,
    u.active_days_last_30, u.last_activity_date,
    t.open_tickets, h.health_score,
    CURRENT_TIMESTAMP() AS synced_at
FROM customers c
LEFT JOIN subscriptions s USING (customer_id)
LEFT JOIN usage u USING (customer_id)
LEFT JOIN tickets t USING (customer_id)
LEFT JOIN health_score h USING (customer_id)
{% if is_incremental() %}
WHERE c.customer_id IN (
    SELECT DISTINCT customer_id FROM {{ ref('
    stg_events__daily_usage_summary') }}
    WHERE activity_date >= DATEADD('day', -2, CURRENT_DATE)
    UNION
    SELECT DISTINCT customer_id FROM {{ ref('stg_zendesk__tickets') }}
    WHERE updated_at >= (SELECT MAX(synced_at) FROM {{ this }})
)
{% endif %}
```

Listing 12.1: dbt model for reverse ETL: reverse_etl_contacts.sql

Tip

Always include a `synced_at` timestamp column in reverse ETL source models. This enables incremental sync logic and freshness verification. It also gives you something to point at during the inevitable "when was this data last updated?" Slack thread.

Change Detection

```
WITH current_snapshot AS (
    SELECT *, MD5(CONCAT_WS('|',
        COALESCE(email, ''), COALESCE(subscription_tier, ''),
        COALESCE(mrr::VARCHAR, ''), COALESCE(health_score::VARCHAR, ''),
        COALESCE(lifecycle_stage, '')
    )) AS record_hash
    FROM {{ ref('reverse_etl_contacts') }}
),
previous_sync AS (
```

```
    SELECT customer_id, record_hash
    FROM {{ ref('reverse_etl_sync_log') }}
    WHERE destination = 'salesforce' AND sync_status = 'success'
    QUALIFY ROW_NUMBER() OVER (
        PARTITION BY customer_id ORDER BY synced_at DESC
    ) = 1
)
SELECT cs.*, CASE
    WHEN ps.customer_id IS NULL THEN 'insert'
    WHEN cs.record_hash <> ps.record_hash THEN 'update'
END AS sync_action
FROM current_snapshot cs
LEFT JOIN previous_sync ps USING (customer_id)
WHERE ps.customer_id IS NULL OR cs.record_hash <> ps.record_hash
```

Listing 12.2: Hash-based change detection for incremental sync

Warning

Ensure null handling is consistent in hash-based change detection: `COALESCE` every field to a deterministic default before hashing, or a null-to-empty-string change will not trigger a sync. This is the kind of bug that silently corrupts data for three weeks before anyone notices, by which point you have synced 50,000 wrong records to Salesforce and the VP of Sales has opinions.

12.4 Customer 360 Views

A ***Customer 360 view*** unifies every data point about a customer into a single, terrifyingly comprehensive record. Claude Code generates a layered approach: an intermediate model joining sources via an identity graph, then a final mart adding RFM scoring and churn risk classification. The result is a table that knows more about your customers than your customers know about themselves.

```
WITH rfm AS (
    SELECT customer_id,
        DATEDIFF('day', GREATEST(last_active_date,
            last_payment_date), CURRENT_DATE) AS recency_days,
        COALESCE(total_active_days, 0) + COALESCE(total_invoices, 0) AS
    frequency_raw,
        COALESCE(lifetime_revenue, 0) AS monetary_value
    FROM {{ ref('int_customer_360_base') }}
```

```
),
rfm_scored AS (
    SELECT *, NTILE(5) OVER (ORDER BY recency_days DESC) AS
    recency_score,
        NTILE(5) OVER (ORDER BY frequency_raw ASC) AS frequency_score,
        NTILE(5) OVER (ORDER BY monetary_value ASC) AS monetary_score
    FROM rfm
)
SELECT b.*, r.recency_score, r.frequency_score, r.monetary_score,
    CASE
        WHEN r.recency_score >= 4 AND r.frequency_score >= 4 AND r.
    monetary_score >= 4 THEN 'Champion'
        WHEN r.recency_score >= 3 AND r.frequency_score >= 3 THEN 'Loyal'

        WHEN r.recency_score <= 2 AND r.frequency_score >= 3 THEN 'At
    Risk'
        ELSE 'Needs Attention'
    END AS rfm_segment
FROM {{ ref('int_customer_360_base') }} b
LEFT JOIN rfm_scored r USING (customer_id)
```

Listing 12.3: Customer 360 mart with RFM scoring (abbreviated)

12.5 Syncing to SaaS Tools

Each destination has unique API semantics and rate limits. Claude Code generates integration code for Salesforce (REST + Bulk API), HubSpot (batch upsert by email), and Intercom (rate-limit-aware per-user updates). Each also has unique ways of making your life difficult, which Claude Code handles with the quiet resignation of someone who has read too many API changelogs.

```
class SalesforceReverseETL:
    def __init__(self, sf_username, sf_password, sf_security_token,
                 bulk_threshold=500):
        self.sf = Salesforce(username=sf_username,
            password=sf_password, security_token=sf_security_token)
        self.bulk_threshold = bulk_threshold

    def upsert_records(self, object_name, external_id_field, records):
        if len(records) > self.bulk_threshold:
            results = self.sf.bulk.__getattr__(
                object_name
```

```
            ).upsert(records, external_id_field, batch_size=5000)
        else:
            for record in records:
                ext_id = record.pop(external_id_field)
                getattr(self.sf, object_name).upsert(
                    f"{external_id_field}/{ext_id}", record)
```

Listing 12.4: Salesforce reverse ETL sync (core logic)

Tip

Intercom's API has stricter rate limits than Salesforce or HubSpot. For large syncs, use bulk import endpoints or schedule during off-peak hours. Alternatively, use CLAUDE CODE to write a rate limiter so sophisticated it could negotiate international trade agreements.

12.6 Claude Code for Audience Segmentation

Marketing teams describe segments in natural language. CLAUDE CODE translates them into precise SQL with proper null handling. This eliminates the traditional workflow of marketing writing a requirements doc, data engineering misinterpreting it, marketing saying "that's not what I meant," and everyone going home unhappy.

```
WITH current_state AS (
    SELECT *, DATEDIFF('day', last_active_date, CURRENT_DATE) AS
    days_since_active,
        DATEDIFF('day', signup_date, CURRENT_DATE) AS days_since_signup
    FROM marts.customer_360
)
-- Trial Power Users
SELECT customer_id, email, 'Trial Power Users' AS segment_name
FROM current_state
WHERE days_since_signup <= 30 AND active_days_last_30 >= 10
  AND lifecycle_stage IN ('trialing', 'free')
UNION ALL
-- Churn Risk
SELECT customer_id, email, 'Churn Risk'
FROM current_state
WHERE lifecycle_stage = 'paying'
  AND (health_score < 40 OR days_since_active > 14)
UNION ALL
```

```sql
-- Expansion Ready
SELECT customer_id, email, 'Expansion Ready'
FROM current_state
WHERE lifecycle_stage = 'paying' AND current_mrr < 500 AND health_score
    >= 80
```

Listing 12.5: Audience segmentation SQL (abbreviated)

12.7 Full Example: Customer Activation Pipeline

A complete pipeline using Dagster: refresh Customer 360 (dbt), run segmentation, detect changes, sync to Salesforce, sync segments to HubSpot, push alerts to Slack, and log results. Seven steps. What could possibly go wrong? (Do not answer that.)

```python
from dagster import asset, Output, MetadataValue, Definitions,
    ScheduleDefinition

@asset(group_name="reverse_etl")
def customer_360_model(context, dbt_resource):
    result = dbt_resource.cli(["build", "--select", "customer_360"])
    return Output(None, metadata={"refreshed_at": MetadataValue.text(
        datetime.utcnow().isoformat())})

@asset(group_name="reverse_etl", deps=["customer_360_model"])
def audience_segments(context, snowflake_resource):
    df = snowflake_resource.execute_query(SEGMENT_QUERY, fetch_pandas=
    True)
    return Output(df, metadata={
        "total_records": MetadataValue.int(len(df))})

@asset(group_name="reverse_etl", deps=["salesforce_change_set"])
def salesforce_sync(context, salesforce_change_set, salesforce_resource):

    if salesforce_change_set.empty:
        return Output({"inserted": 0, "updated": 0, "errors": 0})
    return Output(salesforce_resource.map_and_upsert(
        df=salesforce_change_set, object_name="Contact",
        external_id_field="Email", field_mapping=FIELD_MAPPING))

@asset(group_name="reverse_etl", deps=["audience_segments"])
def slack_alerts(context, audience_segments, slack_resource):
    for segment in ["Churn Risk", "VIP At Risk"]:
```

```
        segment_df = audience_segments[
            audience_segments["segment_name"] == segment]
        if not segment_df.empty:
            slack_resource.post_message(
                channel="#customer-alerts",
                text=f"*{segment}*: {len(segment_df)} customers
    identified")
```

Listing 12.6: Dagster assets for activation pipeline (core)

12.8 Operational Alerting System

Claude Code generates a flexible YAML-based alert configuration with warehouse queries, severity levels, and multi-channel dispatch (Slack, PagerDuty). Because if a revenue drop happens and nobody gets paged, did it really happen? (Yes. Yes it did. And someone will find out on Monday.)

```
alerts:
  - name: revenue_drop
    description: "Daily revenue dropped >20% vs 7-day avg"
    severity: critical
    query: |
      WITH daily AS (
        SELECT DATE(ordered_at) AS d, SUM(net_revenue) AS rev
        FROM analytics.fct_order_items
        WHERE ordered_at >= DATEADD('day', -30, CURRENT_DATE)
        GROUP BY 1
      ), comparison AS (
        SELECT d, rev, AVG(rev) OVER (
          ORDER BY d ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
        ) AS avg_7d FROM daily
      )
      SELECT d, rev, avg_7d FROM comparison
      WHERE d = CURRENT_DATE - 1 AND (rev - avg_7d)/NULLIF(avg_7d,0) <
    -0.20
    channels:
      - {type: slack, target: "#revenue-alerts"}
      - {type: pagerduty, target: "revenue-team"}
  - name: data_freshness
    description: "Customer 360 stale (>3 hours)"
    severity: critical
    query: |
```

```
      SELECT MAX(model_refreshed_at) AS last_refresh,
        DATEDIFF('minute', MAX(model_refreshed_at), CURRENT_TIMESTAMP())
      AS mins
      FROM marts.customer_360 HAVING mins > 180
    channels:
      - {type: slack, target: "#data-engineering"}
```

Listing 12.7: Alert definitions (excerpt)

12.9 Event-Driven Reverse ETL

Batch syncs work for many use cases, but some scenarios demand near-real-time delivery. Event-driven reverse ETL uses CDC or streaming to push updates within seconds. Because apparently hourly syncs are "too slow" now. We live in an era where a three-second delay in updating a Salesforce field is considered a production incident.

```
CREATE OR REPLACE STREAM analytics.marts.customer_360_stream
ON TABLE analytics.marts.customer_360
APPEND_ONLY = FALSE;

-- Query pending changes
SELECT METADATA$ACTION, METADATA$ISUPDATE,
    customer_id, email, health_score, churn_risk_bucket
FROM analytics.marts.customer_360_stream
LIMIT 100;
```

Listing 12.8: Creating a Snowflake Stream for reverse ETL

```
class SnowflakeStreamPoller:
    def __init__(self, stream_name):
        self.stream_name = stream_name
        self.conn = sf_connect(account=os.environ["SNOWFLAKE_ACCOUNT"],
            user=os.environ["SNOWFLAKE_USER"],
            password=os.environ["SNOWFLAKE_PASSWORD"])

    def consume_stream(self):
        cursor = self.conn.cursor()
        cursor.execute(f"""
            SELECT *, METADATA$ACTION AS _action,
```

```
                METADATA$ISUPDATE AS _is_update
            FROM {self.stream_name}""")
        columns = [desc[0] for desc in cursor.description]
        return [dict(zip(columns, row)) for row in cursor]

class KafkaChangePublisher:
    def __init__(self, bootstrap_servers, topic):
        self.topic = topic
        self.producer = Producer({"bootstrap.servers": bootstrap_servers
    })

    def publish_changes(self, changes):
        for change in changes:
            key = str(change.get("CUSTOMER_ID", "unknown"))
            event = {"action": change.get("_ACTION", "INSERT"),
                     "data": {k: v for k, v in change.items()
                              if not k.startswith("_")}}
            self.producer.produce(self.topic, key=key.encode(),
                value=json.dumps(event, default=str).encode())
        self.producer.flush()
```

Listing 12.9: Stream poller publishing to Kafka

Warning

Snowflake Streams have a data retention period (default 14 days). If your consumer falls behind, the stream becomes stale and must be recreated. Monitor stream lag proactively. Discovering your stream expired is like discovering your gym membership lapsed—except with more data loss and fewer excuses.

12.10 Monitoring and Observability

Key metrics to track: sync latency, record throughput, error rate, data freshness, and API quota utilization. Create a sync logging table and freshness monitoring view in the warehouse. If you are not monitoring your reverse ETL, you are not doing reverse ETL—you are doing reverse ETL and hoping for the best, which is a different discipline entirely.

```
CREATE OR REPLACE VIEW analytics.ops.sync_freshness AS
SELECT sync_name, destination,
    MAX(completed_at) AS last_success,
```

```
    DATEDIFF('minute', MAX(completed_at), CURRENT_TIMESTAMP()) AS
    staleness_minutes,
    SUM(CASE WHEN status = 'failed' THEN 1 ELSE 0 END)::FLOAT
        / NULLIF(COUNT(*), 0) AS failure_rate_7d
FROM analytics.ops.reverse_etl_sync_log
WHERE started_at >= DATEADD('day', -7, CURRENT_DATE)
GROUP BY 1, 2;
```

Listing 12.10: Sync freshness monitoring view

Warning

Monitor API quota utilization for every destination. A runaway sync that exhausts API quota can disrupt operations across your entire organization. There is no Slack message more terrifying than "Why can't anyone log into Salesforce?" followed by the realization that your sync consumed all the API calls for the day.

12.11 Exercises

1. **Design a Reverse ETL Pipeline.** Generate a dbt health score model, field mapping, change detection query, and monitoring query for syncing to Salesforce. Then use Claude Code to generate the apology email template for when the first sync overwrites everyone's contact notes.

2. **Audience Segmentation Challenge.** Translate five plain-English segment definitions into SQL. Handle the case where a customer qualifies for multiple segments by assigning highest-priority only. Use Claude Code to also generate the segment names, because marketing will reject your first five suggestions anyway.

3. **Multi-Destination Sync.** Build a configuration-driven sync engine that reads YAML mapping files and routes records to Salesforce, HubSpot, and Intercom simultaneously. Contemplate the existential implications of being responsible for data consistency across three SaaS platforms.

4. **Alert Engineering.** Write alert definitions for: signup volume spike ($> 3\sigma$), payment failure rate $> 10\%$ in the last hour, and feature adoption rate dropping below 5%. Use Claude Code to also write the PagerDuty escalation policy, because someone has to get woken up at 3 AM and it should not always be you.

13

Machine Learning Feature Engineering

"Garbage in, garbage out" is the polite version. The full version is "garbage in, a model that confidently predicts garbage, a dashboard that visualizes garbage, and an executive who makes decisions based on garbage."

Every ML model is only as good as its features. Raw data must be transformed into meaningful numerical representations before an algorithm can learn. This transformation—***feature engineering***—is one of the most time-consuming stages of the ML lifecycle, and also the stage most likely to be described as "almost done" for six consecutive sprints. Claude Code changes the economics: instead of spending hours brainstorming and hand-coding, describe the business domain and schema to Claude Code and receive ranked feature candidates with ready-to-run code.

In this chapter we cover feature engineering fundamentals, ***feature store*** architecture (Feast and Tecton), and how Claude Code accelerates feature discovery, generation, embedding management, documentation, and time-series work. Essentially, Claude Code does the tedious parts so you can focus on the glamorous work of explaining to stakeholders why the model's predictions are not magic.

13.1 Feature Engineering Fundamentals

A ***feature*** is a measurable property of an observed phenomenon. Features fall into categories: raw (taken directly from source), derived (computed from columns),

aggregate (summaries over windows), embedding (dense vectors from deep learning), and interaction (combinations of features).

Tip

Before productionizing notebook feature logic, always ask: "What entity does this feature describe, and what is its time granularity?" These two questions prevent most join-key and temporal-leakage bugs. They also prevent the classic "it works in the notebook" followed by "it produces nonsense in production" arc that we have all lived through at least twice.

Common Pitfalls

Pitfall	Description
Target leakage	Including information unavailable at prediction time.
Train/serve skew	Features computed differently in training vs. serving.
Entity misalignment	Joining features at the wrong granularity.
Stale features	Serving values computed hours or days ago.
Feature explosion	Thousands of features without pruning.

Warning

Target leakage is the most common source of "too good to be true" performance. When Claude Code generates feature candidates, verify each one: "Would this value be known at prediction time in production?" If your churn model has 99.7% accuracy, it is not because you are brilliant. It is because you leaked the target.

13.2 Feature Stores: Concepts and Architecture

A ***feature store*** manages the lifecycle of features: computation, storage, serving, monitoring, and discovery. It provides a registry (catalog of definitions), an offline store (warehouse for training), an online store (key-value DB for inference), materialization pipelines, and a serving API. It is, in effect, a warehouse for your warehouse's data, which is exactly the kind of recursion that keeps data engineers employed.

Feast vs. Tecton

Criterion	Feast	Tecton
Deployment	Self-hosted	Managed SaaS
Streaming features	Limited	Native support
Monitoring	Community plugins	Built-in
Best for	Teams with strong infra skills	Teams wanting managed ML infra

Note

Feast and Tecton are complementary. Feast is ideal when you want full infrastructure control and enjoy debugging Kubernetes at 2 AM. Tecton is better for fully managed platforms with streaming support and the budget to prove it.

13.3 Using Claude Code for Feature Discovery

The most effective prompt pattern has four components: business context (what you are predicting), schema description, constraints (what data is available at prediction time), and output format (feature name, SQL, rationale, importance, leakage risk). In other words, give Claude Code the same context you would give a new hire, except Claude Code will not spend the first week setting up its development environment.

```
I am building a churn prediction model for a B2B SaaS product.
Schema:
- users (user_id, company_id, created_at, plan_type, role)
- events (event_id, user_id, event_type, timestamp)
- subscriptions (sub_id, company_id, plan, mrr, start_date, end_date)
- support_tickets (ticket_id, company_id, created_at, priority,
                   resolved_at, satisfaction_score)

Target: will this company churn in the next 30 days?
Only data available up to the prediction date can be used.

Generate 20 candidate features with: name, SQL, rationale,
expected importance (high/medium/low), leakage risk.
```

Claude Code's output includes features like `active_users_ratio_30d` (high importance, no leakage), `support_tickets_open_14d` (high), `days_since_last_login` (high), and `mrr_change_pct_90d` (medium). After the initial round, use follow-up prompts to generate exact SQL, unit tests, and edge cases for the top candidates. It is like pair programming with someone who has read every feature engineering blog post ever written.

13.4 Automated Feature Generation

```
from datetime import timedelta
from feast import BigQuerySource, Entity, FeatureView, Field, ValueType
from feast.types import Float64, Int64

customer = Entity(name="customer", join_keys=["customer_id"],
    value_type=ValueType.STRING)

customer_features_source = BigQuerySource(
    name="customer_features_daily",
    table="analytics.customer_features_daily",
    timestamp_field="feature_date",
)

customer_feature_view = FeatureView(
    name="customer_features_daily",
    entities=[customer],
    ttl=timedelta(days=2),
    schema=[
        Field(name="total_orders_30d", dtype=Int64,
              description="Orders placed in last 30 days"),
        Field(name="avg_order_value_30d", dtype=Float64,
              description="Mean order value (USD) last 30 days"),
        Field(name="days_since_last_order", dtype=Int64,
              description="Days since most recent order"),
        Field(name="total_revenue_lifetime", dtype=Float64,
              description="Cumulative customer revenue"),
        Field(name="support_tickets_open", dtype=Int64,
              description="Currently open support tickets"),
    ],
    source=customer_features_source,
    online=True,
    tags={"team": "growth", "refresh_cadence": "daily"},
)
```

Listing 13.1: Feast FeatureView definition generated by Claude Code

```
WITH order_stats AS (
    SELECT customer_id, CURRENT_DATE() AS feature_date,
        COUNT(*) AS total_orders_30d,
        AVG(order_total) AS avg_order_value_30d,
        DATE_DIFF(CURRENT_DATE(), MAX(order_date), DAY)
            AS days_since_last_order
    FROM `warehouse.orders`
    WHERE order_date >= DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY)
      AND order_status IN ('completed', 'shipped')
    GROUP BY customer_id
),
lifetime AS (
    SELECT customer_id, SUM(order_total) AS total_revenue_lifetime
    FROM `warehouse.orders`
    WHERE order_status IN ('completed', 'shipped')
    GROUP BY customer_id
),
support AS (
    SELECT customer_id, COUNTIF(status = 'open') AS support_tickets_open
    FROM `warehouse.support_tickets` GROUP BY customer_id
)
SELECT c.customer_id, CURRENT_TIMESTAMP() AS feature_date,
    COALESCE(o.total_orders_30d, 0),
    COALESCE(o.avg_order_value_30d, 0.0),
    COALESCE(o.days_since_last_order, 9999),
    COALESCE(l.total_revenue_lifetime, 0.0),
    COALESCE(s.support_tickets_open, 0)
FROM `warehouse.customers` c
LEFT JOIN order_stats o USING (customer_id)
LEFT JOIN lifetime l USING (customer_id)
LEFT JOIN support s USING (customer_id);
```

Listing 13.2: Feature computation SQL generated by Claude Code

Tip

When asking Claude Code to generate Feast definitions, always specify the Feast version. The API changed significantly between 0.24 and 0.30+. Feast's changelog reads like a thriller novel where the protagonist keeps changing their

name and appearance between chapters.

13.5 Embedding Generation and Management

Embeddings are dense vector representations capturing semantic meaning—essential for text, images, user behavior, and high-cardinality categoricals. They are also the answer to the question "how do I turn words into numbers without anyone understanding how?"

```python
from sentence_transformers import SentenceTransformer
import numpy as np
import pandas as pd

def generate_embeddings(texts: list[str], model_name="all-MiniLM-L6-v2"):
    model = SentenceTransformer(model_name)
    return model.encode(texts, batch_size=256, normalize_embeddings=True)

def embeddings_to_dataframe(product_ids, embeddings):
    cols = {f"emb_{i}": embeddings[:, i] for i in range(embeddings.shape[1])}
    df = pd.DataFrame(cols)
    df["product_id"] = product_ids
    df["event_timestamp"] = pd.Timestamp.now(tz="UTC")
    return df
```

Listing 13.3: Embedding generation pipeline

Tip

Always store embedding model metadata (model name, dimensionality, normalization method) alongside the vectors. This prevents silent dimension mismatches when models are swapped. There is no debugging experience quite as soul-crushing as discovering your 384-dimensional vectors were being compared to 768-dimensional vectors for the past month.

13.6 Time-Series Feature Engineering

Time-series features include rolling aggregates, lag features, difference features, calendar features, Fourier features, and exponential smoothing. If that sounds like a lot, it is. Use Claude Code to generate them all at once. You were not put on this earth to manually write rolling window aggregates.

```python
def add_rolling_features(df, value_col, entity_col,
                         windows=(7, 14, 30), aggs=("mean", "std")):
    df = df.sort_values([entity_col, "date"])
    grouped = df.groupby(entity_col)[value_col]
    for window in windows:
        rolling = grouped.rolling(window=window, min_periods=1)
        for agg in aggs:
            df[f"{value_col}_rolling_{agg}_{window}d"] = \
                getattr(rolling, agg)().reset_index(level=0, drop=True)
    return df

def add_lag_features(df, value_col, entity_col, lags=(1, 7, 30)):
    df = df.sort_values([entity_col, "date"])
    for lag in lags:
        df[f"{value_col}_lag_{lag}d"] = \
            df.groupby(entity_col)[value_col].shift(lag)
    return df

def add_calendar_features(df, date_col="date"):
    dt = pd.to_datetime(df[date_col])
    df["day_of_week"] = dt.dt.dayofweek
    df["month"] = dt.dt.month
    df["is_weekend"] = (dt.dt.dayofweek >= 5).astype(int)
    return df
```

Listing 13.4: Time-series feature utilities

Warning

A common mistake is computing a rolling average that includes the current row's value. If the current value correlates with the target, this introduces leakage. Always shift the window by one period. This is the kind of bug that lets you publish a paper about your amazing model performance before the retraction.

13.7 Full Example: Building a Feature Store with Feast

We build a complete feature store for e-commerce serving churn prediction and recommendation models, with Claude Code assisting at every stage. By "assisting" we mean "doing approximately 90% of the work while we provide the domain context and take credit in the architecture review."

Project Structure and Configuration

```
# feature_store.yaml
project: ecommerce_features
registry: data/registry.db
provider: local
online_store:
  type: redis
  connection_string: "redis://localhost:6379"
offline_store:
  type: file
```

Feature Services and Retrieval

```
from feast import FeatureStore, FeatureService

churn_service = FeatureService(
    name="churn_prediction",
    features=[customer_engagement_fv, customer_support_fv],
    description="All features for churn prediction model v3.",
)

# Historical retrieval for training
store = FeatureStore(repo_path="feature_repo/")
training_df = store.get_historical_features(
    entity_df=entity_df,
    features=store.get_feature_service("churn_prediction"),
).to_df()

# Online retrieval for inference
feature_vector = store.get_online_features(
    features=store.get_feature_service("churn_prediction"),
```

```
    entity_rows=[{"customer_id": "C001"}],
).to_dict()
```

Listing 13.5: Feature service and training retrieval

Freshness Validation

```
def validate_freshness(store, max_staleness_hours=48):
    errors = []
    cutoff = datetime.now(timezone.utc) - timedelta(hours=
    max_staleness_hours)
    for fv in store.list_feature_views():
        if fv.materialization_intervals:
            latest = max(i.end_date for i in fv.
    materialization_intervals)
            if latest < cutoff:
                errors.append(f"'{fv.name}' last materialized at {latest}
    ")
        else:
            errors.append(f"'{fv.name}' has never been materialized.")
    return errors
```

Listing 13.6: Feature freshness validation

Tip

Schedule freshness validation immediately after materialization in your orchestration DAG. If it fails, alert the on-call engineer *before* the model consumes stale features. Serving a model with 48-hour-old features is like navigating with a two-day-old weather forecast—technically a forecast, practically useless.

13.8 Full Example: Feature Discovery for Recommendations

Recommendation systems are among the most feature-intensive ML applications. Here we use Claude Code to discover features for a hybrid recommender serving 50,000 products across 200 categories. The alternative is brainstorming features in a whiteboard session, which will produce exactly seven features before devolving into a debate about lunch.

User Features

```
USER_FEATURES = {
    "user_total_pageviews_7d": {
        "sql": """SELECT user_id, COUNT(*) AS user_total_pageviews_7d
            FROM page_views
            WHERE timestamp >= CURRENT_DATE - INTERVAL '7 days'
            GROUP BY user_id""",
        "rationale": "Active users have more diverse browsing patterns",
        "importance": "high",
    },
    "user_avg_purchase_price_90d": {
        "sql": """SELECT user_id,
            AVG(revenue / NULLIF(quantity, 0)) AS
    user_avg_purchase_price_90d
            FROM orders WHERE order_date >= CURRENT_DATE - INTERVAL '90
    days'
            GROUP BY user_id""",
        "rationale": "Price affinity filters out-of-range products",
        "importance": "high",
    },
    "user_cart_abandonment_rate_30d": {
        "sql": """WITH adds AS (
            SELECT user_id, COUNT(*) AS add_count FROM cart_events
            WHERE action = 'add' AND timestamp >= CURRENT_DATE -
    INTERVAL '30 days'
            GROUP BY user_id
        ), purchases AS (
            SELECT user_id, COUNT(DISTINCT product_id) AS buy_count FROM
     orders
            WHERE order_date >= CURRENT_DATE - INTERVAL '30 days'
            GROUP BY user_id
        )
        SELECT a.user_id, 1.0 - COALESCE(p.buy_count, 0)::FLOAT
            / NULLIF(a.add_count, 0) AS user_cart_abandonment_rate_30d
        FROM adds a LEFT JOIN purchases p USING (user_id)""",
        "rationale": "High abandonment signals price sensitivity",
        "importance": "medium",
    },
}
```

Listing 13.7: User-level features for recommendations

Item Features

```
ITEM_FEATURES = {
    "item_view_to_purchase_rate_30d": {
        "sql": """WITH views AS (
            SELECT product_id, COUNT(DISTINCT user_id) AS viewers
            FROM page_views WHERE timestamp >= CURRENT_DATE - INTERVAL
    '30 days'
            GROUP BY product_id
        ), buys AS (
            SELECT product_id, COUNT(DISTINCT user_id) AS buyers
            FROM orders WHERE order_date >= CURRENT_DATE - INTERVAL '30
    days'
            GROUP BY product_id
        )
        SELECT v.product_id, COALESCE(b.buyers, 0)::FLOAT
            / NULLIF(v.viewers, 0) AS item_view_to_purchase_rate_30d
        FROM views v LEFT JOIN buys b USING (product_id)""",
        "rationale": "High conversion items are strong recommendation
    candidates",
        "importance": "high",
    },
    "item_trending_score_7d": {
        "sql": """WITH curr AS (
            SELECT product_id, COUNT(*) AS views_7d FROM page_views
            WHERE timestamp >= CURRENT_DATE - INTERVAL '7 days' GROUP BY
     1
        ), prev AS (
            SELECT product_id, COUNT(*) AS views_prev FROM page_views
            WHERE timestamp BETWEEN CURRENT_DATE - INTERVAL '14 days'
                AND CURRENT_DATE - INTERVAL '7 days' GROUP BY 1
        )
        SELECT c.product_id, (c.views_7d - COALESCE(p.views_prev, 0))::
    FLOAT
            / NULLIF(COALESCE(p.views_prev, 1), 0) AS
    item_trending_score_7d
        FROM curr c LEFT JOIN prev p USING (product_id)""",
        "rationale": "Trending products capture emerging demand",
        "importance": "medium",
    },
}
```

Listing 13.8: Item-level features

Interaction Features

Key interaction features include `user_item_view_count_30d` (repeated views indicate interest), `user_item_in_cart` (current cart items are strong purchase signals), and `user_purchased_same_brand` (brand affinity drives repeats).

Note

In production, store credentials in a secrets manager, not in code. Claude Code can generate boilerplate for AWS Secrets Manager, HashiCorp Vault, or your orchestrator's secrets facility. It can also generate the boilerplate for rotating those secrets, which is the part everyone forgets until the credentials expire on a Friday afternoon.

13.9 Best Practices for Production Feature Pipelines

Idempotency

Every feature computation should be idempotent: running it twice with the same inputs produces the same outputs. Use the reference date as a partition key. If your feature pipeline is not idempotent, you are not doing feature engineering—you are doing feature roulette.

Schema Enforcement

Use Pandera or Great Expectations to enforce feature schemas at write time:

```
import pandera as pa

customer_schema = pa.DataFrameSchema({
    "customer_id": pa.Column(str, nullable=False, unique=True),
    "total_orders_30d": pa.Column(int, pa.Check.ge(0), nullable=False),
    "avg_order_value_30d": pa.Column(float, pa.Check.ge(0.0), nullable=
    False),
}, strict=True, coerce=True)
```

Drift Monitoring

Track feature distributions using Population Stability Index (PSI): < 0.1 = no shift, 0.1–0.2 = moderate, > 0.2 = significant. A PSI above 0.2 means either the world changed or your pipeline broke. Both possibilities are equally alarming.

```python
def compute_psi(reference, current, bins=10, epsilon=1e-4):
    bin_edges = np.percentile(reference, np.linspace(0, 100, bins + 1))
    bin_edges[0], bin_edges[-1] = -np.inf, np.inf
    ref_pct = np.histogram(reference, bins=bin_edges)[0] / len(reference) + epsilon
    cur_pct = np.histogram(current, bins=bin_edges)[0] / len(current) + epsilon
    return float(np.sum((cur_pct - ref_pct) * np.log(cur_pct / ref_pct)))
```

Listing 13.9: PSI computation

Warning

Never test feature pipelines only in isolation. The most dangerous bugs appear at boundaries between stages: type coercions, timezone mismatches, and fan-out joins. Integration tests on realistic sample data catch these. Unit tests catch the bugs you imagined. Integration tests catch the bugs that actually happen.

13.10 Exercises

1. **Feature Discovery.** For a fraud detection model with `transactions`, `users`, and `merchants` tables, use Claude Code to generate 15 candidate features with SQL, leakage assessment, and rationale. Bonus: ask Claude Code to also predict which features your stakeholder will reject for being "too complicated to explain."

2. **Time-Series Features.** Given hourly server metrics, write Python code for rolling, lag, difference, and Fourier features. Add Pandera schema validation and three edge-case unit tests. Use Claude Code to write the Fourier features because nobody remembers the Fourier transform after the exam.

3. **Feature Store Setup.** Set up a local Feast store with two entities, three feature views, a materialization script, and freshness validation. If you complete this in under two hours, you are either lying or have done this before.

4. **Drift Monitoring.** Implement PSI monitoring with Slack alerting when the threshold is exceeded. Verify the implementation returns 0 for identical distributions. Use Claude Code to also generate the Slack message, because "PSI exceeded threshold" will mean nothing to your on-call engineer at 3 AM.

Part V

Orchestration and Infrastructure

14

Orchestrating Data Pipelines with Claude Code

> *"Without an orchestrator, we had cron jobs. With an orchestrator, we have cron jobs that send Slack messages when they fail. This is what progress looks like."*

Modern data engineering demands robust orchestration: scheduling, dependency management, error handling, and monitoring across hundreds of interconnected tasks. In this chapter we explore how Claude Code accelerates every phase of orchestration across three dominant frameworks: Apache Airflow, Dagster, and Prefect. Sure, you could write this Terraform yourself. Wait, wrong chapter. Sure, you could hand-write all your DAGs yourself. You could also walk to work instead of driving. Both are technically possible and equally inadvisable.

14.1 Why Orchestration Matters

Orchestration coordinates discrete tasks according to a dependency graph and schedule: extraction, transformation, loading, validation, and notification. Without an orchestrator, teams resort to cron jobs and ad hoc retry logic—a pattern that collapses under hundreds of interdependent tasks and exactly one sick day from the person who understands the crontab.

The challenge is that each orchestrator imposes its own abstractions. Airflow has Operators, Sensors, and XComs; Dagster has Assets, Ops, and Resources; Prefect has Tasks, Flows, and Blocks. Claude Code collapses this learning curve by trans-

lating intent into idiomatic orchestrator code. It has memorized all three frameworks so you do not have to, which is fortunate because memorizing all three frameworks is how hobbies die.

A well-orchestrated data platform has three properties: **determinism** (given the same inputs, a pipeline produces the same outputs), **observability** (you can see what ran, when, and whether it succeeded), and **recoverability** (when something fails, you can retry from the failure point without reprocessing everything). Achieving all three simultaneously is referred to in the industry as "the dream."

Note

All three orchestrators are open-source and offer managed cloud products. Claude Code can help evaluate trade-offs between self-hosting and managed offerings based on team size, compliance requirements, and budget constraints. It cannot, however, help you win the internal political battle over which orchestrator to choose. That requires a different kind of intelligence.

14.2 Airflow DAG Generation

```
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.providers.slack.operators.slack_webhook import
    SlackWebhookOperator
from airflow.utils.trigger_rule import TriggerRule
import pendulum
from datetime import timedelta

default_args = {
    "owner": "data-engineering",
    "retries": 3,
    "retry_delay": timedelta(minutes=5),
    "retry_exponential_backoff": True,
    "execution_timeout": timedelta(hours=1),
    "sla": timedelta(hours=2),
}

TABLES = ["orders", "customers"]

@dag(dag_id="elt_postgres_to_snowflake",
     schedule="0 2 * * *",
     start_date=pendulum.datetime(2025, 1, 1, tz="UTC"),
```

```
     catchup=False, default_args=default_args,
     tags=["elt", "production"])
def elt_pipeline():

    @task(pool="postgres_pool", pool_slots=1)
    def extract_and_load(table: str, **context):
        # Extract from Postgres, stage into Snowflake
        pass

    dbt_build = BashOperator(
        task_id="dbt_build",
        bash_command="cd /opt/airflow/dbt && dbt build --select marts",
    )
    validate = BashOperator(
        task_id="validate",
        bash_command="great_expectations checkpoint run marts_checkpoint
    ",
    )
    notify_success = SlackWebhookOperator(
        task_id="notify_success",
        slack_webhook_conn_id="slack_data",
        message=":white_check_mark: ELT completed for {{ ds }}",
        trigger_rule=TriggerRule.ALL_SUCCESS,
    )
    notify_failure = SlackWebhookOperator(
        task_id="notify_failure",
        slack_webhook_conn_id="slack_data",
        message=":x: ELT FAILED for {{ ds }}",
        trigger_rule=TriggerRule.ONE_FAILED,
    )

    loaded = extract_and_load.expand(table=TABLES)
    loaded >> dbt_build >> validate >> [notify_success, notify_failure]

elt_pipeline()
```

Listing 14.1: Production Airflow DAG generated with Claude Code

Key production features: dynamic task mapping via `.expand()`, pool limiting to avoid overwhelming sources, exponential backoff on retries, and separate success/-failure notification paths.

Tip

When prompting Claude Code for Airflow DAGs, always specify your Airflow version. Airflow 2.x introduced the TaskFlow API, dynamic task mapping, and dataset-driven scheduling. Code written for Airflow 1.x will not work on 2.x and vice versa. Discovering this at deploy time is a rite of passage, but not a pleasant one.

Common Airflow Anti-Patterns

Claude Code can identify and fix common Airflow anti-patterns: top-level imports of heavy libraries (which slow DAG parsing), using `PythonOperator` where `@task` is cleaner, hardcoded connection strings instead of Airflow Connections, and monolithic DAGs that should be split into smaller, independently schedulable units. In other words, Claude Code can review your DAGs and diplomatically suggest that you rewrite most of them.

14.3 Dagster Asset Definitions

Dagster defines *software-defined assets*—the data artifacts your pipeline produces—and infers the execution graph from dependencies.

```
from dagster import (asset, AssetExecutionContext, MaterializeResult,
    MetadataValue, Definitions, ScheduleDefinition, define_asset_job,
    RetryPolicy, Backoff)

@asset(group_name="raw",
       retry_policy=RetryPolicy(max_retries=3, delay=30,
                                backoff=Backoff.EXPONENTIAL))
def raw_orders(context, snowflake):
    with snowflake.get_connection() as conn:
        row_count = conn.cursor().execute("COPY INTO raw.orders ...").
    fetchone()[0]
    return MaterializeResult(
        metadata={"row_count": MetadataValue.int(row_count)})

@asset(group_name="marts", deps=["stg_orders", "stg_customers"])
def order_metrics(context, snowflake):
    with snowflake.get_connection() as conn:
        conn.cursor().execute("""
            CREATE OR REPLACE TABLE marts.order_metrics AS
```

```
            SELECT c.region, DATE_TRUNC('month', o.order_date) AS month,
                COUNT(DISTINCT o.order_id) AS orders,
                SUM(o.total_amount) AS revenue
            FROM staging.orders o JOIN staging.customers c USING (customer_id)
            GROUP BY 1, 2""")
    return MaterializeResult(metadata={"model": MetadataValue.text("marts.order_metrics")})

defs = Definitions(
    assets=[raw_orders, order_metrics],
    jobs=[define_asset_job("nightly_elt")],
    schedules=[ScheduleDefinition(job_name="nightly_elt", cron_schedule="0 2 * * *")],
)
```

Listing 14.2: Dagster assets for an ELT pipeline

Dagster assets are plain Python functions that can be unit-tested without the orchestrator running. Dagster also natively supports freshness-based scheduling, where downstream assets are materialized when their upstream dependencies are refreshed. It is almost as if someone designed an orchestrator for the way data actually works. Revolutionary.

Note

Dagster vs. Airflow: When to Choose Which Choose Airflow when you need a battle-tested, widely-adopted orchestrator with a massive ecosystem of providers and a community large enough to answer your Stack Overflow questions within the hour. Choose Dagster when you want asset-based thinking, strong typing, and built-in data quality checks. Claude Code can translate between the two if you decide to migrate, which it will do with zero complaints—unlike your engineering team.

14.4 Prefect Flow Authoring

Prefect is the most Pythonic orchestrator. Flows and tasks are decorated functions with minimal boilerplate. If you can write a Python function, you can write a Prefect flow. If you cannot write a Python function, Claude Code can write both.

```
from prefect import flow, task, get_run_logger
```

```
from prefect.tasks import task_input_hash
from datetime import timedelta
import httpx

@task(retries=3, retry_delay_seconds=[30, 60, 120],
      cache_key_fn=task_input_hash, cache_expiration=timedelta(hours=1))
def extract_api_page(endpoint: str, page: int, api_key: str):
    resp = httpx.get(f"https://api.example.com/v2/{endpoint}",
        params={"page": page}, headers={"Authorization": f"Bearer {
    api_key}"})
    resp.raise_for_status()
    return resp.json()["results"]

@flow(name="incremental-api-ingest", retries=1, timeout_seconds=3600)
def incremental_ingest(endpoint="transactions", pages=10):
    from prefect.blocks.system import Secret
    api_key = Secret.load("example-api-key").get()
    page_futures = extract_api_page.map(
        endpoint=[endpoint]*pages, page=list(range(1, pages+1)),
        api_key=[api_key]*pages)
    all_records = []
    for future in page_futures:
        all_records.extend(future.result())
    load_to_snowflake(records=all_records, table=endpoint)
```

Listing 14.3: Prefect flow for API ingestion

Tip

Prefect Task Caching Notice the `cache_key_fn=task_input_hash` parameter. This caches task results based on inputs, so retrying a flow skips already-completed tasks. This is especially valuable for expensive extraction tasks where re-fetching data wastes time and API quota. It is also valuable for your sanity, which is a non-renewable resource.

14.5 Task Dependency Analysis

Claude Code can parse SQL queries to extract table-level lineage, building an adjacency list that can be fed into any orchestrator to generate correct task ordering. This is particularly valuable when migrating from ad hoc SQL scripts to an orchestrated pipeline—a process that archaeologists refer to as "discovering the de-

pendency graph that was there all along, hidden in 47 cron jobs and a shared Google Sheet."

```python
import re
from collections import defaultdict

def extract_table_lineage(sql: str):
    normalized = sql.strip().upper()
    targets = set()
    for p in [r"CREATE\s+(?:OR\s+REPLACE\s+)?TABLE\s+([\w.]+)\s+AS",
              r"INSERT\s+INTO\s+([\w.]+)"]:
        m = re.search(p, normalized)
        if m: targets.add(m.group(1).lower())
    sources = set()
    for p in [r"FROM\s+([\w.]+)", r"JOIN\s+([\w.]+)"]:
        for m in re.finditer(p, normalized):
            sources.add(m.group(1).lower())
    return targets, sources - targets

def build_dependency_graph(sql_files: dict[str, str]):
    graph = defaultdict(list)
    target_map = {}
    for name, sql in sql_files.items():
        targets, _ = extract_table_lineage(sql)
        for t in targets: target_map[t] = name
    for name, sql in sql_files.items():
        _, sources = extract_table_lineage(sql)
        for s in sources:
            graph[name].append(target_map.get(s, f"[external] {s}"))
    return dict(graph)
```

Listing 14.4: SQL-based dependency extractor

Warning

Regex Limitations for SQL Parsing The regex-based approach above handles simple SQL but misses CTEs, subqueries, and dynamic SQL. For production lineage extraction, use a proper SQL parser like `sqlglot` or ask Claude Code to generate a parser that handles your specific SQL dialect's edge cases. Using regex to parse SQL is like using a butter knife to perform surgery—it might work, but you will not be proud of the results.

14.6 Error Handling and Retry Logic

Claude Code helps implement layered error handling that distinguishes transient from permanent failures. Classify errors into TRANSIENT (retry with backoff), THROTTLED (retry after longer delay), PERMANENT (alert on-call), and DATA_QUALITY (route to data team). This prevents the common anti-pattern of retrying a permissions error seventeen times before giving up and paging someone who also cannot fix it.

```
from enum import Enum, auto
import re

class ErrorCategory(Enum):
    TRANSIENT = auto()
    THROTTLED = auto()
    PERMANENT = auto()
    DATA_QUALITY = auto()

def classify_error(exc):
    msg = str(exc).lower()
    if re.search(r"429|rate.limit|throttl", msg):
        return ErrorCategory.THROTTLED
    if re.search(r"timeout|connection.reset|503", msg):
        return ErrorCategory.TRANSIENT
    if re.search(r"not.null.constraint|duplicate.key", msg):
        return ErrorCategory.DATA_QUALITY
    return ErrorCategory.PERMANENT

RETRY_DELAYS = {
    ErrorCategory.TRANSIENT: [30, 60, 120],
    ErrorCategory.THROTTLED: [60, 300, 900],
    ErrorCategory.DATA_QUALITY: [],  # no retry
    ErrorCategory.PERMANENT: [],     # no retry
}
```

Listing 14.5: Error classification for retry decisions

Tip

Combine failure callbacks with `on_retry_callback` to track retry frequency by error category. This telemetry identifies flaky dependencies before they cause outages. A task that retries three times every night is a warning sign

even if it eventually succeeds. It is the pipeline equivalent of a check engine light that you have been ignoring for six months.

14.7 Monitoring and Alerting

Orchestrator-native UIs are a starting point, but production needs deeper observability. Claude Code generates Prometheus exporters and Grafana dashboards that go beyond the orchestrator's built-in metrics. Because if something fails and no dashboard shows it, you will find out from the CFO, and that is always worse.

```
from prometheus_client import start_http_server, Counter, Histogram,
    Gauge

TASK_DURATION = Histogram(
    "airflow_task_duration_seconds",
    "Task execution time",
    ["dag_id", "task_id"],
    buckets=[10, 30, 60, 300, 600, 1800, 3600],
)
TASK_FAILURES = Counter(
    "airflow_task_failures_total",
    "Total task failures",
    ["dag_id", "task_id"],
)
SLA_MISSES = Counter(
    "airflow_sla_misses_total",
    "Total SLA misses",
    ["dag_id"],
)
DAG_LAST_SUCCESS = Gauge(
    "airflow_dag_last_success_timestamp",
    "Timestamp of last successful DAG run",
    ["dag_id"],
)
```

Listing 14.6: Prometheus metrics for Airflow

The four essential dashboards for a data platform are: (1) Pipeline Health (success/-failure rates, SLA compliance), (2) Performance Trends (task durations over time, identifying degradation), (3) Resource Utilization (worker CPU, memory, queue depth), and (4) Data Freshness (time since last successful materialization of each

dataset). If you have fewer than four dashboards, you are flying blind. If you have more than forty, you are dashboard-hoarding and need an intervention.

> **Note**
>
> For Dagster, stream the event log to Kafka for custom metrics. For Prefect, use the API's run-level telemetry. Claude Code can transform either into Prometheus-compatible exporters with minimal custom code. It will also name your metrics sensibly, which is more than can be said for most humans.

14.8 Dynamic DAG Generation

The most robust dynamic pattern reads a metadata table at DAG parse time and generates tasks accordingly. Airflow 2.4+ also supports *datasets* as a scheduling primitive—a producer DAG declares an output dataset, and consumer DAGs trigger when it updates.

```
from airflow import Dataset
from airflow.decorators import dag, task
import pendulum

orders_dataset = Dataset("snowflake://analytics/staging/orders")

@dag(dag_id="produce_staging", schedule="0 2 * * *",
     start_date=pendulum.datetime(2025, 1, 1, tz="UTC"), catchup=False)
def produce():
    @task(outlets=[orders_dataset])
    def refresh_orders():
        pass
    refresh_orders()

produce()

@dag(dag_id="build_marts", schedule=[orders_dataset],
     start_date=pendulum.datetime(2025, 1, 1, tz="UTC"), catchup=False)
def consume():
    @task
    def build_metrics():
        pass
    build_metrics()

consume()
```

Listing 14.7: Dataset-triggered DAGs in Airflow 2.4+

Warning

Dynamically generated DAGs can confuse Airflow's serialization if the factory produces inconsistent DAG counts across restarts. Pin config files in version control and avoid reading from volatile sources at parse time. Test DAG generation in CI to catch serialization issues before deployment. Nothing is more humbling than a production outage caused by a YAML typo.

Configuration-Driven DAG Factories

For teams managing dozens of similar pipelines, a DAG factory reads YAML configuration and generates Airflow DAGs programmatically. Each YAML file specifies the source, destination, schedule, and any custom transformation logic. Claude Code can generate the factory code and the initial set of YAML configurations from your existing ad hoc pipelines. It can also generate the documentation explaining why you have a DAG factory, which you will need when someone new joins and asks "why don't we just write DAGs normally?"

14.9 Scheduling Optimization

Naive scheduling—running everything at midnight—causes resource contention. It is the data engineering equivalent of everyone trying to leave the parking garage at 5 PM. Claude Code generates analysis scripts that examine historical run data and recommend staggered schedules to minimize peak concurrency.

```
def analyze_schedule_conflicts(run_history: list[dict]) -> dict:
    """Analyze historical runs to find scheduling bottlenecks."""
    from collections import Counter
    hour_counts = Counter()
    for run in run_history:
        start_hour = run["start_time"].hour
        hour_counts[start_hour] += 1
    peak_hour = hour_counts.most_common(1)[0]
    return {
        "peak_hour": peak_hour[0],
```

```
        "peak_concurrent": peak_hour[1],
        "recommendation": "Stagger pipelines across 01:00-05:00 UTC"
            if peak_hour[1] > 10 else "Current schedule is acceptable",
    }
```

Listing 14.8: Schedule optimization analysis

Warning

Shifting a pipeline's schedule may break downstream consumers expecting fresh data by a certain hour. Always validate schedule changes against downstream SLA requirements. Document the dependency chain so that schedule changes propagate correctly. The phrase "I moved the pipeline to 4 AM and didn't tell anyone" has ended careers.

14.10 Migration Between Orchestrators

Claude Code can translate between frameworks. The key mappings: Airflow DAG → Dagster Job/Asset Group, Operator → Op/Asset, XCom → IO Manager, Connection → Resource, Sensor → Freshness Policy.

Migration is not just code translation—it requires rethinking how you model your pipelines. Airflow thinks in tasks and dependencies; Dagster thinks in assets and their relationships. Claude Code can analyze your Airflow DAGs and recommend which Dagster patterns (assets, ops, or a hybrid) best fit each pipeline. It approaches this with the enthusiasm of someone who does not have to sit through the six-month migration that follows.

Tip

When migrating, Claude Code recommends a parallel-run strategy: keep both old and new pipelines running, compare outputs, and decommission the old pipeline only after two weeks of consistent agreement. This approach catches subtle differences in scheduling, retry behavior, and error handling. It also doubles your infrastructure costs for two weeks, which is why you should not mention it to Finance until after it is done.

14.11 Putting It All Together

1. **Start with asset-based thinking.** Model pipelines as data products with clear inputs and outputs. Even if you use Airflow, think about what data each task produces.

2. **Parameterize everything.** Use DAG factories, YAML configs, or metadata tables. Never hardcode connection strings, table names, or schedules in DAG code.

3. **Layer error handling.** Combine native retries with error classification and intelligent alerting. Not every failure deserves a page. Most failures deserve a page even less than you think.

4. **Instrument aggressively.** Export metrics to Prometheus; the orchestrator UI is for development, not production monitoring.

5. **Optimize schedules.** Analyze historical run data to spread load and meet SLAs.

6. **Plan for migration.** Write modular logic loosely coupled to your orchestrator's API. Business logic should live in pure Python functions, not in operator callbacks. Future You will either thank you or inherit your technical debt. Be kind to Future You.

Note

Orchestrator Lock-In The biggest risk in orchestration is coupling business logic to framework-specific APIs. Keep your core transformation logic in plain Python modules that the orchestrator calls but does not define. This makes migration between frameworks a rewiring exercise rather than a rewrite. It also means Claude Code can rewrite the orchestration layer without touching your business logic, which is the kind of modularity that brings a tear to an architect's eye.

14.12 Exercises

1. **Multi-Framework Implementation**: Take a simple ELT pipeline (extract from an API, load to a database, run a transformation) and implement it in all three frameworks: Airflow, Dagster, and Prefect. Compare the code volume, testing experience, and debugging workflow. Use Claude Code to generate

all three, then argue with your team about which one is "best" for the next two sprints.

2. **Error Handling Drill**: Create a pipeline with intentional failure points (a task that fails 30% of the time, a task with a 10-second timeout that occasionally takes 15 seconds). Implement the error classification and retry logic from this chapter. Measure how many runs complete successfully with and without intelligent retry logic. Contemplate how this mirrors life.

3. **Dynamic DAG Factory**: Build a YAML-driven DAG factory that generates Airflow DAGs from configuration files. Start with three similar pipelines, then add a fourth by writing only a YAML file. Verify that all DAGs serialize correctly. Experience the quiet satisfaction of never writing another DAG by hand.

4. **Monitoring Dashboard**: Set up Prometheus metrics for your orchestrator and build a Grafana dashboard with the four essential panels described in Section 14.7. Simulate failures and verify that alerts fire correctly. Use Claude Code to also generate the runbook for when those alerts fire, because alerts without runbooks are just expensive anxiety.

5. **Orchestrator Migration**: Take an existing Airflow DAG (from this chapter or your own) and migrate it to Dagster using Claude Code. Document every decision point where the migration required rethinking the pipeline structure, not just translating syntax. This exercise takes approximately three times longer than you estimate.

6. **Scheduling Optimization**: Collect (or simulate) two weeks of historical run data for at least 10 pipelines. Analyze the data to identify scheduling conflicts, peak concurrency windows, and SLA risks. Use Claude Code to recommend an optimized schedule and validate that no downstream SLAs are violated. Present the results to your team and watch them ignore the recommendations for six months before implementing them during an outage.

15

Infrastructure as Code with Claude Code

"Sure, you could write this Terraform yourself. You could also walk to work instead of driving. Both are technically possible and equally inadvisable."

Infrastructure as Code (IaC) expresses infrastructure in declarative or imperative configuration files that live in version control alongside application code. In this chapter we explore how Claude Code transforms the IaC workflow—from authoring Terraform modules and CloudFormation templates to reviewing plans, estimating costs, detecting drift, and hardening security posture—across the three dominant IaC frameworks: Terraform, AWS CloudFormation, and Pulumi. Because clicking through the AWS console is not "infrastructure as code," no matter how many screenshots you paste into Confluence.

15.1 IaC Fundamentals

Every IaC framework addresses three concerns:

1. **Desired-state declaration**: You describe *what* you want, and the framework determines *how* to reach that state. This is a beautiful abstraction right up until the framework decides to destroy and recreate your production database to "reach that state."

2. **State management**: The framework tracks which resources it has already created so that subsequent runs apply only the delta.

3. **Execution planning**: Before making changes, the framework shows you a plan of creates, updates, and destroys. Reading this plan carefully is the difference between a good day and an incident report.

Note

If you are new to IaC, think of it as the infrastructure equivalent of a database migration framework like Alembic or Flyway. Just as those tools track schema state and apply diffs, IaC tools track cloud resource state and apply diffs. Except when they do not, and then you learn about "state drift" in the most educational way possible.

Terraform (HashiCorp) uses HCL and is cloud-agnostic. **AWS CloudFormation** is AWS-native using JSON or YAML. **Pulumi** uses general-purpose programming languages (Python, TypeScript, Go) instead of DSLs.

Data platforms are infrastructure-heavy—object storage, data warehouses, compute clusters, networking, IAM, and monitoring. Managing all of this manually is untenable at scale. IaC brings reproducibility (recreate your entire platform from scratch), auditability (every change is a git commit), and speed (provision a new environment in minutes rather than days). It also brings the terrifying ability to delete your entire platform from scratch, which is why we have code review.

Warning

State File Security Terraform and Pulumi store sensitive information in their state files: database passwords, API keys, and resource ARNs. Never commit state files to version control. Use remote state backends (S3 + DynamoDB for Terraform, Pulumi Cloud for Pulumi) with encryption at rest and strict access controls. A state file in a public GitHub repo is not a hypothetical risk—it is a Tuesday on r/DevOps.

Tip

Start with Modules, Not Monoliths Structure your IaC as reusable modules from day one. A monolithic Terraform file with 500 resources becomes unmaintainable. Organize by concern: networking, storage, compute, IAM, monitoring. Claude Code can refactor a monolith into modules if you describe

the desired boundaries. It can also generate the modules from scratch, which is faster than refactoring and does not require apologizing to the person who wrote the monolith.

15.2 Using Claude Code to Generate Terraform Modules

When asking Claude Code to generate a Terraform module, specificity pays dividends. A prompt like "Create a Terraform module for an S3 bucket" gives almost no context and produces the infrastructure equivalent of a participation trophy. Instead, specify encryption, versioning, lifecycle rules, access logging, naming conventions, and tags.

A Complete S3 Data Lake Module

```
variable "bucket_name" {
  description = "Name of the S3 bucket for the data lake"
  type        = string
  validation {
    condition     = can(regex("^[a-z0-9][a-z0-9.-]{1,61}[a-z0-9]$",
                    var.bucket_name))
    error_message = "Bucket name must be 3-63 chars, lowercase."
  }
}

variable "environment" {
  description = "Deployment environment"
  type        = string
  default     = "production"
  validation {
    condition     = contains(["development", "staging", "production"],
                    var.environment)
    error_message = "Must be development, staging, or production."
  }
}

variable "cost_center" {
  description = "Cost center for billing allocation"
  type        = string
}

```

```
variable "vpc_endpoint_id" {
  description = "VPC endpoint ID for bucket policy restriction"
  type        = string
}

variable "logging_bucket_id" {
  description = "ID of the S3 bucket for access logging"
  type        = string
}
```

Listing 15.1: Terraform module: S3 data lake bucket (variables.tf)

```
locals {
  common_tags = {
    Environment = var.environment
    CostCenter  = var.cost_center
    ManagedBy   = "terraform"
  }
}

resource "aws_kms_key" "data_lake" {
  description             = "KMS key for ${var.bucket_name}"
  deletion_window_in_days = 30
  enable_key_rotation     = true
  tags                    = local.common_tags
}

resource "aws_s3_bucket" "data_lake" {
  bucket = var.bucket_name
  tags   = local.common_tags
}

resource "aws_s3_bucket_versioning" "data_lake" {
  bucket = aws_s3_bucket.data_lake.id
  versioning_configuration { status = "Enabled" }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "data_lake
    " {
  bucket = aws_s3_bucket.data_lake.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm     = "aws:kms"
      kms_master_key_id = aws_kms_key.data_lake.arn
```

```
    }
    bucket_key_enabled = true
  }
}

resource "aws_s3_bucket_lifecycle_configuration" "data_lake" {
  bucket = aws_s3_bucket.data_lake.id
  rule {
    id     = "intelligent-tiering"
    status = "Enabled"
    transition { days = 30; storage_class = "STANDARD_IA" }
    transition { days = 90; storage_class = "GLACIER" }
    noncurrent_version_expiration { noncurrent_days = 365 }
  }
}

resource "aws_s3_bucket_public_access_block" "data_lake" {
  bucket                  = aws_s3_bucket.data_lake.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_policy" "data_lake" {
  bucket = aws_s3_bucket.data_lake.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Sid       = "DenyNonVPCEndpoint"
      Effect    = "Deny"
      Principal = "*"
      Action    = "s3:*"
      Resource  = [aws_s3_bucket.data_lake.arn,
                   "${aws_s3_bucket.data_lake.arn}/*"]
      Condition = {
        StringNotEquals = { "aws:sourceVpce" = var.vpc_endpoint_id }
      }
    }]
  })
}
```

Listing 15.2: Terraform module: S3 data lake bucket (main.tf)

Tip

Always ask Claude Code to include `validation` blocks in your variables. These catch configuration errors at `terraform plan` time rather than during apply, when the feedback loop is much slower and the blast radius is larger. The difference between "invalid bucket name" at plan time and "invalid bucket name" during a production deploy is the difference between a typo and an incident.

Outputs and Documentation

A well-designed module exports outputs that downstream modules consume. Claude Code generates `outputs.tf` with descriptions, and can also produce a `README.md` with usage examples, input/output tables, and prerequisite documentation. It will write better documentation than you would, and it will not complain about writing it, which makes it better than approximately 100% of engineers.

```
output "bucket_id" {
  description = "The ID of the data lake S3 bucket"
  value       = aws_s3_bucket.data_lake.id
}

output "bucket_arn" {
  description = "The ARN of the data lake S3 bucket"
  value       = aws_s3_bucket.data_lake.arn
}

output "kms_key_arn" {
  description = "The ARN of the KMS key used for encryption"
  value       = aws_kms_key.data_lake.arn
}

output "kms_key_id" {
  description = "The ID of the KMS key used for encryption"
  value       = aws_kms_key.data_lake.id
}
```

Listing 15.3: Terraform module: S3 data lake bucket (outputs.tf)

15.3 Claude Code for Terraform Plan Review and Risk Assessment

Running `terraform plan` produces a textual diff of intended changes. For large refactors the plan can be hundreds of lines long. Claude Code can parse this output and produce a structured risk assessment in seconds. It reads Terraform plans faster than you read Slack messages, and with considerably more attention to detail.

```
terraform plan -out=plan.bin
terraform show plan.bin -no-color > plan.txt

claude --print "Review this Terraform plan for risks. Flag any
  destructive changes, security regressions, cost implications,
  and potential downtime. Rate overall risk as LOW, MEDIUM, or HIGH.
  $(cat plan.txt)"
```

Listing 15.4: Capturing and reviewing a Terraform plan

When reviewing a plan, Claude Code evaluates destructive operations (resources being destroyed or recreated), security regressions (wide CIDR ranges, `*` actions in IAM), cost implications (instance size changes), ordering risks, and state drift indicators.

Tip

For CI/CD integration, pipe the plan output to Claude Code as part of your pull request workflow. Claude Code can post its risk assessment as a PR comment, giving reviewers a structured summary before they read the raw plan diff. This is the only PR comment format that reviewers actually read.

Warning

Destructive Plan Changes Pay special attention when `terraform plan` shows `destroy` followed by `create` for a resource. This often means the resource will be recreated, which can cause downtime. Common triggers include changing a resource name, modifying an immutable attribute, or moving a resource between modules. The word “destroy” in a Terraform plan should trigger the same physiological response as a fire alarm.

15.4 Pulumi and Programmatic IaC with Claude Code

Pulumi uses general-purpose programming languages, so data engineers can define infrastructure in the same Python they use for pipelines. This eliminates the context switch between HCL and Python. It also means you can introduce bugs in your infrastructure using the same language you introduce bugs in your application code. Consistency is a virtue.

```python
import pulumi
import pulumi_aws as aws
from pulumi import Config

config = Config()
environment = config.require("environment")

common_tags = {"Environment": environment, "ManagedBy": "pulumi"}

data_lake_key = aws.kms.Key("data-lake-key",
    description=f"KMS key for {environment} data lake",
    enable_key_rotation=True, tags=common_tags)

zones = ["raw", "staging", "curated"]
buckets = {}

for zone in zones:
    bucket = aws.s3.BucketV2(f"data-lake-{zone}",
        bucket=f"company-data-lake-{zone}-{environment}",
        tags={**common_tags, "Zone": zone})

    aws.s3.BucketVersioningV2(f"data-lake-{zone}-versioning",
        bucket=bucket.id,
        versioning_configuration={"status": "Enabled"})

    aws.s3.BucketServerSideEncryptionConfigurationV2(
        f"data-lake-{zone}-encryption", bucket=bucket.id,
        rules=[{"apply_server_side_encryption_by_default": {
            "sse_algorithm": "aws:kms",
            "kms_master_key_id": data_lake_key.arn,
        }, "bucket_key_enabled": True}])

    aws.s3.BucketPublicAccessBlock(f"data-lake-{zone}-public-block",
        bucket=bucket.id, block_public_acls=True,
        block_public_policy=True, ignore_public_acls=True,
        restrict_public_buckets=True)
```

```python

    buckets[zone] = bucket

for zone, bucket in buckets.items():
    pulumi.export(f"{zone}_bucket_name", bucket.bucket)
    pulumi.export(f"{zone}_bucket_arn", bucket.arn)

pulumi.export("kms_key_arn", data_lake_key.arn)
```

Listing 15.5: Pulumi: Data lake infrastructure in Python

Note

Notice how the Python `for` loop eliminates the repetition you would see in Terraform. This is Pulumi's core advantage: general-purpose language constructs applied to infrastructure. Conditionals, loops, functions, and classes all work as expected. The disadvantage is that you can now write infinitely complex infrastructure code, which some engineers interpret as an invitation rather than a warning.

15.5 Security Review of Infrastructure Code

Misconfigured infrastructure is a leading cause of data breaches. Claude Code can serve as a systematic security reviewer, evaluating encryption at rest, encryption in transit, network exposure, IAM least privilege, public access, logging, backup, and secrets management. It reviews infrastructure code with the thoroughness of someone who has read every AWS security blog post and the paranoia of someone who has lived through a breach.

```hcl
# BEFORE -- overly permissive (Claude Code flags as HIGH severity)
resource "aws_iam_role_policy" "glue_job" {
  role = aws_iam_role.glue_job.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"; Action = "s3:*"; Resource = "*"
    }]
  })
}

# AFTER -- least privilege (Claude Code's recommended fix)
```

```
resource "aws_iam_role_policy" "glue_job" {
  role = aws_iam_role.glue_job.id
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid      = "ReadSourceBucket"
        Effect   = "Allow"
        Action   = ["s3:GetObject", "s3:ListBucket"]
        Resource = [aws_s3_bucket.raw.arn,
                    "${aws_s3_bucket.raw.arn}/events/*"]
      },
      {
        Sid      = "WriteTargetBucket"
        Effect   = "Allow"
        Action   = ["s3:PutObject", "s3:DeleteObject"]
        Resource = ["${aws_s3_bucket.curated.arn}/analytics/*"]
      },
      {
        Sid      = "UseEncryptionKey"
        Effect   = "Allow"
        Action   = ["kms:Decrypt", "kms:GenerateDataKey"]
        Resource = [aws_kms_key.data_lake.arn]
      }
    ]
  })
}
```

Listing 15.6: Before and after: Tightening a Glue job IAM policy

Warning

Overly permissive Glue job roles. Data engineers often grant Glue jobs `s3:*` on `*` to "get things working." This is one of the most common security anti-patterns in data platforms. It is also the most common thing you will find when you inherit someone else's infrastructure. Always scope S3 permissions to specific bucket ARNs and prefixes, and separate read permissions from write permissions. "I'll tighten the permissions later" is the data engineering equivalent of "I'll start going to the gym on Monday."

Automated Security Scanning in CI

Integrate Claude Code-powered security reviews into your CI pipeline alongside tools like `checkov`, `tfsec`, and `terrascan`. Static analysis tools catch known patterns (open security groups, unencrypted buckets), while Claude Code catches contextual issues (a policy that is technically valid but overly broad for its stated purpose).

Note

Defense in Depth No single security tool catches everything. Use a layered approach: `tfsec` for known misconfigurations, `checkov` for compliance frameworks (CIS, SOC2), and Claude Code for contextual review. The overlap between tools is a feature, not a bug—redundant detection reduces the chance of a security gap reaching production. Think of it as wearing a belt and suspenders. Both at once is not paranoid; it is prudent.

Tip

Security Review Checklist When asking Claude Code to review infrastructure code, provide context: "This Glue job reads from the raw zone and writes to the curated zone. Review the IAM policy for least privilege." Without context, Claude Code cannot assess whether permissions are appropriate for the workload. It is very good at security review, but it cannot read your mind. Yet.

15.6 Drift Detection and Cost Estimation

Configuration drift occurs when actual infrastructure diverges from the desired state in code. Terraform detects drift automatically when you run `terraform plan`. Claude Code can analyze unexpected plan changes and recommend whether to adopt the drift or revert to desired state.

Data platforms are particularly susceptible to drift: someone opens a port "temporarily" for debugging, a developer adds an inline IAM policy through the console, or auto-scaling adjusts instance counts. Each of these creates a discrepancy between what your code says and what actually exists. The word "temporarily" in infrastructure has a half-life of approximately infinity.

Note

To reduce drift, enforce a policy that *all* infrastructure changes go through Terraform, even emergency fixes. Use SCP (Service Control Policies) to restrict console access to read-only for production accounts. When emergency console changes are unavoidable, import them into Terraform state within 24 hours. Treat every console change as a small fire that needs to be put out before it spreads.

For cost estimation, ***Infracost*** estimates cloud costs from Terraform code. Claude Code can interpret Infracost output and suggest optimizations: spot instances for Spark executors (60–70% savings), S3 Intelligent Tiering, reserved capacity for always-on resources, right-sizing clusters, and VPC endpoints to reduce NAT gateway costs. Your cloud bill will thank you. Your cloud provider will not.

Table 15.1: Common Data Platform Cost Optimizations (a.k.a. "Where Did All the Money Go?")

Resource	Optimization	Typical Savings
EC2/EMR workers	Spot instances	60–70%
S3 storage	Intelligent Tiering	20–40%
Redshift/BigQuery	Reserved capacity	30–50%
NAT Gateway	VPC endpoints for S3/DynamoDB	50–80%
EBS volumes	GP3 instead of GP2	15–20%

Tip

Cost Tagging Strategy Tag every resource with `CostCenter`, `Team`, `Environment`, and `Pipeline`. Without tags, you cannot attribute costs to specific teams or workloads. Claude Code can enforce tagging by generating `default_tags` in your provider configuration. Untagged resources are the infrastructure equivalent of unmarked bills—impossible to trace and always suspicious.

```
infracost breakdown --path . --format json > cost.json

claude --print "Analyze this Infracost report and suggest
  optimizations that could reduce monthly costs by at least 20%
  without impacting data pipeline reliability.
  $(cat cost.json)"
```

Listing 15.7: Integrating Infracost with Claude Code review

15.7 Exercises

1. **Module Generation.** Ask Claude Code to generate a Terraform module for an Amazon Redshift Serverless namespace with customer-managed KMS encryption, VPC connectivity, a usage limit, and IAM authentication. Validate the output for completeness and security best practices. Run `terraform validate` and fix any issues. If Claude Code produces valid Terraform on the first try, consider what that means for your career trajectory.

2. **Plan Review Drill.** Take the Terraform plan output from a recent infrastructure change (or generate a synthetic one). Feed it to Claude Code and ask for a risk assessment. Compare findings against your own manual review. Note any risks that Claude Code caught that you missed, and vice versa. This exercise is humbling in both directions.

3. **Security Hardening.** Start with an intentionally insecure Terraform configuration (unencrypted S3 bucket, open security group, public RDS instance, wildcard IAM policy, disabled logging, no versioning). Ask Claude Code to identify and fix all security issues. A thorough review should find at least 10 distinct issues. If it finds fewer, your "insecure" configuration was not insecure enough, which is an unusual problem to have.

4. **Cost Optimization Challenge.** Provide Claude Code with a complete data lake infrastructure (S3 buckets, Glue jobs, Redshift cluster, EMR cluster, NAT gateways) and ask it to find ways to reduce monthly costs by at least 30% without sacrificing data durability or query performance. Implement the top three recommendations. Present the savings to management and bask in the brief, warm glow of executive approval.

5. **Pulumi Translation.** Take the Terraform S3 module from this chapter and translate it to Pulumi Python using Claude Code. Compare the two implementations in terms of readability, testability, and the ability to express conditional logic. Form strong opinions about which is better. Change those opinions within six months.

6. **Drift Detection Simulation.** Deploy a simple infrastructure (S3 bucket + IAM role) via Terraform. Make three manual changes through the AWS

console (add a tag, modify the bucket policy, attach an inline IAM policy). Run `terraform plan`, feed the output to Claude Code, and evaluate its recommendations for each drift item. Decide whether to adopt or revert each change. Practice saying "who made this console change?" in an appropriately disappointed tone.

16

DataOps and CI/CD

Modern data engineering demands the same rigor in deployment, testing, and operational excellence that software engineering has cultivated over decades. ***DataOps*** *brings DevOps principles into the data world, combining agile methodology, continuous integration, and continuous delivery to accelerate data pipeline development while maintaining quality and reliability. In this chapter, we explore how* Claude Code *can serve as a force multiplier across every stage of the DataOps lifecycle.*

16.1 DataOps Principles and Practices

DataOps rests on three pillars: ***agile data development*** (short feedback loops, frequent releases), ***continuous integration and delivery for data*** (every pipeline change passes through automated validation), and ***monitoring and observability*** (visibility into data quality, freshness, volume, and lineage).

The core insight of DataOps is that data pipelines are software. They deserve the same development practices: version control, code review, automated testing, staged deployments, and monitoring. Yet many data teams still deploy by running SQL scripts manually, skip testing because "it's just a query," and learn about failures from angry stakeholders rather than alerts.

Note

DataOps is often confused with ***data governance***, but they are complementary. Data governance defines the policies and standards; DataOps provides the practices and tooling to enforce them continuously. Think of governance as the "what" and DataOps as the "how."

Tip

Start your DataOps journey by mapping your current data pipeline deployment process end to end. Identify every manual step, every approval bottleneck, and every point where errors tend to occur. These are your highest-impact automation targets. Most teams find that 60–70% of deployment time is spent on manual validation that could be automated.

AI-assisted DataOps is about amplifying engineer effectiveness. Claude Code excels at the repetitive, pattern-heavy tasks that consume disproportionate time: writing tests, reviewing SQL for anti-patterns, generating migration scripts, drafting runbooks, and summarizing incident timelines. The goal is not to remove engineers from the loop but to reduce the toil that prevents them from doing higher-value work.

16.2 CI/CD for Data Pipelines

Data pipeline CI/CD differs from traditional software CI/CD in important ways: pipelines are stateful (they read from and write to persistent stores), schema changes can be destructive (dropping a column destroys data), and testing requires realistic data (synthetic data may not exercise important edge cases). A well-designed system includes linting and static analysis, unit testing, integration testing, data quality validation, performance benchmarking, deployment, and post-deployment validation.

Warning

Never skip the integration testing stage, even when changes appear trivial. A single-character typo in a SQL `JOIN` condition can silently produce incorrect results that propagate through downstream systems for days before anyone notices.

Each branch should map to a corresponding database schema or environment. When

a developer creates a feature branch, CI should automatically provision a sandboxed environment with isolated schemas, seed data, and the ability to run the full pipeline end to end.

Environment Isolation Strategies

There are three common approaches to environment isolation for data pipelines:

1. **Schema-per-branch**: Each branch gets a dedicated schema (e.g., `dev_feature_123`) in a shared database. Simple to implement, but no compute isolation.
2. **Database-per-branch**: Each branch gets a full database clone. Better isolation, but expensive for large databases.
3. **Zero-copy clones**: Supported by Snowflake, Databricks, and some PostgreSQL extensions. Full isolation with minimal storage overhead.

Claude Code can generate the provisioning and teardown scripts for any of these strategies, including the CI/CD hooks that create environments on branch creation and destroy them on merge.

16.3 Claude Code for Pull Request Reviews of Data Code

Claude Code is particularly effective at identifying join correctness issues, NULL handling errors, aggregation mistakes, type coercion problems, performance anti-patterns, schema compatibility issues, and business logic errors in data code. These are precisely the issues that are hardest to catch in manual code review because they require holding the entire data model in your head.

Code Example

```
import anthropic
import subprocess
import json
from pathlib import Path


def review_data_code(filepath: str) -> dict:
    """Perform a Claude Code-powered review of a data file."""
    client = anthropic.Anthropic()
```

```
    diff = subprocess.run(
        ["git", "diff", "main", "--", filepath],
        capture_output=True, text=True, check=True
    ).stdout

    full_content = Path(filepath).read_text()

    review_prompt = f"""You are a senior data engineer reviewing
a pull request. Review the following data code change.

## Changed File: {filepath}

### Full file content:
```sql
{full_content}
```

Diff:
```diff
{diff}
```

Review Checklist:
1. Join correctness 2. NULL handling
3. Performance   4. Data quality tests
5. Schema compat   6. Business logic
7. Incremental logic 8. Documentation

Provide your review as JSON with: "summary", "severity"
(approve/request_changes/comment), "findings" (list with
line, severity, category, message), "suggestions"."""

    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        messages=[{"role": "user", "content": review_prompt}]
)

    text = response.content[0].text
    start = text.find("{")
    end = text.rfind("}") + 1
    return json.loads(text[start:end])
```

Tip

Contextual Reviews For the best review quality, provide Claude Code with both the diff *and* the full file content. The diff shows what changed; the full file shows the context. Without context, Claude Code cannot assess whether a JOIN is correct or whether a WHERE clause is consistent with the rest of the query.

16.4 Automated Migration Generation and Validation

Database migrations are error-prone because they operate on live data with real consequences. A migration that works perfectly on an empty test database may lock a production table for hours. Claude Code can generate migration scripts that account for edge cases: default values for existing rows, index recreation, constraint validation, and rollback procedures.

Always validate migrations before execution—Claude Code can perform multi-point validation checking for data loss risk, reversibility, lock impact, idempotency, and transaction safety.

```
def validate_migration(forward_sql: str, rollback_sql: str) -> dict:
    client = anthropic.Anthropic()
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        messages=[{"role": "user", "content": f"""Validate this
database migration for safety.

Forward migration:
```sql
{forward_sql}
```

Rollback migration:
```sql
{rollback_sql}
```

Check: 1. Data loss risk 2. Reversibility
3. Lock impact on large tables 4. Idempotency
5. Transaction safety 6. Index impact
Return JSON with: safe (bool), risks (list), suggestions (list)."""}]
)
```

```
    text = response.content[0].text
    start = text.find("{")
    end = text.rfind("}") + 1
    return json.loads(text[start:end])
```

Listing 16.1: Migration validation with Claude Code

Warning

Never execute Claude Code-generated migrations directly in production without human review and testing in a staging environment. Even well-validated migrations can have unexpected interactions with concurrent workloads, replication, and database-specific behaviors.

16.5 Testing Strategies for Data Pipelines

Data pipeline testing spans three levels: unit tests (individual transformation logic), integration tests (components working together), and end-to-end tests (full pipeline validation against realistic data).

Code Example

```
import pytest
import pandas as pd
from decimal import Decimal
from datetime import date


class TestRevenueCalculation:
    """Unit tests for the daily revenue aggregation model."""

    @pytest.fixture
    def sample_transactions(self) -> pd.DataFrame:
        return pd.DataFrame({
            "transaction_id": [1, 2, 3, 4, 5, 6],
            "customer_id": [101, 102, 101, 103, 102, 101],
            "amount": [Decimal("99.99"), Decimal("149.50"),
                       Decimal("29.99"), Decimal("0.00"),
                       Decimal("199.99"), Decimal("-29.99")],
            "status": ["completed", "completed", "completed",
```

```
                        "failed", "completed", "refunded"],
            "transaction_date": [
                date(2025, 1, 15), date(2025, 1, 15),
                date(2025, 1, 15), date(2025, 1, 15),
                date(2025, 1, 16), date(2025, 1, 16)],
        })

    def test_daily_revenue_excludes_failed(self, sample_transactions)
    :
        result = calculate_daily_revenue(sample_transactions)
        jan_15 = result[result["date_day"] == date(2025, 1, 15)]
        assert jan_15["total_revenue"].iloc[0] == Decimal("279.48")

    def test_daily_revenue_handles_refunds(self, sample_transactions)
    :
        result = calculate_daily_revenue(sample_transactions)
        jan_16 = result[result["date_day"] == date(2025, 1, 16)]
        assert jan_16["total_revenue"].iloc[0] == Decimal("170.00")

    def test_daily_revenue_no_transactions(self):
        empty_df = pd.DataFrame(columns=[
            "transaction_id", "customer_id", "amount",
            "status", "transaction_date"])
        result = calculate_daily_revenue(empty_df)
        assert len(result) == 0
```

Integration tests validate that pipeline components work together, running against a real database with controlled test data. Use `testcontainers` to spin up temporary PostgreSQL containers for isolated testing.

Tip

Use Claude Code to generate comprehensive test fixtures. Describe your schema and business rules, and ask Claude Code to generate test data covering edge cases: NULL values, boundary conditions, Unicode characters, timezone edge cases, extreme values, and the specific scenarios that have caused production incidents in the past.

Note

The Testing Pyramid for Data Apply the testing pyramid to data pipelines: many unit tests (fast, cheap, isolated), fewer integration tests (slower, test component interaction), and a handful of end-to-end tests (slowest, test the full pipeline). Claude Code can generate tests at all three levels from a single description of your pipeline's expected behavior.

16.6 Deployment Validation and Incident Response

Every deployment should have a pre-deployment risk assessment, a corresponding rollback plan, and post-deployment validation. Claude Code can generate rollback scripts that account for the specific changes being deployed.

Post-deployment validation goes beyond "did it run?" to "did it produce correct results?" Common validation checks include row count comparisons (new run vs. previous run), distribution checks (no unexpected NULL spikes), freshness verification (data is as recent as expected), and cross-system consistency (totals match between source and target).

```python
def validate_deployment(table: str, conn) -> list[str]:
    """Run post-deployment validation and return any failures."""
    failures = []
    # Row count sanity check
    current = conn.execute(f"SELECT COUNT(*) FROM {table}").scalar()
    previous = conn.execute(
        f"SELECT row_count FROM pipeline_metadata "
        f"WHERE table_name = '{table}' ORDER BY run_at DESC LIMIT 1"
    ).scalar()
    if previous and current < previous * 0.9:
        failures.append(
            f"Row count dropped >10%: {previous} -> {current}")
    # NULL spike detection
    for col in get_required_columns(table):
        null_pct = conn.execute(
            f"SELECT COUNT(*) FILTER (WHERE {col} IS NULL) * 100.0 "
            f"/ COUNT(*) FROM {table}").scalar()
        if null_pct > 5:
            failures.append(f"NULL spike in {col}: {null_pct:.1f}%")
    return failures
```

Listing 16.2: Post-deployment validation checks

When data incidents occur, Claude Code can accelerate every phase: detection, diagnosis, resolution, and postmortem. It can analyze monitoring alerts, classify severity, identify likely root causes, and draft blameless postmortem documents from incident timelines.

Tip

Store postmortems in a version-controlled repository alongside your pipeline code. Over time, Claude Code can analyze your postmortem history to identify recurring patterns and suggest systemic improvements. Common patterns include missing monitoring for newly deployed pipelines and inadequate testing of schema changes.

Warning

Rollback Is Not Always Safe For pipelines that write to external systems (reverse ETL, API pushes, email triggers), rollback may not undo side effects. A rolled-back pipeline does not unsend emails or un-update CRM records. For these pipelines, deploy with extra caution and consider a shadow-mode testing phase.

16.7 Full Example: GitHub Actions Pipeline

The following GitHub Actions workflow demonstrates a complete CI/CD pipeline for a dbt-based data project. It includes linting, AI-powered review, integration testing, and gated production deployment.

Code Example

```
# .github/workflows/data-pipeline-ci.yaml
name: Data Pipeline CI/CD

on:
  pull_request:
    branches: [main, staging]
    paths: ['models/**', 'macros/**', 'tests/**']

jobs:
  lint:
```

```yaml
  name: SQL Linting
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - run: pip install sqlfluff sqlfluff-templater-dbt
    - run: sqlfluff lint models/ --dialect postgres
           --format github-annotation-native

claude-review:
  name: Claude Code AI Review
  runs-on: ubuntu-latest
  needs: lint
  permissions: { pull-requests: write, contents: read }
  steps:
    - uses: actions/checkout@v4
      with: { fetch-depth: 0 }
    - uses: actions/setup-python@v5
      with: { python-version: '3.11' }
    - run: pip install anthropic pyyaml
    - id: changed-files
      run: |
        FILES=$(git diff --name-only origin/${{ github.base_ref }}
          -- '*.sql' '*.yml' | tr '\n' ',')
        echo "files=${FILES}" >> "$GITHUB_OUTPUT"
    - if: steps.changed-files.outputs.files != ''
      env:
        ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
      run: python ci/scripts/claude_review.py
        --files "${{ steps.changed-files.outputs.files }}"
        --pr-number ${{ github.event.pull_request.number }}

integration-tests:
  name: Integration Tests
  runs-on: ubuntu-latest
  needs: [lint, claude-review]
  services:
    postgres:
      image: postgres:15
      env: { POSTGRES_USER: dbt_test,
             POSTGRES_PASSWORD: dbt_test,
             POSTGRES_DB: dbt_test }
      ports: ['5432:5432']
  steps:
    - uses: actions/checkout@v4
```

```
      - run: pip install dbt-postgres==1.7.*
      - run: dbt seed --target ci
      - run: dbt run --target ci
      - run: dbt test --target ci

  deploy-production:
    name: Deploy to Production
    runs-on: ubuntu-latest
    needs: integration-tests
    if: github.base_ref == 'main'
    environment: production
    steps:
      - uses: actions/checkout@v4
      - run: pip install dbt-postgres==1.7.*
      - run: python ci/scripts/create_snapshot.py
          --target production --version ${{ github.sha }}
      - run: dbt run --target production --fail-fast
      - run: python ci/scripts/post_deploy_validation.py
          --target production --version ${{ github.sha }}
```

Each stage serves a specific purpose: linting catches style and syntax issues early, the AI review catches semantic issues that linters miss, integration tests verify correctness against real data, and the production deploy includes both pre-deployment snapshots and post-deployment validation.

16.8 Exercises

Exercises

1. **CI/CD Pipeline Design** Design a CI/CD pipeline for a dbt project that includes at least five stages. Write the GitHub Actions YAML file and explain the purpose of each stage. Include at least one Claude Code-powered step. Test the pipeline against a sample dbt project with intentional issues.

2. **Migration Safety Analysis** You need to add a `NOT NULL` constraint to an existing column that contains some NULL values. Write the migration script (both up and down), including the data backfill step. Use Claude Code to validate its safety and identify edge cases you may have missed.

3. **Incident Response Runbook** Create a comprehensive incident response runbook for a data pipeline failure, including detection criteria, severity classifi-

cation, escalation paths, diagnostic procedures, resolution steps, communication templates, and a postmortem template. Use Claude Code to draft the runbook and then refine it based on your team's specific infrastructure.

4. **Review Automation** Implement a simplified Claude Code-powered PR review system that accepts a SQL file, sends it to Claude Code for review, and outputs structured findings. Test it against at least three SQL files with intentional issues (incorrect JOIN, NULL mishandling, missing WHERE clause, performance anti-pattern).

5. **Test Generation Challenge** Take an existing data transformation (a dbt model or SQL query) and use Claude Code to generate a comprehensive test suite. The suite should include unit tests, integration tests, and edge case tests. Measure code coverage and identify any scenarios that Claude Code missed.

6. **Post-Deployment Monitoring** Implement the post-deployment validation function from Section 16.6 for a table of your choice. Define thresholds for row count deviation, NULL spike detection, and freshness. Deploy a change that intentionally violates one threshold and verify that the validation catches it.

Part VI

Security, Governance, and Ethics

17

Data Governance with Claude Code

Tip

Data governance is not a one-time project—it is a continuous discipline. Claude Code can accelerate every facet of governance, from cataloging metadata to enforcing compliance policies, but human oversight remains essential for setting organizational priorities and interpreting regulatory nuance.

Modern data engineering teams operate under increasing pressure to treat data as a first-class organizational asset. Regulations such as GDPR, CCPA, and HIPAA impose strict requirements on how data is collected, stored, processed, and shared. Data governance provides the framework that bridges these demands—and Claude Code can serve as a tireless governance co-pilot.

This chapter covers four pillars of AI-augmented governance: automated data cataloging, lineage tracing and PII detection, compliance automation, and governance-as-code policy enforcement. Each section provides production-ready code and patterns you can adapt to your environment.

17.1 The Governance Challenge

Most organizations struggle with governance not because they lack policies, but because they lack the capacity to implement and enforce them consistently. A data team managing 500 tables, 200 dbt models, and 50 pipelines simply does not have

the bandwidth to manually classify every column, document every transformation, and audit every access pattern.

This is precisely where Claude Code changes the equation. The mechanical work of governance—scanning schemas, classifying columns, generating documentation, checking compliance—can be automated. The judgment work—setting policy, interpreting regulations, making trade-off decisions—remains with humans.

Note

Governance Maturity Organizations typically progress through governance maturity levels: (1) No formal governance—data is a free-for-all; (2) Reactive governance—policies exist but are enforced after incidents; (3) Proactive governance—policies are embedded in pipelines and CI/CD; (4) Automated governance—AI-augmented continuous monitoring and enforcement. This chapter helps you reach level 3–4.

The costs of poor governance are concrete:

- **Regulatory fines**: GDPR penalties can reach 4% of global annual turnover. CCPA violations carry fines of $2,500–$7,500 per violation.
- **Data breaches**: The average cost of a data breach exceeded $4.5 million in 2023, with healthcare breaches averaging nearly $11 million.
- **Lost productivity**: Data engineers spend an estimated 30–40% of their time on data quality issues, many of which stem from poor governance.
- **Decision risk**: Executives making decisions on ungoverned data face unreliable metrics, inconsistent definitions, and unreproducible results.

17.2 Automated Data Cataloging with Claude Code

A data catalog is the single source of truth for what data exists in your organization, who owns it, what it means, and how it should be handled. Claude Code can automate the heavy lifting by inspecting database schemas, reading source code, and generating human-readable descriptions.

```
import anthropic, psycopg2, json

def extract_schema_metadata(conn_string: str, schema: str = "public"):
    conn = psycopg2.connect(conn_string)
```

```
    cur = conn.cursor()
    cur.execute("""
        SELECT t.table_name, c.column_name, c.data_type,
               c.is_nullable, c.column_default,
               pgd.description AS column_comment
        FROM information_schema.tables t
        JOIN information_schema.columns c
            ON t.table_name = c.table_name
            AND t.table_schema = c.table_schema
        LEFT JOIN pg_catalog.pg_statio_all_tables st
            ON st.relname = t.table_name AND st.schemaname = t.
    table_schema
        LEFT JOIN pg_catalog.pg_description pgd
            ON pgd.objoid = st.relid AND pgd.objsubid = c.
    ordinal_position
        WHERE t.table_schema = %s AND t.table_type = 'BASE TABLE'
        ORDER BY t.table_name, c.ordinal_position
    """, (schema,))
    tables = {}
    for row in cur.fetchall():
        table_name = row[0]
        if table_name not in tables:
            tables[table_name] = {"columns": []}
        tables[table_name]["columns"].append({
            "name": row[1], "data_type": row[2],
            "nullable": row[3], "default": row[4],
            "existing_comment": row[5]})
    cur.close(); conn.close()
    return tables

def generate_catalog_descriptions(tables: dict) -> dict:
    client = anthropic.Anthropic()
    catalog = {}
    for table_name, meta in tables.items():
        prompt = f"""Given the table schema, generate:
1. A concise business description (2-3 sentences).
2. A business description for each column.
3. Suggested data classification (public/internal/confidential/
    restricted).
4. Suggested data owner role (e.g., "finance-team", "engineering").
5. Retention recommendation with justification.

Table: {table_name}
Columns: {json.dumps(meta['columns'], indent=2, default=str)}

Return valid JSON with keys: table_description, columns,
```

```
48 classification, suggested_owner, retention."""
49         response = client.messages.create(
50             model="claude-sonnet-4-20250514", max_tokens=2048,
51             messages=[{"role": "user", "content": prompt}])
52         catalog[table_name] = json.loads(response.content[0].text)
53     return catalog
```

Listing 17.1: Automated metadata extraction from PostgreSQL

Enriching Existing Catalogs

Most organizations already have a partial catalog—perhaps maintained in a spreadsheet, a wiki, or a commercial catalog tool. Claude Code can enrich these existing catalogs by identifying gaps (undocumented tables and columns), standardizing descriptions (making terminology consistent across domains), adding classifications (public, internal, confidential, restricted), and suggesting data owners based on table naming patterns and domain grouping.

```
1  def find_catalog_gaps(catalog: dict, schema_metadata: dict) -> dict:
2      """Compare existing catalog against live schema to find gaps."""
3      gaps = {
4          "undocumented_tables": [],
5          "undocumented_columns": [],
6          "stale_entries": [],
7          "missing_classifications": [],
8      }
9      # Tables in schema but not in catalog
10     for table in schema_metadata:
11         if table not in catalog:
12             gaps["undocumented_tables"].append(table)
13         else:
14             catalog_cols = {c["name"] for c in catalog[table].get("
       columns", [])}
15             schema_cols = {c["name"] for c in schema_metadata[table]["
       columns"]}
16             for col in schema_cols - catalog_cols:
17                 gaps["undocumented_columns"].append(f"{table}.{col}")
18             if not catalog[table].get("classification"):
19                 gaps["missing_classifications"].append(table)
20     # Tables in catalog but not in schema (stale)
21     for table in catalog:
22         if table not in schema_metadata:
23             gaps["stale_entries"].append(table)
```

```
    return gaps
```

Listing 17.2: Catalog gap analysis

Note

Treat your metadata catalog with the same rigor as production data. Version the schema with migrations, back it up regularly, and restrict write access to automated governance pipelines. A corrupt or outdated catalog is worse than no catalog because it creates false confidence.

Tip

Catalog Freshness Schedule catalog enrichment runs weekly. Compare the live schema against your catalog to detect new tables, dropped tables, and column changes. Claude Code can generate human-readable change summaries for review by data stewards.

17.3 Data Lineage Tracing and PII Detection

Data lineage answers: *Where did this data come from, and what happened to it?* Claude Code can parse SQL transformations, Airflow DAGs, and dbt models to reconstruct lineage graphs automatically.

SQL-Based Lineage Extraction

For teams using dbt, lineage is partially available through the manifest. But for custom SQL pipelines, stored procedures, and views, lineage must be extracted from the SQL itself.

```
def extract_lineage(sql: str, target_table: str) -> dict:
    """Extract column-level lineage from a SQL transformation."""
    client = anthropic.Anthropic()
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        system="""You are a data lineage expert. Given a SQL statement,
        extract column-level lineage. For each output column, identify:
        1. Source table(s) and column(s)
        2. Transformation applied (direct copy, aggregation,
```

```
            calculation, conditional, etc.)
        3. Whether the transformation preserves PII status""",
        messages=[{"role": "user",
            "content": f"Target table: {target_table}\n\nSQL:\n{sql}"}],
    )
    return json.loads(message.content[0].text)
```

Listing 17.3: SQL lineage extraction with Claude Code

Hybrid PII Detection

For PII detection, combine deterministic regex-based scanning with Claude Code's semantic analysis. Regex catches structured patterns (SSNs, emails, credit cards); Claude Code applies reasoning to detect context-dependent PII that regex misses—such as free-text fields containing addresses, columns named `notes` that contain patient information, or composite fields that become PII when combined.

```
import re, anthropic, json
from typing import List, Dict

PII_PATTERNS = {
    "ssn": re.compile(r"\b\d{3}-\d{2}-\d{4}\b"),
    "email": re.compile(
        r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"),
    "credit_card": re.compile(r"\b(?:\d{4}[-\s]?){3}\d{4}\b"),
    "phone_us": re.compile(
        r"\b(?:\+1[-.\s]?)?\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}\b"),
    "ip_address": re.compile(
        r"\b(?:\d{1,3}\.){3}\d{1,3}\b"),
}

def hybrid_pii_scan(table_name: str, columns: List[str],
                    sample_rows: List[Dict]) -> Dict:
    # Phase 1: Regex scan
    regex_findings = {}
    for row in sample_rows:
        for col in columns:
            val = str(row.get(col, ""))
            for name, pattern in PII_PATTERNS.items():
                if pattern.search(val):
                    regex_findings.setdefault(col, set()).add(name)

    # Phase 2: Claude Code semantic scan
```

```
    client = anthropic.Anthropic()
    prompt = f"""Examine sample rows from '{table_name}' and identify
PII values that regex might miss. Look for:
- Names, addresses, dates of birth in free-text fields
- Columns that become PII when combined (quasi-identifiers)
- Encoded or obfuscated PII
- Device identifiers, IP addresses, geolocation data

Return JSON array with column_name, detected_pattern,
pii_category, confidence, gdpr_special_category (boolean).

Sample data: {json.dumps(sample_rows[:5], indent=2, default=str)}"""
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=2048,
        messages=[{"role": "user", "content": prompt}])
    claude_findings = json.loads(response.content[0].text)

    # Phase 3: Merge
    merged = {}
    for col, patterns in regex_findings.items():
        merged[col] = {"method": "regex", "patterns": list(patterns)}
    for f in claude_findings:
        col = f["column_name"]
        if col in merged:
            merged[col]["method"] = "both"
        else:
            merged[col] = {"method": "claude", "category": f["
    pii_category"]}
    return merged
```

Listing 17.4: Hybrid PII detection pipeline

Warning

Never send real production PII to any external API unless your data processing agreement explicitly permits it. Scan metadata and column names at the schema level, or use synthetic/redacted samples. When in doubt, scan locally and send only aggregate findings (column names and pattern types, not actual values) to Claude Code for classification.

Tip

For GDPR compliance, distinguish between "regular" PII (name, email, address) and "special category" data (racial or ethnic origin, political opinions, religious beliefs, health data, sexual orientation). Special category data requires explicit consent and additional safeguards under Article 9.

17.4 Compliance Automation: GDPR, CCPA, and HIPAA

Compliance penalties are severe—up to 4% of annual turnover under GDPR, up to $7,500 per violation under CCPA, and up to $1.5 million per violation category per year under HIPAA. Claude Code can automate compliance checks, generate Data Subject Access Request (DSAR) queries, and produce Article 30 Records of Processing Activities (ROPA) from your data catalog.

DSAR Query Generation

When a data subject requests access to their personal data (a right under both GDPR and CCPA), you need to identify and extract all data associated with that individual across your entire platform. Claude Code can generate the comprehensive queries needed:

```python
def generate_dsar_queries(
    catalog: dict, subject_identifier: str, identifier_column: str
) -> list[str]:
    """Generate queries to extract all data for a data subject."""
    client = anthropic.Anthropic()
    tables_with_pii = {
        name: meta for name, meta in catalog.items()
        if meta.get("classification") in ("confidential", "restricted")
    }
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        system="""Generate SQL queries to extract all personal data
        for a data subject. Include:
        1. Direct matches on the identifier column
        2. Indirect matches through foreign key relationships
        3. Queries for audit/log tables that reference the subject
        Return as a JSON array of {table, query, data_category} objects.
    """,
        messages=[{"role": "user",
```

```
            "content": f"Catalog:\n{json.dumps(tables_with_pii, indent
    =2)}\n\n"
                       f"Subject identifier: {subject_identifier}\n"
                       f"Identifier column: {identifier_column}"}],
    )
    return json.loads(message.content[0].text)
```

Listing 17.5: DSAR query generator

Right to Erasure ("Right to Be Forgotten")

GDPR Article 17 grants data subjects the right to erasure. Implementing this requires knowing every location where their data exists—including backups, derived tables, and cached aggregations. Claude Code can generate an erasure plan:

```
-- Step 1: Delete from transactional tables
DELETE FROM orders WHERE user_id = :subject_id;
DELETE FROM order_items WHERE order_id IN (
    SELECT id FROM orders WHERE user_id = :subject_id
);

-- Step 2: Anonymize in analytical tables (preserve aggregates)
UPDATE analytics.user_metrics
SET email = 'redacted', name = 'redacted',
    ip_address = 'redacted'
WHERE user_id = :subject_id;

-- Step 3: Remove from search indexes
-- (requires application-level action)

-- Step 4: Log the erasure for compliance audit
INSERT INTO governance.erasure_log (
    subject_id, requested_at, completed_at, tables_affected
) VALUES (
    :subject_id, :request_timestamp, CURRENT_TIMESTAMP,
    'orders,order_items,analytics.user_metrics'
);
```

Listing 17.6: Erasure plan generated by Claude Code

Warning

Claude Code-generated compliance artifacts must be reviewed by your legal and privacy teams before submission to regulators or data subjects. AI-generated content does not constitute legal advice. This is especially critical for erasure operations, which are irreversible.

17.5 Governance Policies as Code

The most effective governance policies are enforced automatically. Express governance rules as code and integrate them into your CI/CD pipeline so that policy violations are caught before they reach production.

```
import psycopg2, json, sys
from dataclasses import dataclass, field
from typing import List, Optional
from enum import Enum

class PolicySeverity(Enum):
    BLOCK = "block"; WARN = "warn"; INFO = "info"

@dataclass
class GovernancePolicy:
    policy_id: str; name: str; description: str
    severity: PolicySeverity; check_sql: Optional[str] = None

POLICIES = [
    GovernancePolicy("GOV-001", "PII columns must be classified",
        "Every PII column must have a classification.",
        PolicySeverity.BLOCK, check_sql="""
            SELECT cc.column_name, dc.table_name
            FROM governance.column_catalog cc
            JOIN governance.data_catalog dc ON dc.catalog_id = cc.
    catalog_id
            WHERE cc.pii_flag = TRUE AND cc.classification IS NULL;"""),
    GovernancePolicy("GOV-002", "Tables must have a data owner",
        "Every table must have an assigned data_owner.",
        PolicySeverity.WARN, check_sql="""
            SELECT table_schema, table_name
            FROM governance.data_dictionary
            WHERE data_owner IS NULL GROUP BY 1, 2;"""),
    GovernancePolicy("GOV-003", "No unencrypted PII in staging",
```

```
        "PII columns in staging must use encryption or masking.",
        PolicySeverity.BLOCK, check_sql="""
            SELECT table_name, column_name
            FROM governance.column_catalog
            WHERE pii_flag = TRUE AND schema_name = 'staging'
            AND encryption_status = 'none'
            AND masking_policy IS NULL;"""),
    GovernancePolicy("GOV-004", "Retention policy required",
        "Every table must have a defined retention period.",
        PolicySeverity.WARN, check_sql="""
            SELECT table_schema, table_name
            FROM governance.data_dictionary
            WHERE retention_days IS NULL;"""),
]

class PolicyEngine:
    def __init__(self, conn_string: str, policies: list):
        self.conn = psycopg2.connect(conn_string)
        self.policies = policies

    def run_all_checks(self) -> bool:
        all_passed = True
        cur = self.conn.cursor()
        for policy in self.policies:
            if not policy.check_sql: continue
            cur.execute(policy.check_sql)
            violations = cur.fetchall()
            if violations:
                marker = "BLOCK" if policy.severity == PolicySeverity.
    BLOCK else "WARN"
                print(f"[{marker}] {policy.policy_id}: {policy.name} "
                      f"- {len(violations)} violation(s)")
                for v in violations[:5]:
                    print(f"  -> {v}")
                if len(violations) > 5:
                    print(f"  ... and {len(violations) - 5} more")
                if policy.severity == PolicySeverity.BLOCK:
                    all_passed = False
        cur.close()
        return all_passed

if __name__ == "__main__":
    engine = PolicyEngine("postgresql://gov:pass@localhost/warehouse",
    POLICIES)
    if not engine.run_all_checks():
        print("\nGovernance check FAILED. Pipeline blocked.")
```

```
        sys.exit(1)
    print("\nAll governance checks passed.")
```

Listing 17.7: Policy enforcement engine

Integrating Governance into CI/CD

The policy engine becomes most valuable when integrated into your CI/CD pipeline. Add it as a step in your GitHub Actions or GitLab CI configuration:

```yaml
# .github/workflows/governance.yml
name: Governance Checks
on:
  pull_request:
    paths:
      - 'dbt/**'
      - 'airflow/**'
      - 'migrations/**'

jobs:
  governance:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run governance policy checks
        run: python governance/policy_engine.py
        env:
          GOVERNANCE_DB_URL: ${{ secrets.GOVERNANCE_DB_URL }}

      - name: Run PII scan on changed models
        run: |
          python governance/pii_scanner.py \
            --changed-files $(git diff --name-only HEAD~1)

      - name: Verify catalog coverage
        run: python governance/catalog_coverage.py --threshold 90
```

Listing 17.8: Governance check in GitHub Actions

Tip

Version your governance policies alongside your data pipeline code. Use pull requests for policy changes so that stakeholders can review modifications before they take effect. This creates an audit trail showing who approved each policy change and when.

Note

Policy Evolution Start with a small set of high-impact policies (PII classification, data ownership) and expand gradually. Introducing 50 policies on day one overwhelms teams and creates resistance. A phased rollout with clear communication about why each policy matters drives better adoption.

17.6 Data Quality as a Governance Concern

Data quality is often treated as a technical concern, but it is fundamentally a governance issue. Poor data quality violates the implicit contract between data producers and consumers. Claude Code can help enforce quality standards as part of governance:

```python
def generate_quality_rules(table_name: str, business_context: str,
                           schema: dict) -> list[dict]:
    """Generate data quality rules from business context."""
    client = anthropic.Anthropic()
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        system="""Generate data quality rules as JSON. For each rule:
        - rule_id, description, severity (critical/warning/info)
        - check_type (not_null, unique, accepted_values, range,
          custom_sql, referential_integrity, freshness)
        - implementation as SQL or dbt test YAML""",
        messages=[{"role": "user",
            "content": f"Table: {table_name}\n"
                       f"Business context: {business_context}\n"
                       f"Schema: {json.dumps(schema, indent=2)}"}],
    )
    return json.loads(message.content[0].text)
```

Listing 17.9: Quality rule generation from business requirements

17.7 Case Study: Governance Transformation at a Healthcare Company

A mid-size healthcare analytics company with 300 tables across three databases had no formal governance. After a close call with a HIPAA audit, leadership mandated a governance program. Using Claude Code, the team achieved the following in 12 weeks:

Weeks 1–2: Ran automated catalog generation across all three databases. Claude Code produced descriptions for all 300 tables and 4,200 columns. Manual review by domain experts refined 15% of descriptions.

Weeks 3–4: Deployed the hybrid PII scanner. Identified 47 columns containing PHI that were not previously classified, including three columns in the analytics schema that contained patient names in free-text "notes" fields.

Weeks 5–8: Implemented governance-as-code with 12 policies. Integrated into CI/CD. Blocked three PRs in the first week that would have exposed unmasked PHI to the analytics layer.

Weeks 9–12: Generated HIPAA compliance documentation: data flow diagrams, access control matrices, and encryption inventories. Passed the subsequent HIPAA audit with zero findings.

> **Warning**
>
> This case study illustrates what is achievable, but every organization's regulatory environment is different. Engage qualified legal and compliance professionals before implementing governance programs for regulated industries.

17.8 Exercises

1. **Catalog Generation**: Run the automated cataloging pipeline against a database you manage. Review the generated descriptions for accuracy. How many are correct without modification? What patterns does Claude Code get wrong?

2. **PII Scanner**: Run the hybrid PII detection pipeline against a staging database. Compare regex-only findings to the hybrid approach. How many additional PII columns does the semantic scan identify?

3. **Policy Engine**: Implement three governance policies relevant to your organization. Integrate them into your CI/CD pipeline. Track how many violations they catch in the first two weeks.

4. **DSAR Simulation**: Using a test database with synthetic data, generate DSAR queries for a fictional data subject. Verify that the queries find all relevant data across all tables.

5. **Governance Roadmap**: Draft a 12-week governance implementation plan for your organization using the case study as a template. Identify the top five policies to implement first and justify your prioritization.

Summary

In this chapter, we built a comprehensive data governance practice powered by Claude Code: automated data cataloging with gap analysis, lineage tracing and column-level lineage extraction, hybrid PII detection combining regex and semantic analysis, compliance automation for GDPR/CCPA/HIPAA including DSAR and erasure support, governance-as-code policy enforcement integrated into CI/CD, and data quality as a governance concern.

Note

Governance is a team sport. Claude Code accelerates the technical work—scanning, classifying, generating documentation—but organizational success depends on executive sponsorship, clear ownership, and a culture that values data as a strategic asset. The best governance program is one that teams actually follow, and automation makes compliance the path of least resistance.

18

Security and Access Control

Tip

Security is not a feature you bolt on at the end of a data pipeline—it is a foundational discipline woven into every layer. Claude Code can dramatically accelerate security tasks such as policy generation, vulnerability assessment, and audit log analysis, but ultimate responsibility for protecting sensitive data remains with your team.

Data engineers work directly with raw datasets, build pipelines that move sensitive information, and design storage layers. A single misconfigured pipeline can expose millions of records. A single over-permissioned role can give an intern access to production PII. This chapter provides production-ready patterns for RBAC/ABAC, encryption, data masking, row-level security, audit log anomaly detection, and secrets management—all augmented by Claude Code.

18.1 Data Security Fundamentals

Before diving into implementation, establish the mental models that guide all security decisions.

The ***CIA triad***—Confidentiality, Integrity, and Availability—is the bedrock of information security. Confidentiality ensures data is accessible only to authorized parties. Integrity ensures data has not been tampered with or corrupted. Availability ensures data is accessible when needed.

Defense in depth layers controls across multiple levels: network (firewalls, VPNs,

private endpoints), identity (SSO, MFA, service accounts), authorization (RBAC, ABAC, row-level security), data (encryption, masking, tokenization), and monitoring (audit logs, anomaly detection, alerting). A failure at one layer should not compromise the system because other layers provide backup protection.

The ***principle of least privilege*** grants only minimum necessary permissions for the minimum necessary duration. This applies to human users, service accounts, and automated pipelines. Every permission granted is an attack surface expanded.

Zero trust assumes no user, device, or network segment is inherently trustworthy. Every access request is authenticated and authorized, regardless of source. This model is particularly important for data platforms that span cloud environments and serve diverse consumer groups.

Warning

Over-permissioning is the most common security anti-pattern in data engineering. It is far easier to grant access than to revoke it after a breach. Start with no permissions and add them incrementally based on documented, approved requests. Audit permissions quarterly and revoke anything unused for 90 days.

The Data Engineer's Security Responsibilities

Data engineers have unique security responsibilities because they operate at the intersection of raw data and transformation logic:

- **Pipeline credentials**: Managing database passwords, API keys, and service account tokens used by automated pipelines.
- **Data movement**: Ensuring that sensitive data is encrypted in transit as it moves between systems.
- **Access control design**: Defining who can read, write, and modify tables, views, and schemas.
- **Data masking**: Ensuring that non-production environments (staging, development) do not contain unmasked PII.
- **Audit compliance**: Maintaining logs that demonstrate who accessed what data and when.

Note

Security as a Career Differentiator Data engineers who understand security deeply are disproportionately valuable. Most data engineering curricula underemphasize security, creating a skills gap. Investing in security knowledge pays dividends in career advancement and organizational impact.

18.2 RBAC and ABAC Design

Role-Based Access Control (RBAC) assigns permissions to roles rather than individual users. Users are granted roles, and roles are organized in a hierarchy where higher roles inherit permissions from lower ones.

```
-- Create the role hierarchy
CREATE ROLE IF NOT EXISTS data_reader;
CREATE ROLE IF NOT EXISTS data_analyst;
CREATE ROLE IF NOT EXISTS data_engineer;
CREATE ROLE IF NOT EXISTS data_admin;
CREATE ROLE IF NOT EXISTS security_admin;

-- Establish hierarchy: analyst inherits reader, etc.
GRANT ROLE data_reader TO ROLE data_analyst;
GRANT ROLE data_analyst TO ROLE data_engineer;
GRANT ROLE data_engineer TO ROLE data_admin;
GRANT ROLE security_admin TO ROLE SYSADMIN;

-- Reader: SELECT on production tables
GRANT USAGE ON DATABASE analytics_warehouse TO ROLE data_reader;
GRANT SELECT ON ALL TABLES IN DATABASE analytics_warehouse
    TO ROLE data_reader;
GRANT SELECT ON FUTURE TABLES IN DATABASE analytics_warehouse
    TO ROLE data_reader;

-- Engineer: full access to staging
GRANT ALL PRIVILEGES ON DATABASE staging TO ROLE data_engineer;

-- Security admin: manage grants
GRANT MANAGE GRANTS ON ACCOUNT TO ROLE security_admin;
```

Listing 18.1: RBAC role hierarchy in Snowflake

RBAC Design Principles

When designing an RBAC hierarchy, follow these principles:

1. **Separation of duties**: No single role should be able to both create data and approve its release. The role that writes to production should not be the role that manages access to production.

2. **Role granularity**: Create roles at the right level of granularity. Too few roles lead to over-permissioning; too many roles become unmanageable. A good heuristic is one role per logical persona (reader, analyst, engineer, admin, security).

3. **Environment isolation**: Separate roles for development, staging, and production. A developer should never accidentally modify production data.

4. **Future-proofing**: Use `FUTURE GRANTS` (Snowflake) or equivalent mechanisms to ensure new objects automatically inherit appropriate permissions.

Tip

Use Claude Code to generate RBAC hierarchies from natural language descriptions of your team structure. Provide the team roles, data assets, and security requirements, and Claude Code will generate the complete SQL. Then review carefully—Claude Code may not know your organization's specific compliance requirements.

Attribute-Based Access Control

ABAC extends RBAC by evaluating attributes of the user, resource, and environment at query time. This enables policies such as "EU users can only access data where `data_region = 'EU'`" or "contractors can only access data from the current quarter."

```
-- Row-level security policy: regional data access
CREATE OR REPLACE ROW ACCESS POLICY regional_access AS
    (region VARCHAR) RETURNS BOOLEAN ->
    CASE
        WHEN CURRENT_ROLE() IN ('DATA_ADMIN', 'SECURITY_ADMIN')
            THEN TRUE  -- Admins see all regions
        WHEN CURRENT_ROLE() = 'EU_ANALYST'
```

```
            THEN region IN ('EU', 'UK')
        WHEN CURRENT_ROLE() = 'US_ANALYST'
            THEN region IN ('US', 'CA')
        ELSE FALSE
    END;

-- Apply to table
ALTER TABLE analytics.customers
    ADD ROW ACCESS POLICY regional_access ON (country_region);
```

Listing 18.2: ABAC-style row-level security in Snowflake

Claude Code can generate ABAC policies from natural-language requirements, translating business rules like "regional managers see data for their region and all sub-regions" into SQL policies with appropriate role mappings.

18.3 Encryption Strategies

Encryption protects data at three levels, each addressing a different threat model.

Encryption at rest with customer-managed keys (CMKs) provides control beyond platform-managed encryption. If you use platform-managed keys, the cloud provider controls the keys—which means a compromised provider or a government subpoena could access your data. CMKs give you the ability to revoke access independently.

Encryption in transit enforces TLS 1.2+ for all data movement: between your application and the database, between pipeline components, and between your warehouse and BI tools. Ensure that your connection strings specify SSL/TLS and that certificate validation is enabled (not disabled for "convenience").

Column-level encryption adds protection for individual sensitive columns (SSNs, credit cards, health identifiers) beyond what at-rest encryption provides. Even if someone gains access to the table, encrypted columns remain unreadable without the decryption key.

```
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
import base64, os

class ColumnEncryptor:
    """Encrypt and decrypt individual column values.

```

```
    Uses PBKDF2 key derivation for security and Fernet
    (AES-128-CBC + HMAC-SHA256) for authenticated encryption.
    """

    def __init__(self, master_key: bytes, salt: bytes = None):
        self.salt = salt or os.urandom(16)
        kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32,
                         salt=self.salt, iterations=480_000)
        key = base64.urlsafe_b64encode(kdf.derive(master_key))
        self.fernet = Fernet(key)

    def encrypt_value(self, plaintext: str) -> str:
        return self.fernet.encrypt(
            plaintext.encode("utf-8")).decode("utf-8")

    def decrypt_value(self, ciphertext: str) -> str:
        return self.fernet.decrypt(
            ciphertext.encode("utf-8")).decode("utf-8")

    def encrypt_dataframe_column(self, df, column_name: str):
        """Encrypt a column in a pandas DataFrame in place."""
        df[column_name] = df[column_name].apply(
            lambda v: self.encrypt_value(str(v)) if v else v
        )
        return df
```

Listing 18.3: Column-level encryption utility

Warning

Column-level encryption makes columns unsearchable and unjoinable without decryption. Plan your query patterns before implementing it. If you need to join on an encrypted column, consider deterministic encryption (same plaintext always produces the same ciphertext) at the cost of reduced security.

18.4 Data Masking and Tokenization

Data masking replaces sensitive values with fictional but realistic substitutes. Unlike encryption, masking is typically irreversible (one-way) and preserves the format and statistical properties of the data.

```
-- Email masking: show full to admins, partial to analysts, none to
    others
CREATE OR REPLACE MASKING POLICY mask_email AS
    (val STRING) RETURNS STRING ->
    CASE
        WHEN CURRENT_ROLE() IN ('DATA_ADMIN', 'SECURITY_ADMIN')
            THEN val
        WHEN CURRENT_ROLE() IN ('DATA_ANALYST')
            THEN REGEXP_REPLACE(val, '^(.{2})(.*)(@.*)$', '\\1****\\3')
        ELSE '****@****.***'
    END;

-- SSN masking: full to security admin, last 4 to admin, none to others
CREATE OR REPLACE MASKING POLICY mask_ssn AS
    (val STRING) RETURNS STRING ->
    CASE
        WHEN CURRENT_ROLE() IN ('SECURITY_ADMIN') THEN val
        WHEN CURRENT_ROLE() IN ('DATA_ADMIN')
            THEN 'XXX-XX-' || RIGHT(val, 4)
        ELSE 'XXX-XX-XXXX'
    END;

-- Phone masking: preserve country code for analytics
CREATE OR REPLACE MASKING POLICY mask_phone AS
    (val STRING) RETURNS STRING ->
    CASE
        WHEN CURRENT_ROLE() IN ('DATA_ADMIN', 'SECURITY_ADMIN')
            THEN val
        ELSE LEFT(val, 3) || '-XXX-XXXX'
    END;

-- Apply policies to columns
ALTER TABLE hr.employees MODIFY COLUMN email
    SET MASKING POLICY mask_email;
ALTER TABLE hr.employees MODIFY COLUMN ssn
    SET MASKING POLICY mask_ssn;
ALTER TABLE hr.employees MODIFY COLUMN phone
    SET MASKING POLICY mask_phone;
```

Listing 18.4: Snowflake dynamic data masking policies

Tokenization

Tokenization replaces sensitive values with opaque tokens that can be reversed by an authorized vault. This differs from encryption in that the token bears no mathematical relationship to the original value—you cannot derive the original from the token without access to the vault's lookup table.

Use **deterministic tokenization** when you need to join on tokenized values across tables. The same input always produces the same token, enabling joins without detokenization. Use **random tokenization** for stronger security when joins are not needed.

Tip

For non-production environments, use masking rather than tokenization. Masking is simpler, does not require vault infrastructure, and is sufficient for development and testing. Reserve tokenization for production scenarios where reversibility is required (e.g., customer service lookups).

Masking for Non-Production Environments

A common security gap is using production data in development and staging environments. Claude Code can generate masking scripts that produce realistic but fictional data:

```
-- Generate a masked copy of the customers table for staging
CREATE TABLE staging.customers AS
SELECT
    customer_id,
    -- Preserve format but replace values
    MD5(first_name || 'salt') AS first_name,
    MD5(last_name || 'salt') AS last_name,
    CONCAT('user', customer_id, '@example.com') AS email,
    'XXX-XX-' || LPAD(MOD(ABS(HASH(ssn)), 10000)::VARCHAR, 4, '0')
        AS ssn,
    -- Preserve distribution but shift dates
    date_of_birth + INTERVAL '30 days' AS date_of_birth,
    -- Keep non-sensitive columns unchanged
    signup_date, customer_segment, region, status
FROM production.customers;
```

Listing 18.5: Production-to-staging masking script

18.5 Row-Level Security

Row-level security (RLS) restricts which rows a user can see based on their identity or role. This is essential for multi-tenant platforms, regional data isolation, and time-based access restrictions.

```
-- Analysts can only see data from the last 90 days
-- Engineers can see all data
CREATE OR REPLACE ROW ACCESS POLICY time_based_access AS
    (event_date DATE) RETURNS BOOLEAN ->
    CASE
        WHEN CURRENT_ROLE() IN ('DATA_ENGINEER', 'DATA_ADMIN')
            THEN TRUE
        WHEN CURRENT_ROLE() = 'DATA_ANALYST'
            THEN event_date >= CURRENT_DATE - 90
        ELSE FALSE
    END;
```

Listing 18.6: Time-based row-level security

Note

RLS Performance Row-level security policies are evaluated at query time and can impact performance. Test RLS policies with realistic data volumes. In Snowflake, RLS policies are pushed down to the micro-partition level, so performance impact is typically minimal. In other systems, the impact may be more significant.

18.6 Audit Log Anomaly Detection with Claude Code

Audit logs are your security camera system. They record who accessed what data, when, and how. But raw logs are only useful if someone—or something—is watching them.

```
CREATE MATERIALIZED VIEW security.user_baselines AS
WITH daily_metrics AS (
    SELECT user_id, event_timestamp::date AS activity_date,
           COUNT(*) AS daily_events,
           COUNT(DISTINCT resource_name) AS daily_resources,
           COUNT(*) FILTER (WHERE result = 'denied') AS daily_denials,
```

```
            COUNT(DISTINCT ip_address) AS daily_unique_ips
    FROM security.audit_log
    WHERE event_timestamp >= NOW() - INTERVAL '30 days'
    GROUP BY user_id, event_timestamp::date
)
SELECT user_id,
    AVG(daily_events) AS avg_daily_events,
    STDDEV(daily_events) AS stddev_daily_events,
    AVG(daily_denials) AS avg_daily_denials,
    AVG(daily_unique_ips) AS avg_daily_ips,
    PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY daily_events)
        AS p95_daily_events
FROM daily_metrics GROUP BY user_id
HAVING COUNT(DISTINCT activity_date) >= 5;

-- Detect anomalies: users exceeding baselines
CREATE VIEW security.anomalous_activity AS
SELECT al.user_id, al.event_timestamp::date AS activity_date,
    COUNT(*) AS today_events, ub.avg_daily_events,
    CASE WHEN ub.stddev_daily_events > 0 THEN
        (COUNT(*) - ub.avg_daily_events) / ub.stddev_daily_events
    ELSE 0 END AS z_score
FROM security.audit_log al
JOIN security.user_baselines ub ON ub.user_id = al.user_id
WHERE al.event_timestamp::date = CURRENT_DATE
GROUP BY al.user_id, al.event_timestamp::date,
         ub.avg_daily_events, ub.stddev_daily_events, ub.avg_daily_ips
HAVING CASE WHEN ub.stddev_daily_events > 0 THEN
    (COUNT(*) - ub.avg_daily_events) / ub.stddev_daily_events
    ELSE 0 END > 3
OR COUNT(*) FILTER (WHERE al.result = 'denied') > 10;
```

Listing 18.7: Computing user behavioral baselines

Claude Code can classify detected anomalies into categories—credential compromise, insider threat, privilege escalation, data exfiltration, brute force, policy violation, misconfiguration, or normal deviation—and generate incident reports for high-severity findings.

```
def classify_anomaly(anomaly_data: dict) -> dict:
    """Use Claude Code to classify a detected anomaly."""
    client = anthropic.Anthropic()
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=2048,
```

```
        system="""You are a security analyst. Classify this anomaly:
        Categories: credential_compromise, insider_threat,
        privilege_escalation, data_exfiltration, brute_force,
        policy_violation, misconfiguration, normal_deviation.
        Return JSON with: category, confidence, severity (critical/
        high/medium/low), recommended_actions, investigation_queries.""",

        messages=[{"role": "user",
            "content": json.dumps(anomaly_data, indent=2)}],
    )
    return json.loads(message.content[0].text)
```

Listing 18.8: Claude Code anomaly classification

Warning

Anomaly detection systems generate false positives. Tune your z-score threshold based on your environment. Start with a threshold of 3 (capturing events more than 3 standard deviations from the mean) and adjust based on alert volume. Too many false positives leads to alert fatigue and missed real incidents.

18.7 Compliance Frameworks

Understanding compliance frameworks helps you design security controls that satisfy multiple requirements simultaneously.

SOC 2 evaluates controls across five Trust Service Criteria: Security, Availability, Processing Integrity, Confidentiality, and Privacy. Most data platforms need SOC 2 Type II, which evaluates controls over a period (typically 6–12 months).

GDPR imposes data minimization, right to erasure, data protection by design and by default, and records of processing. The key principle is "privacy by design"—security controls should be built into the architecture, not added after the fact.

HIPAA requires encryption of Protected Health Information (PHI), access controls, six-year audit trails, and Business Associate Agreements (BAAs) with all vendors who process PHI. This includes your cloud provider, your warehouse vendor, and any AI services used for data processing.

Claude Code can perform gap analysis between your current controls and framework requirements, generating a prioritized remediation plan.

Note

Always review CLAUDE CODE-generated security configurations before applying them to production. Treat the output as a high-quality first draft that requires human validation. Security misconfigurations can be catastrophic, and AI-generated policies may miss organization-specific requirements.

18.8 Secrets Management

Hardcoded credentials are one of the most common vulnerabilities in data engineering. A database password in a Git repository, an API key in an environment variable, or a service account key file on a shared server—each is a breach waiting to happen.

Use a dedicated secrets manager (AWS Secrets Manager, HashiCorp Vault, GCP Secret Manager) with:

- **Encryption**: Secrets encrypted at rest with customer-managed keys.
- **Access logging**: Every secret access logged with timestamp, identity, and source IP.
- **Automatic rotation**: Credentials rotated on a schedule (e.g., every 90 days) without pipeline downtime.
- **Least privilege**: Each pipeline has its own service account with access only to the secrets it needs.
- **Short-lived credentials**: Prefer temporary credentials (AWS STS, GCP workload identity federation) over long-lived keys.

```
import boto3, json
from functools import lru_cache

@lru_cache(maxsize=1)
def get_warehouse_credentials() -> dict:
    """Fetch credentials from AWS Secrets Manager with caching."""
    client = boto3.client("secretsmanager", region_name="us-east-1")
    response = client.get_secret_value(
        SecretId="prod/warehouse/credentials"
    )
    return json.loads(response["SecretString"])

```

```python
# Usage in pipeline code
creds = get_warehouse_credentials()
conn = snowflake.connector.connect(
    account=creds["account"],
    user=creds["user"],
    password=creds["password"],
    warehouse=creds["warehouse"],
    database=creds["database"],
)
```

Listing 18.9: Secrets management best practices

Warning

Never store secrets in environment variables in production. Environment variables are visible to any process in the same context and often appear in crash dumps, log files, and process listings. Use a secrets manager and inject credentials at runtime through the manager's SDK.

Tip

Use Claude Code to scan your codebase for hardcoded credentials. Feed your repository's file list and sample code to Claude Code and ask it to identify potential secrets: API keys, passwords, connection strings, and private keys. Many teams are surprised by what they find.

18.9 Case Study: Securing a Multi-Tenant Data Platform

A B2B SaaS company serves 200 customers from a shared Snowflake warehouse. Each customer's data must be isolated—no customer should ever see another customer's data. The security requirements:

1. **Row-level isolation**: Every query must be filtered by `tenant_id`.
2. **Column-level masking**: PII columns show different levels of detail based on role.
3. **Audit trail**: Every data access logged with tenant context.
4. **SOC 2 compliance**: Full audit trail for 12 months, encryption at rest and in transit.

Using Claude Code, the team generated RLS policies for all 45 tenant-aware tables, masking policies for 12 PII columns across 8 tables, audit log baselines and anomaly detection queries, and SOC 2 control documentation mapping each requirement to an implemented control. The implementation took three weeks instead of the estimated eight, with Claude Code generating approximately 70% of the SQL and documentation, reviewed and refined by the security team.

Exercises

1. **RBAC Design**: Design a complete RBAC hierarchy for a healthcare analytics platform that must comply with HIPAA. Include at least six roles, the SQL to implement it, and a justification for each permission grant.

2. **Masking Policy**: Write Snowflake masking policies for a `customers` table containing: `full_name`, `email`, `ssn`, `date_of_birth`, and `credit_card_number`. Define three roles with different masking levels and test each role's view of the data.

3. **Compliance Gap Analysis**: Use Claude Code to perform a SOC 2 gap analysis for a startup with SSO but no MFA, credentials in environment variables, no formal change management, and basic logging but no anomaly detection. Generate a prioritized remediation plan.

4. **RLS System**: Design a row-level security system that supports time-based policies (e.g., "analysts can only see data from the last 90 days") and hierarchical policies (e.g., "regional managers see all countries in their region"). Implement and test in your warehouse.

5. **Secrets Audit**: Audit your team's current secrets management practices. Identify all hardcoded credentials, environment variable secrets, and unrotated keys. Create a migration plan to move all secrets to a dedicated secrets manager.

Summary

We built a comprehensive security framework: RBAC/ABAC hierarchies with least-privilege design, encryption at rest/in transit/column-level, dynamic data masking and tokenization for production and non-production environments, row-level security for multi-tenant isolation, audit log anomaly detection with Claude Code classifi-

cation, compliance mapping across SOC 2/GDPR/HIPAA, and secrets management best practices.

Note

Security is a journey, not a destination. Threats evolve, regulations change, and your data platform grows. Make security a first-class concern in every sprint, not an afterthought during audit season. The best time to fix a security gap is before it becomes a breach.

19

Ethics of AI in Data Engineering

Every time a data engineer uses Claude Code to generate a query, build a pipeline, or make a design choice, they delegate a portion of their professional judgment to a system that lacks moral agency and accountability for outcomes. This chapter confronts the ethical dimensions head-on: bias in AI-generated code, hallucination risks, privacy implications, accountability frameworks, and governance structures for responsible AI use.

19.1 The Responsibility of AI-Augmented Data Engineering

When an engineer asks Claude Code to generate a query, responsibility does not transfer to the AI. The engineer remains fully accountable for every line of code that enters production. This principle—that ***human accountability persists through AI assistance***—is the foundation of ethical AI use.

This may seem obvious in the abstract, but it becomes complex in practice. Consider these scenarios:

- A Claude Code-generated query silently filters out records for a minority demographic due to a hardcoded assumption in the WHERE clause. Who is responsible when the downstream dashboard underreports that population?
- A pipeline generated by Claude Code handles currency conversion correctly for USD, EUR, and GBP but uses an outdated exchange rate for currencies

from developing nations. The resulting financial reports disadvantage certain regional teams.

- An AI-generated data quality check passes because the test data was representative of US customers but not of the global customer base. PII from non-US customers leaks through a masking gap.

In each case, the answer is the same: the engineer who approved and deployed the code bears responsibility. AI does not change the accountability model—it changes the *workflow* through which accountable decisions are made.

Warning

The most dangerous moment in AI-assisted development is when everything appears to work. Correct-looking output can mask subtle logical errors that only manifest with production data volumes, edge-case inputs, or demographic groups underrepresented in testing.

The Automation Complacency Problem

As AI-generated code becomes more reliable, engineers face ***automation complacency***: the tendency to accept outputs without sufficient scrutiny. This is well-documented in aviation (where autopilot has degraded manual flying skills) and medicine (where AI diagnostic tools can reduce clinical reasoning).

In data engineering, automation complacency manifests as:

- Accepting SQL without reviewing join logic or null handling.
- Deploying migrations without testing rollback procedures.
- Trusting AI-generated quality checks without verifying edge cases.
- Copy-pasting AI-generated pipeline code without understanding error handling paths.
- Skipping code review because "Claude Code wrote it, so it must be good."

The antidote is not to avoid AI, but to build disciplined review processes that match the risk level of each task.

Risk-Proportionate Oversight

AI involvement exists on a risk spectrum, and human oversight should scale accordingly:

Low risk: Boilerplate generation, syntax help, documentation, code formatting. Minimal review needed beyond basic correctness.

Medium risk: Business logic, test generation, query optimization. Standard code review with attention to edge cases.

High risk: Compliance queries, PII handling, access control policies. Detailed review by subject matter expert plus security review.

Critical risk: Healthcare data transformations, financial regulatory reporting, algorithmic decision-making affecting individuals. Formal review process with sign-off, testing against production-representative data, and documentation of AI involvement.

Tip

Create a simple decision matrix for your team: for each category of work (pipeline code, SQL queries, infrastructure config, access policies), define the minimum review requirements when AI is involved. Post it in your team's documentation and reference it in code review checklists.

19.2 Bias in AI-Generated Data Pipelines

Bias in AI-generated code is not always obvious. It does not look like explicit discrimination—it looks like reasonable defaults that happen to disadvantage certain groups.

Sources of Bias

Bias arises from multiple sources:

- **Training data patterns**: Claude Code has seen more US-centric code than code from other regions. It may default to US date formats (`MM/DD/YYYY`), US-centric timezone handling, English-language text processing assumptions, and dollar-denominated financial calculations.

- **Prompt framing**: The way you describe requirements influences the output. "Segment customers into high-value and low-value" may produce a binary classification that disadvantages customers who are valuable in ways not captured by the specified metric.
- **Selection bias in examples**: If your few-shot examples only cover common cases, the generated code may not handle edge cases that disproportionately affect minority populations.
- **Survivorship bias**: Claude Code's training data overrepresents successful, well-documented projects and underrepresents the silent failures and edge cases that are critical to equitable data systems.
- **Proxy discrimination**: A query that filters by zip code may effectively filter by race. A model that uses "years of experience" as a feature may discriminate by age.

A Practical Bias Detection Framework

Code Example

```
import anthropic, json

def audit_query_for_bias(
    query: str, context: str, protected_attributes: list[str]
) -> dict:
    """Audit a SQL query for potential bias against protected groups.
    """
    client = anthropic.Anthropic()
    prompt = f"""Analyze this SQL query for potential bias.

Query: ```sql\n{query}\n```
Business Context: {context}
Protected Attributes: {', '.join(protected_attributes)}

Check for:
1) Direct use of protected attributes in filtering or scoring
2) Proxy variables that correlate with protected attributes
   (e.g., zip code as proxy for race, name patterns for ethnicity)
3) Hardcoded thresholds that disadvantage specific populations
4) Missing handling of international formats, languages, or customs
5) Assumptions about data distributions that may not hold for
```

```
    all demographic groups
6) Aggregation methods that mask disparities (e.g., averages
    that hide bimodal distributions)

Return JSON with "issues" array (each with description, severity,
affected_groups, mitigation) and "overall_risk" field."""

    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        messages=[{"role": "user", "content": prompt}])
    text = response.content[0].text
    return json.loads(text[text.find("{"):text.rfind("}")+1])
```

Bias Mitigation Strategies

Mitigation requires a multi-layered approach:

1. **Diverse review**: Ensure code reviewers include people from different backgrounds who may spot assumptions invisible to the original author.
2. **Bias-specific test cases**: Write tests that explicitly check for differential outcomes across demographic groups. If your pipeline scores customers, test that score distributions are fair across protected attributes.
3. **Explicit fairness requirements in prompts**: When asking Claude Code to generate queries involving people, include fairness constraints in the prompt.
4. **Regular audits**: Schedule quarterly bias audits of production pipelines, especially those that influence decisions about people (credit scoring, hiring, content recommendations).
5. **Documentation**: Record AI involvement in code that affects people, including the prompt used, the model version, and any bias review performed.

Tip

When asking Claude Code to generate queries involving people, always include this in your prompt: "Ensure this query does not directly or indirectly discriminate based on race, gender, age, disability, or other protected characteristics. Flag any assumptions about demographics or geography."

Note

Bias Is Not Just a Technical Problem Technical bias detection catches the symptoms, but the root causes are often in business requirements. "Identify our best customers" may encode biases about who counts as "best." Data engineers should push back on requirements that embed discriminatory assumptions, even when the SQL itself is technically correct.

19.3 Hallucination: Causes, Detection, and Mitigation

Hallucination in the context of data engineering manifests in several dangerous forms:

- **Invented column names**: Claude Code references a column that does not exist in your schema, often choosing a plausible name based on the table context.
- **Incorrect function signatures**: Using a function with the wrong number of arguments or wrong argument types, especially for dialect-specific functions.
- **Fabricated best practices**: Stating "it is best practice to..." followed by advice that sounds authoritative but is not supported by documentation or expert consensus.
- **Plausible but wrong business logic**: Generating a query that looks correct but implements the wrong business rule—for example, calculating revenue as `quantity * price` when the actual formula requires applying discounts and taxes.
- **Non-existent configuration options**: Referencing Airflow operator parameters, dbt config keys, or cloud service options that do not exist.

In data engineering, hallucinated SQL is particularly dangerous because it is executed by machines that cannot distinguish correct from incorrect logic. A hallucinated column name causes an immediate error (detectable); a hallucinated join condition produces wrong results silently (catastrophic).

Detection Strategies

1. **Schema validation**: Always provide actual schema definitions rather than relying on the model to guess. Compare generated column references against the actual schema programmatically.

2. **Dry-run execution**: Execute generated SQL with `EXPLAIN` or `LIMIT 0` before running against real data. This catches non-existent columns and type mismatches.

3. **Dual-prompt verification**: Generate the same query twice with different prompts and compare the results. Disagreements indicate areas requiring human judgment.

4. **Version specification**: Explicitly specify tool and library versions in your prompts. "Using dbt 1.7 with Snowflake adapter 1.7.2" prevents the model from referencing features from other versions.

5. **Runtime assertions**: Embed assertions in generated code that verify assumptions at execution time—row count checks, null count checks, value range checks.

Warning

Never trust Claude Code when it cites specific documentation URLs, version numbers, release dates, or benchmark results. Always verify against official sources. The model generates plausible-sounding citations that may not correspond to real documents.

A Hallucination Severity Matrix

Not all hallucinations are equally dangerous. Classify them by impact:

Compile-time detectable: Non-existent column names, syntax errors, wrong function signatures. These fail immediately and are low risk.

Test-detectable: Wrong join types, incorrect aggregation logic, missing filters. These produce wrong results but are caught by data quality tests.

Silently wrong: Correct syntax but wrong business logic, subtle filter errors, incorrect timezone handling. These are the most dangerous because they produce plausible but incorrect results that may not be caught for days or weeks.

Tip

Invest your review time proportionally to the severity category. Spend less time checking syntax (which your compiler/executor catches) and more time verifying business logic (which only humans can validate).

19.4 Data Privacy and AI

When you interact with Claude Code through the API, your full prompt is transmitted to Anthropic's servers. Under standard API terms, Anthropic does not use API inputs to train models, but engineers should always verify current terms and ensure their organization has appropriate data processing agreements in place.

The privacy implications extend beyond the obvious:

- **Schema metadata**: Even without sending actual data, sending your schema reveals your data model, business entities, and potentially sensitive column names.
- **Query patterns**: The queries you ask Claude Code to generate reveal your business logic, analytical priorities, and potentially sensitive business strategies.
- **Error messages**: Pasting tracebacks may inadvertently include connection strings, file paths, or data samples from error output.
- **Prompt accumulation**: Over time, the sum of all prompts from your organization creates a comprehensive picture of your data infrastructure, even if no single prompt is sensitive.

Warning

Never send production data containing PII, PHI, financial account numbers, or other sensitive information to any AI service without explicit authorization from your security and legal teams. This includes data samples in prompts, error messages containing data, and schema comments that reference real individuals.

Data Classification for AI Interactions

Organizations should establish a data classification scheme that governs what can be shared with AI services:

Public: Non-sensitive data, freely shareable. Schema structures without sensitive column names, generic code patterns, open-source project references.

Internal: Not sensitive but not public. Business logic descriptions, internal tool configurations, non-sensitive schema designs.

Confidential: Requires protection. Production schema metadata, business-specific query patterns, performance data, cost information.

Restricted: Never share with external services. PII, PHI, credentials, encryption keys, financial account numbers, production data samples.

Note

Some organizations deploy self-hosted LLM solutions for restricted data. While this eliminates the external data transmission concern, it introduces new challenges: model quality, infrastructure cost, and update cadence. Evaluate the trade-offs carefully before building in-house.

19.5 Human-in-the-Loop Patterns

For critical data systems, blind automation is unacceptable. ***Human-in-the-loop*** patterns ensure that AI assistance amplifies human judgment rather than replacing it.

Approval Gates

Route AI-generated changes to human reviewers based on risk level. Low-risk changes (documentation updates, formatting) can be auto-approved. Medium-risk changes (new columns, query modifications) require one reviewer. High-risk changes (access control, compliance logic, schema migrations) require two reviewers plus a domain expert.

Confidence-Based Routing

When Claude Code generates output, assess confidence based on factors like prompt specificity, schema coverage, and task complexity. Low-confidence outputs are routed to humans for review; high-confidence outputs go through automated validation (tests, dry runs, schema checks) before deployment.

Shadow Mode

Shadow mode is the gold standard for introducing AI into critical systems. The progression:

Phase 1 (Weeks 1–4): AI outputs are generated but not used. They are compared offline to human-produced results. This calibrates trust and identifies systematic errors.

Phase 2 (Weeks 5–8): AI outputs are shown as suggestions in the IDE or code review tool. Engineers choose to accept, modify, or reject each suggestion. Track acceptance rates by task type.

Phase 3 (Weeks 9–16): AI outputs are used by default for accepted task types. Humans sample a percentage of outputs for review. Reduce sampling rate as confidence grows.

Phase 4 (Month 4+): AI operates autonomously for proven task types with automated validation. Humans are alerted only on anomalies, low-confidence outputs, or failed validations.

Tip

Keep detailed records of each phase transition: what task types were promoted, what evidence supported the promotion, and what fallback procedures are in place. This documentation is invaluable for compliance audits and for replicating the process with new AI capabilities.

19.6 Organizational AI Governance

An effective AI governance framework includes these components:

1. **AI usage policy**: What AI tools are approved, what data can be shared with them, and who has authority to approve new use cases.

2. **Risk assessment process**: How to evaluate the risk of a new AI-assisted workflow before deploying it.

3. **Review and approval workflows**: Who reviews AI-generated code, at what granularity, and with what authority to approve or reject.

4. **Audit trail requirements**: What must be logged (prompt, response, model version, reviewer, outcome) and for how long.

5. **Incident response procedures**: What happens when AI-generated code causes an incident. How is root cause determined? How are similar incidents prevented?

6. **Training and certification**: Minimum training requirements for engineers using AI tools, including bias awareness, hallucination detection, and privacy protocols.

7. **Vendor assessment**: Evaluation criteria for AI service providers, including data handling practices, security certifications, and contractual protections.

8. **Continuous improvement**: Regular review of the governance framework itself, incorporating lessons from incidents, audits, and evolving best practices.

The Regulatory Landscape

The regulatory landscape for AI in data engineering is evolving rapidly and varies by jurisdiction:

EU AI Act: Establishes risk-based regulation with transparency obligations, high-risk classification criteria, data governance requirements, and human oversight mandates. Data pipelines that process personal data or influence automated decisions may fall under high-risk categories.

GDPR: Imposes constraints on automated decision-making (Article 22), requiring meaningful information about the logic involved and the right to human review.

CCPA/CPRA: Grants consumers the right to opt out of automated decision-making technologies.

HIPAA: Imposes strict controls on how AI tools interact with Protected Health Information.

SOX: Requires auditability of financial data processing, which extends to AI-generated transformations of financial data.

Warning

Regulatory compliance is not optional. If your data pipelines process regulated data, consult with legal and compliance teams before integrating AI tools. The regulatory landscape is changing faster than most engineering teams realize.

Note

Keep a regulatory watch list: subscribe to updates from relevant regulatory bodies, review AI-related legislation quarterly, and maintain relationships with your legal team. A proactive approach to compliance is far less costly than reactive remediation.

19.7 Building an Ethical AI Culture

Ethics cannot be enforced through policies alone—it requires a culture where engineers feel empowered to raise concerns, question AI outputs, and prioritize correctness over speed.

Key cultural practices:

- **Blameless postmortems**: When AI-generated code causes an issue, focus on process improvements rather than individual blame. Was the review process adequate? Were the right tests in place?
- **Ethics office hours**: Regular sessions where engineers can discuss ethical concerns about AI use cases, proposed pipelines, or data handling practices.
- **Red team exercises**: Periodically have team members attempt to generate biased, incorrect, or privacy-violating code through AI tools. Use the results to improve prompts, review checklists, and training materials.
- **Transparency by default**: Document AI involvement in all code that enters production. Future maintainers should know which code was AI-generated and what review it received.

Tip

Add an "AI Involvement" section to your code review template. For each PR, the author indicates which parts were AI-generated, what prompts were used, and what validation was performed. This normalizes AI use while maintaining accountability.

19.8 Exercises

Exercises

1. **Bias Audit**: Select a pipeline in your organization that makes decisions about people (customer segmentation, lead scoring, content recommendations). Using the bias detection framework from this chapter, audit it for fairness issues across at least three protected attributes. Document findings and propose specific mitigations.

2. **Hallucination Detection**: Generate five SQL queries using Claude Code for a schema you know well. For each query, apply at least three hallucination detection strategies from this chapter. Document every discrepancy found, classify it by severity, and estimate how long the issue would have persisted if deployed to production without review.

3. **AI Usage Policy Draft**: Using the governance framework from this chapter as a starting point, draft an AI usage policy customized for your organization. Include data classification rules, review requirements by risk level, training obligations, audit trail specifications, and incident response procedures. Present it to your team for feedback.

4. **Shadow Mode Plan**: Design a shadow mode rollout plan for introducing AI-assisted code generation into one of your team's workflows. Define the four phases, the metrics for phase transitions, the fallback procedures, and the documentation requirements. Estimate the timeline and resources needed.

5. **Ethical Dilemma Discussion**: Your team has built an AI-assisted pipeline that is 3x faster to develop but has a 2% higher error rate compared to manually written code. Management wants to adopt it for all pipelines, including financial reporting subject to SOX compliance. Write a one-page memo arguing for or against this decision, addressing risk, compliance, efficiency, and the ethical obligations of data engineers.

Part VII

Advanced Patterns

20

Claude Code as a Data Engineering Agent

"Any sufficiently advanced agent is indistinguishable from a junior engineer who never sleeps, never complains, and never eats the last donut."
—Claude Code, probably

Tip

This chapter explores autonomous agents that can plan, execute, and recover from complex multi-step workflows. That's right—Claude Code building things that use Claude Code. It's Claude Code all the way down. By the end, you will be able to design, build, and deploy Claude Code-powered agents that automate substantial portions of your data engineering operations, so you can finally get some sleep. Or at least pretend to.

AI agents represent the next leap in abstraction—systems that autonomously plan and execute sequences of actions, using tools, maintaining state, and recovering from errors. For data engineers, agents are compelling because pipelines are inherently multi-step, stateful, failure-prone, and span heterogeneous systems. Where a single prompt-response interaction can generate a SQL query or review a config file, an agent can diagnose a pipeline failure, trace the root cause across three services, apply a fix, validate the result, and notify the on-call engineer—all without human intervention.

20.1 What Are AI Agents

An ***AI agent*** uses a large language model as its reasoning core, augmented with the ability to take actions through ***tool use***. It operates in a loop:

1. **Observe**: Gather information from the environment—logs, metrics, query results, API responses.
2. **Reason**: Analyze the observations, form hypotheses, and plan the next action.
3. **Act**: Execute an action using a tool—run a query, call an API, modify a configuration.
4. **Evaluate**: Assess the result of the action and decide whether to continue, try a different approach, or terminate.

This loop continues until the agent achieves its goal, reaches a resource limit, or determines that human intervention is required. So, basically the same loop a human engineer follows, minus the existential dread and coffee breaks.

Agent capabilities range across a maturity spectrum:

Level 0 — Single Turn No tools. The model answers questions from its training data alone. Useful for knowledge queries but cannot take action.

Level 1 — Tool-Augmented Single round of tool use. The model calls one or more tools, processes results, and responds. Covers simple lookup and validation tasks.

Level 2 — Multi-Step Planning Iterative tool use with planning. The agent breaks a complex task into steps, executes them sequentially, and adapts the plan based on intermediate results.

Level 3 — Autonomous with Recovery Handles errors, retries transient failures, tries alternative approaches, and escalates when stuck. This is the sweet spot for most data engineering automation.

Level 4 — Collaborative Multi-Agent Multiple specialized agents coordinate to solve problems that span domains—for example, a pipeline agent handing off to a data quality agent, which escalates to a schema migration agent.

This chapter focuses on building Level 2 and Level 3 agents, which provide the best balance of capability and controllability for data engineering workflows.

Note

Data engineering is uniquely well-suited for agentic automation because of its repetitive operational patterns, rich tool ecosystems (APIs, CLIs, SQL interfaces), clear success and failure criteria, high cost of manual intervention during off-hours, and well-documented procedures in runbooks and playbooks. In other words, data engineering is the perfect job for a tireless robot. Don't take it personally.

20.2 Tool Use and Function Calling

Tools are the bridge between the agent's reasoning and the real world. Each tool is defined by a name, a description, and a JSON schema for its inputs. The quality of tool definitions directly determines agent performance—a well-described tool enables the model to use it correctly, while a vague description leads to hallucinated parameters and misuse.

Code Example

Defining tools for a data engineering agent

```
import anthropic

tools = [
    {
        "name": "run_sql_query",
        "description": (
            "Execute a read-only SQL query against the data warehouse. "
            "Returns results as a list of dictionaries. Max 1000 rows. "
            "Queries are executed with a 30-second timeout. "
            "Only SELECT statements are allowed."),
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "The SQL query to execute. Must be a "
                                   "SELECT statement. Use LIMIT to "
```

```
                                    "constrain result size."},
                "warehouse": {
                    "type": "string",
                    "enum": ["snowflake", "bigquery", "redshift"],
                    "description": "Target warehouse to run the query
 "
                                    "against."},
            },
            "required": ["query", "warehouse"]
        }
    },
    {
        "name": "check_pipeline_status",
        "description": (
            "Check the current status of an Airflow DAG run. "
            "Returns DAG run state, individual task states, "
            "error messages, and duration. If no execution_date "
            "is provided, returns the most recent run."),
        "input_schema": {
            "type": "object",
            "properties": {
                "dag_id": {
                    "type": "string",
                    "description": "The Airflow DAG identifier"},
                "execution_date": {
                    "type": "string",
                    "description": "ISO 8601 execution date. "
                                   "Optional; defaults to latest run.
"},
            },
            "required": ["dag_id"]
        }
    },
    {
        "name": "get_table_schema",
        "description": (
            "Retrieve the schema of a table including column names, "
            "types, nullability, and any comments or descriptions. "
            "Returns metadata as a dictionary."),
        "input_schema": {
            "type": "object",
            "properties": {
                "database": {"type": "string"},
                "schema": {"type": "string"},
```

```
                "table": {"type": "string"},
            },
            "required": ["database", "schema", "table"]
        }
    },
    {
        "name": "restart_failed_task",
        "description": (
            "Clear and restart a failed Airflow task instance. "
            "This is a WRITE operation that modifies pipeline state.
    "
            "Only clears the specific task, not downstream tasks. "
            "Returns the new task state after clearing."),
        "input_schema": {
            "type": "object",
            "properties": {
                "dag_id": {"type": "string"},
                "task_id": {"type": "string"},
                "execution_date": {"type": "string"},
            },
            "required": ["dag_id", "task_id", "execution_date"]
        }
    },
]
```

Warning

Vague or incomplete tool descriptions are the single most common cause of agent failures. Telling Claude Code to use a tool named `do_stuff` with no description is like handing a new hire a keyboard and saying "fix production." For every tool, state clearly: what the tool does, the format of its return value, any side effects it produces, constraints and limitations, and whether it is a read or write operation. Spending an extra five minutes on tool descriptions saves hours of debugging agent misbehavior.

Tool Design Principles

Follow these principles when designing tools for data engineering agents:

1. **Atomic operations**: Each tool should do one thing well. Prefer `get_table_schema`

and `run_sql_query` as separate tools over a single `do_database_stuff` tool.

2. **Clear read/write semantics**: Label tools as read-only or write, and describe side effects explicitly. This enables safety policies to distinguish observation from action.

3. **Bounded output**: Cap result sizes (e.g., max 1000 rows) and return structured data. An agent that receives a 50MB query result will exhaust its context window.

4. **Informative errors**: Return structured error messages that help the agent reason about what went wrong—"Column `user_id` not found in table `events`" is far more useful than "Query failed."

5. **Idempotent where possible**: Tools that can be safely retried simplify error recovery logic.

Always validate tool inputs before execution. Never blindly trust the arguments an LLM provides, especially for tools with side effects. Apply input sanitization, type checking, and authorization checks inside the tool executor, not in the agent loop. Trust, but verify. Actually, just verify.

20.3 The Model Context Protocol (MCP)

The ***Model Context Protocol (MCP)*** is an open standard that defines how AI models connect to external data sources and tools—a universal adapter layer that eliminates the need to build custom integrations for every service. MCP provides three primitives:

Tools Actions the model can take (e.g., run a query, restart a task, create a table).

Resources Data the model can read (e.g., schema definitions, configuration files, documentation).

Prompts Reusable prompt templates for common workflows (e.g., incident investigation, schema review).

The key advantage of MCP is composability: you can connect multiple MCP servers to a single agent, giving it access to your entire operational toolchain through a standardized protocol.

Code Example

Connecting MCP servers to a Claude Code agent

```
from claude_agent_sdk import Agent, MCPServerConfig

agent = Agent(
    model="claude-sonnet-4-20250514",
    mcp_servers=[
        MCPServerConfig(
            name="airflow",
            command="python",
            args=["mcp_servers/airflow_mcp.py"],
        ),
        MCPServerConfig(
            name="snowflake",
            command="python",
            args=["mcp_servers/snowflake_mcp.py"],
        ),
        MCPServerConfig(
            name="dbt",
            command="python",
            args=["mcp_servers/dbt_mcp.py"],
        ),
        MCPServerConfig(
            name="pagerduty",
            command="python",
            args=["mcp_servers/pagerduty_mcp.py"],
        ),
    ],
    system_prompt=(
        "You are a data engineering operations agent. "
        "Diagnose and resolve pipeline issues using the "
        "available tools. Always check pipeline status "
        "before attempting fixes. Log all actions taken."
    ),
)

response = agent.query(
    "The daily_customer_etl DAG failed at 3 AM. "
    "Investigate the root cause and fix it if possible."
)
```

Note

MCP servers can be implemented in any language. The protocol uses JSON-RPC over stdin/stdout, making it straightforward to wrap existing CLI tools or REST APIs. Community-maintained MCP servers exist for most popular data tools including Snowflake, BigQuery, dbt, Airflow, and Kafka.

Building a Custom MCP Server

When a community MCP server does not exist for your tool, building one is straightforward. The following example wraps a simplified Airflow REST API:

Code Example

Minimal MCP server for Airflow

```
from mcp.server import Server
from mcp.types import Tool, TextContent
import httpx, json

app = Server("airflow-mcp")
AIRFLOW_URL = "http://airflow.internal:8080/api/v1"
HEADERS = {"Authorization": "Bearer <token>"}

@app.list_tools()
async def list_tools():
    return [
        Tool(
            name="get_dag_runs",
            description="List recent DAG runs with status",
            inputSchema={
                "type": "object",
                "properties": {
                    "dag_id": {"type": "string"},
                    "limit": {"type": "integer", "default": 5},
                },
                "required": ["dag_id"],
            },
        ),
        Tool(
            name="get_task_logs",
            description="Retrieve logs for a specific task "
                        "instance. Returns the last 200 lines.",
```

```
            inputSchema={
                "type": "object",
                "properties": {
                    "dag_id": {"type": "string"},
                    "task_id": {"type": "string"},
                    "dag_run_id": {"type": "string"},
                },
                "required": ["dag_id", "task_id", "dag_run_id"],
            },
        ),
    ]

@app.call_tool()
async def call_tool(name: str, arguments: dict):
    async with httpx.AsyncClient() as client:
        if name == "get_dag_runs":
            resp = await client.get(
                f"{AIRFLOW_URL}/dags/{arguments['dag_id']}"
                f"/dagRuns?limit={arguments.get('limit', 5)}",
                headers=HEADERS,
            )
            return [TextContent(
                type="text", text=json.dumps(resp.json())
            )]
        elif name == "get_task_logs":
            resp = await client.get(
                f"{AIRFLOW_URL}/dags/{arguments['dag_id']}"
                f"/dagRuns/{arguments['dag_run_id']}"
                f"/taskInstances/{arguments['task_id']}/logs/1",
                headers=HEADERS,
            )
            lines = resp.text.strip().split("\n")
            return [TextContent(
                type="text", text="\n".join(lines[-200:])
            )]

if __name__ == "__main__":
    import asyncio
    from mcp.server.stdio import stdio_server
    asyncio.run(stdio_server(app))
```

20.4 Building a Data Engineering Agent

With tools defined and MCP servers connected, the next step is building the agent loop itself. The core pattern is deceptively simple: send a message with available tools, check if the response includes tool calls, execute the tools, feed results back, and repeat until the model produces a final text response.

Code Example

Core agent loop implementation

```python
import anthropic, json, logging
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Any, Callable

logger = logging.getLogger(__name__)

@dataclass
class AgentMemory:
    """Tracks conversation state and operational limits."""
    conversation_history: list[dict] = field(default_factory=list)
    actions_taken: list[dict] = field(default_factory=list)
    error_count: int = 0
    max_errors: int = 5
    iteration_count: int = 0
    max_iterations: int = 20

@dataclass
class SafetyPolicy:
    """Defines what the agent is and is not allowed to do."""
    allowed_tools: set[str] = field(default_factory=set)
    max_write_actions_per_session: int = 5
    require_confirmation_for: set[str] = field(default_factory=set)
    write_action_count: int = 0
    write_tools: set[str] = field(default_factory=lambda: {
        "restart_failed_task", "run_dbt_command",
        "execute_migration", "update_config",
    })

    def can_execute(self, tool_name: str) -> tuple[bool, str]:
        if self.allowed_tools and tool_name not in self.allowed_tools
    :
```

```
            return False, f"Tool {tool_name} not in allowed set."
        if tool_name in self.require_confirmation_for:
            return False, (
                f"Tool {tool_name} requires human confirmation.")
        if tool_name in self.write_tools:
            if self.write_action_count >= self.
    max_write_actions_per_session:
                return False, "Write action budget exhausted."
            self.write_action_count += 1
        return True, "Allowed."

class DataEngineeringAgent:
    """An autonomous agent for data engineering operations."""

    def __init__(self, tools, tool_executors, system_prompt,
                 safety_policy=None,
                 model="claude-sonnet-4-20250514"):
        self.client = anthropic.Anthropic()
        self.model = model
        self.tools = tools
        self.tool_executors = tool_executors
        self.system_prompt = system_prompt
        self.safety_policy = safety_policy or SafetyPolicy()
        self.memory = AgentMemory()

    def run(self, user_message: str) -> str:
        """Execute the agent loop until completion or limit."""
        self.memory.conversation_history.append(
            {"role": "user", "content": user_message})

        while self.memory.iteration_count < self.memory.
    max_iterations:
            self.memory.iteration_count += 1
            logger.info(
                f"Iteration {self.memory.iteration_count}: "
                f"Calling model...")

            response = self.client.messages.create(
                model=self.model, max_tokens=4096,
                system=self.system_prompt,
                tools=self.tools,
                messages=self.memory.conversation_history)

            self.memory.conversation_history.append(
```

```
                {"role": "assistant", "content": response.content})

            # Extract tool use blocks
            tool_use_blocks = [
                b for b in response.content
                if b.type == "tool_use"
            ]
            if not tool_use_blocks:
                # No tool calls: agent is done
                return "\n".join(
                    b.text for b in response.content
                    if b.type == "text")

            # Execute each tool and collect results
            tool_results = [
                self._execute_tool(tb.name, tb.input, tb.id)
                for tb in tool_use_blocks
            ]
            self.memory.conversation_history.append(
                {"role": "user", "content": tool_results})

        return "Agent reached maximum iteration limit."

    def _execute_tool(self, tool_name, tool_input, tool_use_id):
        """Execute a single tool with safety checks."""
        can_execute, reason = self.safety_policy.can_execute(
            tool_name)
        if not can_execute:
            logger.warning(f"BLOCKED: {tool_name}: {reason}")
            return {
                "type": "tool_result",
                "tool_use_id": tool_use_id,
                "content": f"BLOCKED: {reason}",
                "is_error": True,
            }

        executor = self.tool_executors.get(tool_name)
        if not executor:
            return {
                "type": "tool_result",
                "tool_use_id": tool_use_id,
                "content": f"Unknown tool: {tool_name}",
                "is_error": True,
            }
```

```
118
119         try:
120             result = executor(**tool_input)
121             self.memory.actions_taken.append({
122                 "tool": tool_name,
123                 "input": tool_input,
124                 "timestamp": datetime.now(
125                     timezone.utc).isoformat(),
126                 "success": True,
127             })
128             return {
129                 "type": "tool_result",
130                 "tool_use_id": tool_use_id,
131                 "content": (json.dumps(result)
132                             if not isinstance(result, str)
133                             else result),
134             }
135         except Exception as e:
136             self.memory.error_count += 1
137             logger.error(
138                 f"Tool {tool_name} failed: {e}")
139             return {
140                 "type": "tool_result",
141                 "tool_use_id": tool_use_id,
142                 "content": f"Failed: {str(e)}",
143                 "is_error": True,
144             }
```

20.5 Agent Memory and Error Recovery

Production agents need more than a conversation buffer. They require structured memory systems that persist across sessions and enable learning from past operations. Unlike human engineers, who persist knowledge via "that one Slack thread from 2019 that nobody can find."

Memory Types

Agents benefit from four types of memory:

Working Memory The current conversation context. This is the agent's "short-

term memory"—the messages exchanged during the current session. It is bounded by the model's context window.

Episodic Memory Past incidents and their resolutions, stored in a vector database for similarity search. When the agent encounters a new failure, it retrieves similar past incidents to inform its approach.

Semantic Memory Structured knowledge about the environment: pipeline dependency graphs, table schemas, SLA definitions, team ownership mappings. This is typically loaded into the system prompt or retrieved on demand.

Procedural Memory Runbooks and standard operating procedures, encoded as step-by-step instructions the agent can follow. These provide guardrails and ensure consistency with established practices.

Code Example

Episodic memory with vector search

```python
from dataclasses import dataclass
import numpy as np

@dataclass
class Episode:
    incident_type: str
    symptoms: str
    root_cause: str
    resolution: str
    embedding: list[float]
    timestamp: str

class EpisodicMemory:
    """Stores and retrieves past incident resolutions."""

    def __init__(self, embedding_service):
        self.episodes: list[Episode] = []
        self.embedding_service = embedding_service

    def record(self, incident_type, symptoms,
               root_cause, resolution):
        embedding = self.embedding_service.embed(
            f"{incident_type}: {symptoms}")
        self.episodes.append(Episode(
            incident_type=incident_type,
```

```
            symptoms=symptoms,
            root_cause=root_cause,
            resolution=resolution,
            embedding=embedding,
            timestamp=datetime.now(
                timezone.utc).isoformat(),
        ))

    def recall(self, current_symptoms, top_k=3):
        query_emb = self.embedding_service.embed(
            current_symptoms)
        scored = [
            (ep, self._cosine_sim(query_emb, ep.embedding))
            for ep in self.episodes
        ]
        scored.sort(key=lambda x: x[1], reverse=True)
        return scored[:top_k]

    @staticmethod
    def _cosine_sim(a, b):
        a, b = np.array(a), np.array(b)
        return float(np.dot(a, b) / (
            np.linalg.norm(a) * np.linalg.norm(b)))
```

Error Recovery Strategies

For error recovery, classify errors into three categories and handle each differently:

1. **Transient errors**: Network timeouts, rate limits, temporary service unavailability. Retry with exponential backoff, up to a configurable limit.

2. **Recoverable errors**: Permission denied (try a different credential), query syntax error (fix and retry), missing table (check alternative schemas). The agent can work around these by trying alternative approaches.

3. **Fatal errors**: Data corruption, security violations, unknown system states. The agent must stop immediately and escalate to a human operator.

Warning

Automatic retries can be dangerous for non-idempotent operations. If your tool inserts rows, retrying a partially-completed operation could create duplicates. "Oops, we now have 47 copies of every customer" is not a postmortem title you want to write. Always ensure retried operations are ***idempotent***—producing the same result whether executed once or multiple times. Use techniques like `INSERT ... ON CONFLICT DO NOTHING` or merge/upsert semantics.

Code Example

Error classification and recovery

```python
import time
from enum import Enum

class ErrorCategory(Enum):
    TRANSIENT = "transient"
    RECOVERABLE = "recoverable"
    FATAL = "fatal"

class ErrorClassifier:
    TRANSIENT_PATTERNS = [
        "timeout", "rate limit", "503",
        "connection reset", "temporarily unavailable",
    ]
    FATAL_PATTERNS = [
        "permission denied", "access denied",
        "data corruption", "integrity constraint",
    ]

    def classify(self, error_message: str) -> ErrorCategory:
        lower = error_message.lower()
        if any(p in lower for p in self.TRANSIENT_PATTERNS):
            return ErrorCategory.TRANSIENT
        if any(p in lower for p in self.FATAL_PATTERNS):
            return ErrorCategory.FATAL
        return ErrorCategory.RECOVERABLE

def retry_with_backoff(func, max_retries=3,
                       base_delay=1.0):
    """Retry a function with exponential backoff."""
    for attempt in range(max_retries):
```

```
        try:
            return func()
        except Exception as e:
            if attempt == max_retries - 1:
                raise
            delay = base_delay * (2 ** attempt)
            time.sleep(delay)
```

20.6 Safety Guardrails

Safety is not an afterthought—it is the foundation of any production agent. An agent with access to your data warehouse and pipeline orchestrator has the potential to cause significant damage if not properly constrained. Think of it like giving a toddler a flamethrower: impressive capability, questionable judgment. Every agent should implement these four layers of protection:

1. **Least privilege**: Grant only the minimum permissions required. A monitoring agent should have read-only database access. A remediation agent should have write access only to the specific resources it manages.
2. **Action budgets**: Limit the number of write operations per session. A runaway agent that continuously restarts tasks or modifies configurations can cause cascading failures.
3. **Human-in-the-loop checkpoints**: Require human approval for high-risk operations such as schema migrations, production deployments, or actions affecting multiple pipelines simultaneously.
4. **Audit logging**: Record every action in a tamper-resistant location—including the prompt, the model's reasoning, the tool called, the arguments provided, and the result.

Code Example

Action budget enforcement

```
class ActionBudget:
    """Enforce per-category action limits per session."""

    def __init__(self, budgets: dict[str, int]):
```

```
        self.budgets = budgets  # e.g. {"write": 5, "restart": 3}
        self.usage = {k: 0 for k in budgets}

    def can_perform(self, action_category: str) -> bool:
        if action_category not in self.budgets:
            return False
        return self.usage[action_category] < \
               self.budgets[action_category]

    def record(self, action_category: str):
        self.usage[action_category] = \
            self.usage.get(action_category, 0) + 1

    def remaining(self, action_category: str) -> int:
        if action_category not in self.budgets:
            return 0
        return self.budgets[action_category] - \
               self.usage.get(action_category, 0)

    def summary(self) -> str:
        lines = []
        for cat, budget in self.budgets.items():
            used = self.usage.get(cat, 0)
            lines.append(f"  {cat}: {used}/{budget} used")
        return "\n".join(lines)
```

Tip

Store audit logs in an append-only log store (e.g., an S3 bucket with object lock, or a write-once database table) that the agent itself cannot modify. This ensures that even a compromised or misbehaving agent cannot hide its tracks. The agent equivalent of "this conversation is being recorded for quality assurance purposes."

The Confirmation Gateway

For high-risk operations, implement a confirmation gateway that pauses the agent and waits for human approval. This can be implemented via Slack, email, or a custom approval UI.

Code Example

Human-in-the-loop confirmation

```
import asyncio

class ConfirmationGateway:
    """Pause agent execution for human approval."""

    def __init__(self, notifier, timeout_seconds=300):
        self.notifier = notifier  # Slack, email, etc.
        self.timeout = timeout_seconds

    async def request_approval(self, action_description,
                               context, risk_level):
        """Send approval request and wait for response."""
        request_id = self.notifier.send(
            channel="#data-ops-approvals",
            message=(
                f"*Agent Approval Request*\n"
                f"Action: {action_description}\n"
                f"Risk: {risk_level}\n"
                f"Context: {context}\n"
                f"Reply 'approve {request_id}' or "
                f"'deny {request_id}'"),
        )
        try:
            response = await asyncio.wait_for(
                self.notifier.wait_for_reply(request_id),
                timeout=self.timeout,
            )
            return response.strip().lower() == "approve"
        except asyncio.TimeoutError:
            return False  # Deny on timeout
```

Note

Timeout-on-deny is a critical safety pattern. If no human responds within the timeout window, the agent should treat the action as denied. Never default to approval on timeout—"nobody said no" is not the same as "yes." That logic does not work in engineering any more than it works in real life.

20.7 Multi-Step Pipeline Automation

The true power of agents emerges in multi-step workflows where each action depends on the results of prior actions. Consider a typical pipeline failure investigation:

1. Check the DAG status to identify which task failed
2. Retrieve the task logs to understand the error
3. Query the data warehouse to verify data state
4. Check if this is a known issue by searching episodic memory
5. Apply the fix (restart, rerun with parameters, or modify config)
6. Validate the fix by checking downstream data quality
7. Notify the team with a summary of the incident and resolution

An agent can execute this entire workflow autonomously, typically completing in 2–5 minutes what would take a human engineer 30–60 minutes—especially at 3 AM, when the human's cognitive abilities are roughly equivalent to a `SELECT * FROM brain WHERE caffeine > 0` returning zero rows.

Warning

Multi-step workflows amplify both the benefits and the risks of automation. A mistake in step 5 that goes undetected until step 7 can be difficult to unwind. Always include validation steps after write operations, and design rollback mechanisms for every automated fix.

20.8 Full Example: Schema Migration Agent

To illustrate all concepts in this chapter working together, here is a complete schema migration agent. This agent analyzes the requested change, discovers all downstream consumers, plans reversible migration steps, generates SQL with rollback scripts, tests on staging, and executes if approved.

Code Example

Schema migration agent — orchestration

```
class SchemaMigrationAgent(DataEngineeringAgent):
    """Specialized agent for safe schema migrations."""

    SYSTEM_PROMPT = """You are a schema migration specialist.
When given a schema change request:
1. Analyze the change and identify affected columns/tables
2. Discover all downstream consumers (views, dbt models,
   dashboards, applications) using the dependency tools
3. Plan a backward-compatible migration with these phases:
   - Add new columns/tables (non-breaking)
   - Backfill data into new structure
   - Update consumers to use new structure
   - Deprecate old columns with a removal date
4. Generate migration SQL with rollback scripts
5. Test on staging before requesting production approval
Never drop columns without a deprecation period.
Always create backward-compatible views."""

    def __init__(self, **kwargs):
        migration_tools = self._build_migration_tools()
        super().__init__(
            tools=migration_tools,
            system_prompt=self.SYSTEM_PROMPT,
            safety_policy=SafetyPolicy(
                require_confirmation_for={
                    "execute_migration_production",
                    "drop_column",
                },
                max_write_actions_per_session=10,
            ),
            **kwargs,
        )
```

Code Example

Example migration SQL generated by the agent

```
-- V042__split_customer_address.sql
```

```sql
-- Generated by SchemaMigrationAgent
-- Rollback: V042__split_customer_address_ROLLBACK.sql

-- Step 1: Add new columns (non-breaking)
ALTER TABLE ANALYTICS.PUBLIC.CUSTOMERS
ADD COLUMN street_address VARCHAR(500),
ADD COLUMN city VARCHAR(100),
ADD COLUMN state VARCHAR(50),
ADD COLUMN zip_code VARCHAR(20),
ADD COLUMN country VARCHAR(100);

-- Step 2: Backfill from existing data
UPDATE ANALYTICS.PUBLIC.CUSTOMERS
SET street_address = PARSE_ADDRESS(address, 'street'),
    city = PARSE_ADDRESS(address, 'city'),
    state = PARSE_ADDRESS(address, 'state'),
    zip_code = PARSE_ADDRESS(address, 'zip'),
    country = PARSE_ADDRESS(address, 'country')
WHERE address IS NOT NULL;

-- Step 3: Deprecation comment (sets removal date)
COMMENT ON COLUMN ANALYTICS.PUBLIC.CUSTOMERS.ADDRESS IS
    'DEPRECATED: Use street_address, city, state, zip_code, '
    'country instead. Will be removed after 2026-07-01.';

-- Step 4: Backward-compatible view for consumers
CREATE OR REPLACE VIEW ANALYTICS.PUBLIC.CUSTOMERS_V2 AS
SELECT customer_id, first_name, last_name, email,
    address,  -- kept for backward compatibility
    street_address, city, state, zip_code, country,
    created_at, updated_at
FROM ANALYTICS.PUBLIC.CUSTOMERS;

-- Step 5: Grant permissions to match original table
GRANT SELECT ON ANALYTICS.PUBLIC.CUSTOMERS_V2
TO ROLE ANALYTICS_READER;
```

Tip

The agent also generates a corresponding rollback script that reverses every step. Because CLAUDE CODE learned from humanity's greatest hits—and its greatest "oh no, roll it back" moments. In the example above, the rollback

would drop the new columns, remove the view, and restore the original column comment. Always verify that rollback scripts are tested alongside the migration itself.

20.9 Monitoring and Observability for Agents

Agents are software systems and require the same observability practices as any production service. Key metrics to track include:

- **Iterations per task**: How many agent loop iterations are needed to complete each task type. Increasing iteration counts may indicate degraded tool descriptions or model performance.
- **Token usage per task**: Total input and output tokens consumed. Track this by task type to identify unexpectedly expensive operations.
- **Tool call success rate**: The percentage of tool calls that succeed. A drop signals tool infrastructure issues.
- **Error escalation rate**: How often the agent escalates to human intervention. This is your primary measure of agent autonomy.
- **Time to resolution**: Wall-clock time from task submission to completion. Compare against manual resolution times to quantify ROI.

Note

Set up alerting on agent error escalation rate. If it rises above your baseline (typically 10–20% for well-tuned agents), investigate whether tool definitions need updating, the system prompt needs refinement, or the environment has changed in ways the agent does not understand. If the escalation rate hits 100%, congratulations: you have built a very expensive pager.

20.10 Exercises

1. **Data Quality Agent**: Create an agent that runs Great Expectations checkpoints, analyzes failures, determines root causes, and suggests fixes—distinguishing data issues (bad upstream data), code issues (incorrect transformation logic), and infrastructure issues (timeouts, resource exhaustion). The agent should produce a structured incident report for each failure.

2. **Circuit Breaker Pattern**: Extend the agent with a circuit breaker that disables a tool after three consecutive failures, resetting after a configurable cool-down period. Test this by simulating intermittent tool failures and verifying that the agent gracefully degrades.

3. **Cost-Aware Agent**: Track token usage per iteration, switch to a cheaper model (e.g., Haiku) when session cost exceeds a soft threshold, and terminate gracefully at a hard budget limit. Implement a cost report that summarizes token usage by tool call type.

4. **MCP Server Development**: Build an MCP server for a data system you use regularly (dbt, Databricks, Fivetran, etc.) with at least five tools covering both read and write operations. Connect it to a Claude Code agent and demonstrate an end-to-end workflow such as triggering a dbt run and validating the results.

5. **Multi-Agent Coordination**: Design and implement a system where a "triage agent" receives an alert, classifies it, and dispatches to specialized agents (pipeline agent, data quality agent, infrastructure agent). Implement a shared state store so agents can see each other's findings. Test with three simulated incident types.

21

RAG for Data Engineering

"RAG lets Claude Code look things up before answering. Before RAG, it just made things up confidently. So, like a senior engineer."

Tip

Retrieval-Augmented Generation (RAG) bridges the gap between Claude Code's broad knowledge and your organization's specific data engineering context. This chapter teaches you to build RAG systems that turn your team's runbooks, data catalogs, and incident histories into an always-available engineering assistant—one that actually reads the documentation, which already puts it ahead of most humans.

Large language models know a great deal about the world, but they do not know about *your* data warehouse, *your* pipeline architecture, or *your* on-call procedures. ***Retrieval-Augmented Generation*** solves this by dynamically injecting relevant context from your knowledge base into the prompt at query time. Instead of fine-tuning a model on your documentation—an expensive process that creates a snapshot that goes stale—RAG keeps the model's general capabilities intact while grounding every response in your current, authoritative sources.

The result is a system that can answer questions like "What is the SLA for the customer_events pipeline?" or "How do I recover from a Snowflake warehouse suspension?" by finding the relevant runbook paragraphs and synthesizing a precise answer—complete with source citations. No more asking Dave, who wrote that runbook three years ago, left the company, and took all the tribal knowledge with him. Thanks, Dave.

21.1 RAG Fundamentals

A RAG system has three core components:

1. **Indexing Pipeline**: Process source documents, split them into chunks, generate vector embeddings for each chunk, and store everything in a vector database. This runs offline (batch) or incrementally as documents change.
2. **Retrieval Engine**: At query time, embed the user's question, search the vector database for the most similar chunks, and optionally re-rank or filter results by metadata.
3. **Generation Engine**: Combine the retrieved chunks with the original question into a prompt, send it to Claude Code, and return the response along with source citations.

Warning

Fine-tuning is often the first approach teams consider, because humans love the most expensive option first. But for most data engineering use cases, RAG is superior. Fine-tuning is expensive (thousands of dollars per training run), creates models that go stale as your documentation evolves, can degrade the model's general capabilities, and makes it difficult to attribute answers to specific sources. RAG keeps the model intact while dynamically grounding it in current information. It's like giving the model an open-book exam instead of making it memorize the entire textbook.

RAG offers several advantages that make it the right default for knowledge-grounded applications:

- **Always current**: Re-index when documents change. No retraining required.
- **Attributable**: Every answer can cite its sources, enabling verification.
- **Composable**: Combine runbooks, data catalogs, incident reports, and Slack threads into a single searchable knowledge base.
- **Access-controlled**: Filter retrieved chunks by user permissions—an analyst should not see infrastructure runbooks marked as ops-only.
- **Debuggable**: When an answer is wrong, inspect the retrieved chunks to determine whether the problem is retrieval (wrong chunks found) or generation (model misinterpreted correct chunks).

Code Example

RAG pipeline data structures

```python
from dataclasses import dataclass, field
from typing import Any

@dataclass
class Document:
    """A source document before chunking."""
    content: str
    metadata: dict[str, Any]
    source_type: str  # "runbook", "catalog", "incident"
    source_id: str

@dataclass
class Chunk:
    """A chunk of a document with its embedding."""
    content: str
    metadata: dict[str, Any]
    chunk_index: int
    document_id: str
    embedding: list[float] | None = None
    token_count: int = 0

@dataclass
class RetrievalResult:
    """A ranked retrieval result."""
    chunk: Chunk
    relevance_score: float
    rank: int

@dataclass
class RAGResponse:
    """The final RAG response with provenance."""
    answer: str
    sources: list[RetrievalResult]
    confidence: float
    tokens_used: int
```

21.2 Building a Knowledge Base

The quality of a RAG system depends entirely on the quality and coverage of its knowledge base. For data engineering teams, the most valuable sources include:

- **Runbooks**: Step-by-step procedures for incident response, deployments, and maintenance tasks.
- **Data catalog entries**: Table descriptions, column definitions, ownership, SLAs, and lineage information.
- **Architecture documentation**: System diagrams, design decisions, and integration patterns.
- **Incident reports**: Post-mortems with root cause analysis and remediation steps.
- **dbt project documentation**: Model descriptions, tests, source definitions from `schema.yml` files.
- **Relevant Slack/Teams threads**: Curated discussions containing tribal knowledge.

Code Example

Multi-source document ingestion

```python
from abc import ABC, abstractmethod
import json, yaml, httpx

class DocumentSource(ABC):
    """Base class for document sources."""
    @abstractmethod
    def fetch_documents(self) -> list[Document]:
        ...

class ConfluenceSource(DocumentSource):
    """Ingest runbooks from Confluence."""
    def __init__(self, base_url, space_key, token):
        self.base_url = base_url
        self.space_key = space_key
        self.headers = {"Authorization": f"Bearer {token}"}

    def fetch_documents(self) -> list[Document]:
```

```
        docs = []
        url = (f"{self.base_url}/rest/api/content"
               f"?spaceKey={self.space_key}"
               f"&expand=body.storage,metadata.labels"
               f"&limit=100")
        resp = httpx.get(url, headers=self.headers)
        for page in resp.json()["results"]:
            labels = [l["name"] for l in
                      page.get("metadata", {})
                      .get("labels", {})
                      .get("results", [])]
            docs.append(Document(
                content=self._html_to_text(
                    page["body"]["storage"]["value"]),
                metadata={
                    "title": page["title"],
                    "labels": labels,
                    "last_updated": page["version"]["when"],
                    "author": page["version"]["by"]
                                  ["displayName"],
                },
                source_type="runbook",
                source_id=f"confluence:{page['id']}",
            ))
        return docs

    def _html_to_text(self, html: str) -> str:
        from bs4 import BeautifulSoup
        return BeautifulSoup(html, "html.parser").get_text(
            separator="\n")

class DbtSource(DocumentSource):
    """Ingest documentation from a dbt project."""
    def __init__(self, manifest_path: str):
        self.manifest_path = manifest_path

    def fetch_documents(self) -> list[Document]:
        with open(self.manifest_path) as f:
            manifest = json.load(f)
        docs = []
        for node_id, node in manifest["nodes"].items():
            if node["resource_type"] != "model":
                continue
            content_parts = [
```

```
                f"Model: {node['name']}",
                f"Description: {node.get('description', 'N/A')}",
                f"Schema: {node.get('schema', 'N/A')}",
                f"Materialization: "
                f"{node['config'].get('materialized', 'view')}",
            ]
            # Include column descriptions
            for col_name, col in node.get("columns", {}).items():
                desc = col.get("description", "")
                content_parts.append(
                    f"  Column {col_name}: {desc}")
            docs.append(Document(
                content="\n".join(content_parts),
                metadata={
                    "model_name": node["name"],
                    "schema": node.get("schema"),
                    "tags": node.get("tags", []),
                    "depends_on": node.get("depends_on", {})
                                      .get("nodes", []),
                },
                source_type="catalog",
                source_id=f"dbt:{node_id}",
            ))
        return docs

class IncidentReportSource(DocumentSource):
    """Ingest incident reports from a directory."""
    def __init__(self, directory: str):
        self.directory = directory

    def fetch_documents(self) -> list[Document]:
        from pathlib import Path
        docs = []
        for path in Path(self.directory).glob("*.md"):
            content = path.read_text()
            # Parse YAML front matter if present
            metadata = {}
            if content.startswith("---"):
                parts = content.split("---", 2)
                if len(parts) >= 3:
                    metadata = yaml.safe_load(parts[1])
                    content = parts[2]
            docs.append(Document(
                content=content,
```

```
                metadata=metadata,
                source_type="incident",
                source_id=f"incident:{path.stem}",
            ))
        return docs
```

Tip

Schedule your indexing pipeline to run on a cron (e.g., every 6 hours) and track document checksums to avoid re-indexing unchanged documents. For Confluence and similar wikis, use the "last modified" timestamp to fetch only updated pages since the last indexing run. Yes, you're building a pipeline to feed a pipeline. Welcome to data engineering.

21.3 Embedding Models and Vector Stores

Embeddings are dense vector representations of text that capture semantic meaning. Two pieces of text with similar meanings will have embeddings that are close together in vector space, even if they use completely different words. This property enables ***semantic search***—finding relevant content based on meaning rather than keyword matching.

Choosing an Embedding Model

Several factors influence embedding model selection:

- **Dimension size**: Higher dimensions (e.g., 1024 or 1536) capture more nuance but require more storage and compute. For most data engineering documentation, 1024 dimensions provide an excellent balance.
- **Context window**: Ensure the model's context window accommodates your chunk sizes. Most modern embedding models support 512–8192 tokens.
- **Domain performance**: Some models perform better on technical content. Test with a sample of your actual documents.
- **Cost and latency**: Embedding is a high-volume operation during indexing. Factor in cost per token and throughput.

Code Example

Embedding service wrapper

```python
import voyageai
from typing import Optional

class EmbeddingService:
    """Wrapper for generating text embeddings."""

    def __init__(self, model: str = "voyage-3",
                 batch_size: int = 32):
        self.client = voyageai.Client()
        self.model = model
        self.batch_size = batch_size

    def embed_query(self, text: str) -> list[float]:
        """Embed a single query string."""
        result = self.client.embed(
            [text], model=self.model,
            input_type="query")
        return result.embeddings[0]

    def embed_documents(self,
                        texts: list[str]) -> list[list[float]]:
        """Embed a batch of document chunks."""
        all_embeddings = []
        for i in range(0, len(texts), self.batch_size):
            batch = texts[i:i + self.batch_size]
            result = self.client.embed(
                batch, model=self.model,
                input_type="document")
            all_embeddings.extend(result.embeddings)
        return all_embeddings
```

Note

Many embedding APIs distinguish between "query" and "document" input types. Using the correct type improves retrieval quality because the model optimizes the embedding differently for short queries versus longer document passages. Always use `input_type="query"` for user questions and `input_type="document"` for chunks being indexed.

Vector Store Implementation

Code Example

Vector store with pgvector

```
import psycopg2, json
from psycopg2.extras import execute_values

class PgVectorStore:
    """Vector store backed by PostgreSQL with pgvector."""

    def __init__(self, connection_string: str,
                 embedding_dim: int = 1024):
        self.conn = psycopg2.connect(connection_string)
        self.embedding_dim = embedding_dim
        self._ensure_schema()

    def _ensure_schema(self):
        with self.conn.cursor() as cur:
            cur.execute(
                "CREATE EXTENSION IF NOT EXISTS vector;")
            cur.execute(f"""
                CREATE TABLE IF NOT EXISTS document_chunks (
                    id SERIAL PRIMARY KEY,
                    content TEXT NOT NULL,
                    metadata JSONB NOT NULL DEFAULT '{{}}',
                    source_type VARCHAR(50) NOT NULL,
                    source_id VARCHAR(255) NOT NULL,
                    chunk_index INTEGER NOT NULL,
                    embedding vector({self.embedding_dim}),
                    created_at TIMESTAMPTZ DEFAULT NOW(),
                    UNIQUE (source_id, chunk_index)
                );""")
            cur.execute("""
                CREATE INDEX IF NOT EXISTS idx_chunks_embedding
                ON document_chunks
                USING ivfflat (embedding vector_cosine_ops)
                WITH (lists = 100);""")
            cur.execute("""
                CREATE INDEX IF NOT EXISTS idx_chunks_source
                ON document_chunks (source_type, source_id);
                """)
            self.conn.commit()
```

```
39
40      def upsert(self, chunks: list[dict]):
41          """Insert or update chunks with embeddings."""
42          with self.conn.cursor() as cur:
43              for chunk in chunks:
44                  cur.execute("""
45                      INSERT INTO document_chunks
46                          (content, metadata, source_type,
47                           source_id, chunk_index, embedding)
48                      VALUES (%s, %s, %s, %s, %s, %s)
49                      ON CONFLICT (source_id, chunk_index)
50                      DO UPDATE SET
51                          content = EXCLUDED.content,
52                          metadata = EXCLUDED.metadata,
53                          embedding = EXCLUDED.embedding,
54                          created_at = NOW()
55                  """, (
56                      chunk["content"],
57                      json.dumps(chunk["metadata"]),
58                      chunk["source_type"],
59                      chunk["source_id"],
60                      chunk["chunk_index"],
61                      chunk["embedding"],
62                  ))
63              self.conn.commit()
64
65      def search(self, query_embedding, top_k=10,
66                 source_type_filter=None,
67                 metadata_filter=None):
68          """Semantic search with optional filters."""
69          conditions = ["1=1"]
70          params = [str(query_embedding)]
71          if source_type_filter:
72              conditions.append("source_type = %s")
73              params.append(source_type_filter)
74          if metadata_filter:
75              for key, value in metadata_filter.items():
76                  conditions.append(
77                      f"metadata->>'{key}' = %s")
78                  params.append(str(value))
79          where_clause = " AND ".join(conditions)
80          params.extend([str(query_embedding), top_k])
81
82          with self.conn.cursor() as cur:
```

```
            cur.execute(f"""
                SELECT content, metadata, source_type,
                       source_id, chunk_index,
                       1 - (embedding <=> %s::vector)
                           AS similarity
                FROM document_chunks
                WHERE {where_clause}
                ORDER BY embedding <=> %s::vector
                LIMIT %s""", params)
            results = []
            for rank, row in enumerate(cur.fetchall()):
                chunk = Chunk(
                    content=row[0],
                    metadata=row[1],
                    chunk_index=row[4],
                    document_id=row[3])
                results.append(RetrievalResult(
                    chunk=chunk,
                    relevance_score=float(row[5]),
                    rank=rank + 1))
        return results

    def delete_by_source(self, source_id: str):
        """Remove all chunks for a given source."""
        with self.conn.cursor() as cur:
            cur.execute(
                "DELETE FROM document_chunks "
                "WHERE source_id = %s", (source_id,))
            self.conn.commit()
```

Warning

The `ivfflat` index requires the table to have data before it can be effectively built. If you create the index on an empty table and then bulk-load data, query performance will be poor. Either create the index after the initial data load, or use `REINDEX` after bulk loading. For tables with fewer than 10,000 chunks, an exact (brute-force) search without an index may actually be faster.

21.4 Chunking Strategies for Technical Documentation

Chunking—the process of splitting documents into smaller pieces for embedding and retrieval—is one of the most impactful design decisions in a RAG system. It is also one of the least glamorous. Nobody has ever won an award for "Best Chunking Strategy," but bad chunking has certainly caused award-winning postmortems. Poor chunking leads to irrelevant retrieval, which leads to wrong answers, regardless of how good your embedding model or LLM is.

Technical documents present unique chunking challenges:

- **Code blocks** should never be split mid-statement. A Python function split across two chunks loses its meaning in both.
- **Tables** are meaningless without their headers. Each table should be kept as a single chunk or include the header row in every chunk.
- **Step-by-step instructions** lose procedural context when split. Step 5 without steps 1–4 is often useless.
- **Configuration examples** are meaningless if truncated. A half-complete YAML config is worse than no config.

Recommended Strategies

Structure-aware chunking Parse the document's structure (Markdown headers, HTML tags) and split at natural boundaries. Each chunk corresponds to a section or subsection. This preserves the author's intended information grouping and works well for technical documentation.

Parent-child chunking Create small chunks (200–300 tokens) for retrieval precision, but link each to a larger parent chunk (800–1200 tokens). Search against the small chunks, but return the parent when assembling context for the LLM. This gives you the best of both worlds: precise matching and rich context.

Sliding window with overlap Use fixed-size windows (e.g., 500 tokens) with 10–20% overlap between consecutive chunks. The overlap ensures that information at chunk boundaries is not lost. This is the simplest strategy and works adequately for prose-heavy documents, but performs poorly on code-heavy content.

Code Example

Structure-aware Markdown chunker

```
import re
from dataclasses import dataclass

@dataclass
class MarkdownSection:
    heading: str
    level: int
    content: str
    parent_heading: str | None = None

class MarkdownChunker:
    """Split Markdown documents at heading boundaries."""

    def __init__(self, max_chunk_tokens: int = 800,
                 min_chunk_tokens: int = 100):
        self.max_tokens = max_chunk_tokens
        self.min_tokens = min_chunk_tokens

    def chunk(self, document: Document) -> list[Chunk]:
        sections = self._parse_sections(document.content)
        chunks = []
        for i, section in enumerate(sections):
            # Prepend heading hierarchy for context
            header = section.heading
            if section.parent_heading:
                header = (f"{section.parent_heading} > "
                          f"{section.heading}")
            chunk_content = f"# {header}\n\n{section.content}"

            # Split oversized sections
            if self._estimate_tokens(chunk_content) \
                    > self.max_tokens:
                sub_chunks = self._split_large_section(
                    chunk_content)
                for j, sc in enumerate(sub_chunks):
                    chunks.append(Chunk(
                        content=sc,
                        metadata={
                            **document.metadata,
                            "heading": header},
```

```
                        chunk_index=len(chunks),
                        document_id=document.source_id))
            else:
                chunks.append(Chunk(
                    content=chunk_content,
                    metadata={
                        **document.metadata,
                        "heading": header},
                    chunk_index=len(chunks),
                    document_id=document.source_id))
        return chunks

    def _parse_sections(self, text):
        lines = text.split("\n")
        sections = []
        current_heading = "Introduction"
        current_content = []
        parent = None
        current_level = 1

        for line in lines:
            heading_match = re.match(
                r'^(#{1,4})\s+(.+)$', line)
            if heading_match:
                if current_content:
                    sections.append(MarkdownSection(
                        heading=current_heading,
                        level=current_level,
                        content="\n".join(current_content),
                        parent_heading=parent))
                level = len(heading_match.group(1))
                current_heading = heading_match.group(2)
                current_level = level
                if level > 1:
                    parent = (sections[-1].heading
                              if sections else None)
                else:
                    parent = None
                current_content = []
            else:
                current_content.append(line)

        if current_content:
            sections.append(MarkdownSection(
```

```
                heading=current_heading,
                level=current_level,
                content="\n".join(current_content),
                parent_heading=parent))
        return sections

    def _estimate_tokens(self, text: str) -> int:
        return len(text) // 4  # rough estimate

    def _split_large_section(self, text: str) -> list[str]:
        paragraphs = text.split("\n\n")
        chunks, current = [], []
        current_size = 0
        for para in paragraphs:
            para_tokens = self._estimate_tokens(para)
            if (current_size + para_tokens > self.max_tokens
                    and current):
                chunks.append("\n\n".join(current))
                current = [para]
                current_size = para_tokens
            else:
                current.append(para)
                current_size += para_tokens
        if current:
            chunks.append("\n\n".join(current))
        return chunks
```

Tip

For code-heavy documentation, create separate chunks for code blocks and their surrounding explanatory text, linked by metadata. This lets the retrieval engine find code examples directly while keeping explanations available for context. Store the code language (Python, SQL, YAML) in chunk metadata so you can filter retrieval by language when appropriate.

21.5 RAG for Runbook Automation

Runbooks are the most immediately valuable content for a data engineering RAG system. Every team has runbooks—step-by-step procedures for handling common incidents, performing maintenance tasks, and executing deployments. But runbooks

suffer from chronic problems: they are scattered across wikis, they go stale, and engineers under pressure at 3 AM cannot find the right one.

A RAG-powered runbook assistant solves all three problems: it searches across all sources, it surfaces the most relevant procedure regardless of where it lives, and it synthesizes information from multiple runbooks when no single document covers the situation. It is, in essence, the colleague who has actually read every page of every runbook—a mythical creature that, until now, did not exist.

Code Example

RAG-powered runbook assistant

```
import anthropic

class RunbookRAGAssistant:
    """A RAG assistant specialized for runbook queries."""

    SYSTEM_PROMPT = """You are a data engineering operations
assistant with access to the team's runbooks, documentation,
and incident history.

When answering questions:
1. Search the knowledge base for relevant runbooks
2. Present steps in order with warnings and prerequisites
3. Always cite your sources with document titles
4. If you find conflicting information across sources,
   flag the conflict and present both versions
5. If no relevant information exists, say so clearly
   and suggest who to contact

Never fabricate procedures. If a step seems incomplete
in the source material, note it explicitly."""

    def __init__(self, vector_store, embedding_service):
        self.client = anthropic.Anthropic()
        self.vector_store = vector_store
        self.embedding_service = embedding_service

    def answer(self, question, source_filter=None,
               top_k=8):
        """Answer a question using RAG retrieval."""
        query_embedding = self.embedding_service.embed_query(
            question)
```

```
        results = self.vector_store.search(
            query_embedding=query_embedding,
            top_k=top_k,
            source_type_filter=source_filter)

        if not results:
            return RAGResponse(
                answer="No relevant documents found in the "
                       "knowledge base for this question.",
                sources=[], confidence=0.0, tokens_used=0)

        # Build context from retrieved chunks
        context_parts = []
        for r in results:
            source_info = (
                f"[Source: "
                f"{r.chunk.metadata.get('title',
                    r.chunk.document_id)} "
                f"(relevance: {r.relevance_score:.2f})]")
            context_parts.append(
                f"{source_info}\n{r.chunk.content}")
        context = "\n\n---\n\n".join(context_parts)

        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096,
            system=self.SYSTEM_PROMPT,
            messages=[{
                "role": "user",
                "content": (
                    f"Context from knowledge base:"
                    f"\n\n{context}\n\n---\n\n"
                    f"Question: {question}")
            }])

        # Calculate confidence from retrieval scores
        top_score = results[0].relevance_score
        avg_score = sum(
            r.relevance_score for r in results[:3]
        ) / min(3, len(results))

        return RAGResponse(
            answer=response.content[0].text,
            sources=results,
```

```
            confidence=(top_score * 0.6 + avg_score * 0.4),
            tokens_used=(response.usage.input_tokens
                         + response.usage.output_tokens))
```

21.6 RAG for Data Catalog Search

Traditional data catalog search relies on keyword matching—if you do not know the exact table name or column name, you cannot find what you need. RAG enables ***semantic search*** over your catalog, allowing users to describe what they need in natural language and find the right tables.

A user can ask "I need customer purchase history with product details" and find the `orders` table joined with `order_items` and `products`, even if they do not know any of those table names. The embedding model captures the semantic relationship between "purchase history" and "orders."

Note

Semantic catalog search is particularly powerful for new team members who know *what* data they need but not *where* it lives. It also helps experienced engineers discover datasets they did not know existed—a common scenario in organizations with hundreds or thousands of tables, roughly 40% of which are named `temp_final_v2_REAL_USE_THIS_ONE`.

Code Example

Semantic catalog search with metadata enrichment

```
class CatalogSearchAssistant:
    """Semantic search over data catalog entries."""

    SYSTEM_PROMPT = """You are a data catalog assistant.
When a user describes the data they need:
1. Search for relevant tables and columns
2. Explain what each table contains and its freshness
3. Suggest joins between tables when appropriate
4. Note any data quality caveats or access restrictions
5. Provide example SQL queries to get started"""

```

```
    def __init__(self, vector_store, embedding_service):
        self.client = anthropic.Anthropic()
        self.vector_store = vector_store
        self.embedding_service = embedding_service

    def search(self, description: str) -> RAGResponse:
        embedding = self.embedding_service.embed_query(
            description)
        results = self.vector_store.search(
            query_embedding=embedding,
            top_k=15,
            source_type_filter="catalog")

        context = "\n\n".join(
            f"Table: {r.chunk.metadata.get('model_name', '?')}"
            f"\n{r.chunk.content}"
            for r in results)

        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=2048,
            system=self.SYSTEM_PROMPT,
            messages=[{
                "role": "user",
                "content": (
                    f"Available tables:\n\n{context}"
                    f"\n\nI need: {description}")
            }])
        return RAGResponse(
            answer=response.content[0].text,
            sources=results,
            confidence=results[0].relevance_score
                       if results else 0.0,
            tokens_used=(response.usage.input_tokens
                         + response.usage.output_tokens))
```

21.7 Hybrid Search: Combining Vector and Keyword

Pure vector search excels at finding semantically similar content but can miss exact matches on technical terms, table names, or error codes. Pure keyword search (BM25) finds exact matches but misses semantic relationships. ***Hybrid search***

combines both approaches for the best of both worlds.

Code Example

Hybrid search combining vector and BM25

```python
class HybridSearcher:
    """Combine vector similarity with keyword matching."""

    def __init__(self, vector_store, embedding_service,
                 vector_weight=0.7, keyword_weight=0.3):
        self.vector_store = vector_store
        self.embedding_service = embedding_service
        self.vector_weight = vector_weight
        self.keyword_weight = keyword_weight

    def search(self, query: str, top_k: int = 10,
               source_type_filter=None):
        # Vector search
        embedding = self.embedding_service.embed_query(query)
        vector_results = self.vector_store.search(
            query_embedding=embedding,
            top_k=top_k * 2,  # fetch more for re-ranking
            source_type_filter=source_type_filter)

        # Keyword search (using PostgreSQL full-text search)
        keyword_results = self.vector_store.keyword_search(
            query=query, top_k=top_k * 2,
            source_type_filter=source_type_filter)

        # Reciprocal Rank Fusion (RRF) to merge results
        scores: dict[str, float] = {}
        chunk_map: dict[str, RetrievalResult] = {}

        for r in vector_results:
            key = f"{r.chunk.document_id}:{r.chunk.chunk_index}"
            rrf = self.vector_weight / (60 + r.rank)
            scores[key] = scores.get(key, 0) + rrf
            chunk_map[key] = r

        for r in keyword_results:
            key = f"{r.chunk.document_id}:{r.chunk.chunk_index}"
            rrf = self.keyword_weight / (60 + r.rank)
            scores[key] = scores.get(key, 0) + rrf
```

```
            if key not in chunk_map:
                chunk_map[key] = r

        # Sort by combined score and return top_k
        ranked = sorted(scores.items(),
                        key=lambda x: x[1], reverse=True)
        return [
            RetrievalResult(
                chunk=chunk_map[key].chunk,
                relevance_score=score,
                rank=i + 1)
            for i, (key, score) in enumerate(ranked[:top_k])
        ]
```

Tip

Reciprocal Rank Fusion (RRF) is a simple but effective method for combining ranked lists from different retrieval strategies. The constant `k=60` in the RRF formula `1/(k + rank)` controls how much weight is given to top-ranked results versus lower-ranked ones. Higher values of `k` flatten the distribution; lower values amplify the top results.

21.8 Evaluation Metrics for RAG Systems

A RAG system that cannot be measured cannot be improved. Evaluation spans both the retrieval stage and the generation stage, and each requires different metrics.

Retrieval Metrics

Precision@K Of the K chunks retrieved, how many are actually relevant? Low precision means the LLM receives noise that can lead to incorrect answers.

Recall@K Of all relevant chunks in the corpus, how many appear in the top K results? Low recall means the LLM is missing information it needs.

Mean Reciprocal Rank (MRR) How high does the first relevant result rank? An MRR close to 1.0 means the most relevant chunk is almost always the top result.

Generation Metrics

Faithfulness Does the answer accurately reflect the retrieved context? Hallucinated information not present in the retrieved chunks indicates a faithfulness problem.

Relevance Does the answer actually address the user's question? A faithful but off-topic answer is not useful.

Completeness Does the answer cover all aspects of the question? Partial answers indicate retrieval gaps or generation truncation.

Attribution Can every claim in the answer be traced to a specific source? Unattributed claims are unverifiable.

Code Example

Automated RAG evaluation

```python
class RAGEvaluator:
    """Evaluate RAG system quality."""

    def __init__(self, rag_assistant):
        self.assistant = rag_assistant
        self.client = anthropic.Anthropic()

    def evaluate_batch(self,
                       test_cases: list[dict]) -> dict:
        """Run evaluation on a set of test cases.

        Each test case has: question, expected_answer,
        relevant_source_ids.
        """
        results = {
            "precision_at_5": [],
            "recall_at_5": [],
            "faithfulness": [],
            "relevance": [],
        }
        for case in test_cases:
            response = self.assistant.answer(
                case["question"], top_k=5)

            # Retrieval metrics
```

```
            retrieved_ids = {
                r.chunk.document_id
                for r in response.sources[:5]}
            relevant_ids = set(case["relevant_source_ids"])
            hits = retrieved_ids & relevant_ids
            precision = (len(hits) / len(retrieved_ids)
                         if retrieved_ids else 0)
            recall = (len(hits) / len(relevant_ids)
                      if relevant_ids else 0)
            results["precision_at_5"].append(precision)
            results["recall_at_5"].append(recall)

            # LLM-judged faithfulness and relevance
            judge_resp = self.client.messages.create(
                model="claude-sonnet-4-20250514",
                max_tokens=200,
                messages=[{"role": "user", "content":
                    f"Rate 1-5.\n"
                    f"Question: {case['question']}\n"
                    f"Answer: {response.answer}\n"
                    f"Expected: {case['expected_answer']}\n"
                    f"Faithfulness (1-5): "
                    f"Relevance (1-5):"}])
            # Parse scores from response
            scores = self._parse_scores(
                judge_resp.content[0].text)
            results["faithfulness"].append(
                scores.get("faithfulness", 3))
            results["relevance"].append(
                scores.get("relevance", 3))

        # Average all metrics
        return {k: sum(v) / len(v)
                for k, v in results.items() if v}

    def _parse_scores(self, text):
        scores = {}
        for line in text.split("\n"):
            lower = line.lower()
            for metric in ["faithfulness", "relevance"]:
                if metric in lower:
                    nums = re.findall(r'\d', line)
                    if nums:
                        scores[metric] = int(nums[0])
```

```
        return scores
```

Warning

LLM-as-judge evaluations are useful for rapid iteration but not perfect. Having an LLM grade its own work is like having a student grade their own exam—surprisingly useful, but don't bet your GPA on it. Models can exhibit position bias (favoring the first option), verbosity bias (preferring longer answers), and self-preference bias (rating their own outputs higher). For critical production systems, combine LLM-based evaluation with human evaluation on a regular cadence—monthly reviews of 50–100 randomly sampled queries provide a reliable ground truth signal.

Note

Build your evaluation dataset incrementally. Start with 30 question-answer pairs covering your most common query types. Add 10 new pairs each month, focusing on queries where the system performed poorly. After six months, you will have a comprehensive benchmark that captures the diversity of real-world usage patterns.

21.9 Production Deployment Considerations

Moving a RAG system from prototype to production introduces several concerns that do not arise during development:

- **Latency**: Embedding the query, searching the vector store, and generating the response adds 2–5 seconds of latency. For interactive use cases, consider pre-computing embeddings for common queries and caching frequent results.
- **Freshness**: Documents change. Your indexing pipeline must detect and re-index modified documents within your freshness SLA. Stale retrieval results erode user trust quickly.
- **Access control**: Not all users should see all chunks. Implement metadata-based filtering that respects your organization's data classification and role-based access control.

- **Cost management**: Each query involves an embedding API call and an LLM API call. At 100 queries per day, costs are negligible; at 10,000 queries per day, they require budgeting and optimization.

Tip

Implement a feedback mechanism where users can thumbs-up or thumbs-down RAG responses. Track the thumbs-down rate by source type and query category. This provides a real-time quality signal that complements your formal evaluation metrics and helps prioritize improvements.

21.10 Exercises

1. **Multi-Source Indexer**: Build an indexing pipeline that pulls documents from at least three sources (Confluence, a dbt project, and incident reports). Index into a pgvector store and verify that search returns relevant results for ten test queries spanning all three source types.

2. **Hybrid Search**: Extend the vector store with PostgreSQL full-text search to implement hybrid retrieval using Reciprocal Rank Fusion. Compare retrieval quality (Precision@5 and Recall@5) of pure vector, pure keyword, and hybrid search on 20 test queries. Document which query types benefit most from each approach.

3. **Conversational RAG**: Extend the runbook assistant to handle multi-turn conversations where follow-up questions depend on previous context. For example: "What tables have customer data?" followed by "Which of those are updated daily?" The system should rewrite follow-up questions to be self-contained before embedding.

4. **Evaluation Dataset**: Create 30+ question-answer pairs spanning runbooks (operational), catalog entries (discovery), and incident reports (diagnostic). Run the evaluator, identify the weakest category, and implement at least one improvement (better chunking, hybrid search, or prompt refinement) that measurably improves scores.

5. **Access-Controlled RAG**: Implement a metadata-based access control system where chunks are tagged with team ownership and sensitivity levels. Build a search endpoint that accepts a user role and filters results accordingly. Verify that an "analyst" role cannot retrieve ops-only runbooks.

22

Real-Time Data Processing with Claude Code

"Real-time data processing: because batch was too easy, and you had too much free time."

Real-time data processing is one of the most demanding domains in data engineering. This chapter explores how Claude Code can assist in building, optimizing, and maintaining real-time systems—from generating stream processing logic in Apache Flink and ksqlDB, to designing anomaly detection algorithms, to building fraud detection pipelines. We also examine the practical constraints of integrating LLMs into latency-sensitive streaming architectures. Spoiler: you cannot call an LLM for every event when you have a million events per second. Math is cruel that way.

22.1 Real-Time Data Engineering Fundamentals

Defining Real-Time

We distinguish three tiers of "real-time":

- **Hard real-time**: Processing under 10ms. Common in embedded systems and trading. LLMs are not suitable for this path. Claude Code is many things, but "sub-10ms" is not one of them.

- **Soft real-time**: Processing under 1 second, with occasional delays tolerable. Common in web applications and monitoring.
- **Near real-time**: Processing within seconds to minutes. Common in dashboards, alerting, and data sync. This is where Claude Code is most useful in the processing loop. Also where most "real-time" systems actually live, despite what the marketing page says.

Stream Processing Architecture Patterns

Event sourcing. Every state change is captured as an immutable event; current state is derived by replaying events.

CQRS. Write and read operations use separate models, often with different stores optimized for their access patterns.

Kappa architecture. All data flows through a single stream processing layer, simplifying the architecture compared to Lambda's separate batch and speed layers.

Key Concepts in Stream Processing

- ***Event time*** vs. ***processing time***: Correctly handling the gap between when events occur and when they are processed is a core challenge.
- ***Windowing***: Grouping events into finite sets—tumbling (fixed, non-overlapping), sliding (fixed, overlapping), and session (gap-based) windows.
- ***Watermarks***: Mechanisms for tracking event-time progress and handling late-arriving data.
- ***Exactly-once semantics***: Guaranteeing each event is processed exactly once despite failures and retries.
- ***Backpressure***: Handling situations where producers outpace consumers.

22.2 Kafka, Kinesis, and Pub/Sub Overview

Apache Kafka is the most widely adopted event streaming platform, offering durable ordered logs, consumer groups, Schema Registry, Kafka Connect, and ksqlDB. ***Amazon Kinesis*** differs with shard-based capacity, 24-hour default retention, and native AWS integration. ***Google Cloud Pub/Sub*** provides automatic scaling, at-least-once delivery, and native Dataflow integration.

Tip

When asking Claude Code to generate stream processing code, always specify which platform you are using. Saying "write me a streaming pipeline" without specifying the platform is like ordering "food" at a restaurant. The APIs, configuration parameters, and best practices differ significantly.

22.3 Claude Code for Stream Processing Logic Generation

Stream processing code is complex, with subtle correctness requirements around windowing, state management, and error handling. Claude Code can generate both Flink applications and ksqlDB queries from high-level descriptions.

Generating Apache Flink Applications

Apache Flink supports complex event processing, stateful computations, and exactly-once semantics. Claude Code can generate Flink application logic from high-level descriptions. The key elements to specify in your prompt are: the source topic and serialization format, the windowing strategy, the aggregation logic, and the sink configuration. Always include checkpointing configuration for production—Claude Code generates this correctly when you mention exactly-once requirements.

Code Example

```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.window import TumblingEventTimeWindows
from pyflink.common.time import Time
from pyflink.common.watermark_strategy import WatermarkStrategy

def build_order_analytics_pipeline():
    """Build a Flink streaming pipeline for order analytics."""
    env = StreamExecutionEnvironment.get_execution_environment()
    env.set_parallelism(4)
    env.enable_checkpointing(60000)

    from pyflink.datastream.connectors.kafka import (
        KafkaSource, KafkaOffsetsInitializer,
    )
    from pyflink.common.serialization import SimpleStringSchema
```

```

    kafka_source = (
        KafkaSource.builder()
        .set_bootstrap_servers("kafka:9092")
        .set_topics("orders")
        .set_group_id("order-analytics")
        .set_starting_offsets(KafkaOffsetsInitializer.earliest())
        .set_value_only_deserializer(SimpleStringSchema())
        .build()
    )

    watermark_strategy = (
        WatermarkStrategy
        .for_bounded_out_of_orderness(Time.seconds(30))
    )

    order_stream = (
        env.from_source(kafka_source, watermark_strategy, "Orders")
        .map(OrderParser())
        .key_by(lambda x: x[0])  # Key by customer_id
        .window(TumblingEventTimeWindows.of(Time.minutes(5)))
        .process(OrderAggregator())
    )
    env.execute("Order Analytics Pipeline")
```

Generating ksqlDB Queries

ksqlDB provides a SQL-like interface for stream processing on Kafka. CLAUDE CODE is particularly effective here because of the similarity to standard SQL:

Code Example

```
-- ksqlDB stream processing for real-time analytics

-- Create a stream from the raw orders topic
CREATE STREAM orders_raw (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    amount DOUBLE,
    status VARCHAR,
```

```
    product_category VARCHAR,
    event_time TIMESTAMP
) WITH (
    KAFKA_TOPIC = 'orders',
    VALUE_FORMAT = 'JSON',
    TIMESTAMP = 'event_time'
);

-- Filtered stream of completed orders
CREATE STREAM completed_orders AS
    SELECT order_id, customer_id, amount,
           product_category, event_time
    FROM orders_raw
    WHERE status = 'completed'
    EMIT CHANGES;

-- Tumbling window aggregation
CREATE TABLE revenue_per_category_5min AS
    SELECT
        product_category,
        WINDOWSTART AS window_start,
        COUNT(*) AS order_count,
        SUM(amount) AS total_revenue,
        AVG(amount) AS avg_order_value
    FROM completed_orders
    WINDOW TUMBLING (SIZE 5 MINUTES)
    GROUP BY product_category
    EMIT CHANGES;

-- Detect rapid ordering patterns (potential fraud)
CREATE TABLE rapid_orders AS
    SELECT
        customer_id,
        WINDOWSTART AS window_start,
        COUNT(*) AS order_count,
        SUM(amount) AS total_amount
    FROM orders_raw
    WINDOW HOPPING (SIZE 5 MINUTES, ADVANCE BY 1 MINUTE)
    GROUP BY customer_id
    HAVING COUNT(*) >= 10
    EMIT CHANGES;
```

22.4 Real-Time Anomaly Detection with Claude Code

Anomaly detection in streaming data requires statistical methods, domain knowledge, and adaptive thresholds. Claude Code can help design detection logic and tune parameters.

Code Example

```
import math
from collections import deque
from dataclasses import dataclass
from datetime import datetime


@dataclass
class AnomalyResult:
    is_anomaly: bool
    value: float
    expected_range: tuple[float, float]
    z_score: float
    timestamp: datetime
    metric_name: str
    severity: str = "info"


class StreamingAnomalyDetector:
    """Real-time anomaly detection using Welford's algorithm."""

    def __init__(
        self, window_size: int = 1000,
        z_threshold: float = 3.0,
        min_samples: int = 30,
    ):
        self.window_size = window_size
        self.z_threshold = z_threshold
        self.min_samples = min_samples
        self.count = 0
        self.mean = 0.0
        self.m2 = 0.0
        self.window: deque[float] = deque(maxlen=window_size)

    def update(
        self, value: float, timestamp: datetime,
```

```
        metric_name: str = "default",
    ) -> AnomalyResult:
        """Process a new value and check for anomalies."""
        self.count += 1
        delta = value - self.mean
        self.mean += delta / self.count
        delta2 = value - self.mean
        self.m2 += delta * delta2
        self.window.append(value)

        if self.count < self.min_samples:
            return AnomalyResult(
                is_anomaly=False, value=value,
                expected_range=(0.0, 0.0), z_score=0.0,
                timestamp=timestamp, metric_name=metric_name,
            )

        variance = self.m2 / (self.count - 1)
        std_dev = math.sqrt(variance) if variance > 0 else 1.0
        z_score = abs((value - self.mean) / std_dev)
        is_anomaly = z_score > self.z_threshold

        lower = self.mean - self.z_threshold * std_dev
        upper = self.mean + self.z_threshold * std_dev

        severity = "info"
        if is_anomaly:
            if z_score > self.z_threshold * 2:
                severity = "critical"
            elif z_score > self.z_threshold * 1.5:
                severity = "warning"

        return AnomalyResult(
            is_anomaly=is_anomaly, value=value,
            expected_range=(round(lower, 2), round(upper, 2)),
            z_score=round(z_score, 4),
            timestamp=timestamp, metric_name=metric_name,
            severity=severity,
        )
```

Tip

When using Claude Code to design anomaly detection, provide sample data characteristics: typical value ranges, seasonal patterns, expected event frequency, and examples of both normal and anomalous behavior. Without this context, Claude Code will design a perfectly generic detector that flags everything or nothing. Both are technically anomaly detectors; neither is useful.

22.5 Event Schema Validation and Evolution

In streaming systems, schema management is critical because producers and consumers are decoupled. Claude Code can assist in designing schema evolution strategies and generating validation logic.

Code Example

```
from enum import Enum


class Compatibility(Enum):
    BACKWARD = "backward"
    FORWARD = "forward"
    FULL = "full"
    NONE = "none"


class SchemaEvolutionValidator:
    """Validate schema changes for streaming compatibility."""

    def __init__(self, compatibility: Compatibility = Compatibility.
    FULL):
        self.compatibility = compatibility

    def validate_evolution(self, old_schema: dict, new_schema: dict)
    -> dict:
        """Validate schema change for compatibility."""
        issues = []
        old_fields = {f["name"]: f for f in old_schema.get("fields",
    [])}
        new_fields = {f["name"]: f for f in new_schema.get("fields",
    [])}
```

```

        removed = set(old_fields) - set(new_fields)
        if removed and self.compatibility in (
            Compatibility.BACKWARD, Compatibility.FULL
        ):
            issues.append({
                "type": "removed_field", "severity": "error",
                "fields": list(removed),
                "message": f"Removing fields {removed} breaks "
                           f"{self.compatibility.value} compatibility.",
            })

        added = set(new_fields) - set(old_fields)
        for field_name in added:
            field_def = new_fields[field_name]
            has_default = "default" in field_def
            is_nullable = isinstance(field_def.get("type"), list) and "null" in field_def["type"]
            if not has_default and not is_nullable:
                if self.compatibility in (Compatibility.FORWARD, Compatibility.FULL):
                    issues.append({
                        "type": "required_field_added", "severity": "error",
                        "field": field_name,
                        "message": f"Adding required field '{field_name}' without "
                                   f"a default breaks compatibility.",
                    })

        is_compatible = not any(i["severity"] == "error" for i in issues)
        return {
            "compatible": is_compatible,
            "compatibility_mode": self.compatibility.value,
            "issues": issues,
            "added_fields": list(added),
            "removed_fields": list(removed),
        }
```

22.6 Building Real-Time Data Enrichment Pipelines

Data enrichment adds context and value to raw events by joining them with reference data, computing derived fields, or augmenting with external information. Common enrichment types include:

- **Lookup enrichment**: Join events with reference data (customer profiles, product catalogs) using a cache for performance.
- **Computed enrichment**: Derive new fields from existing ones (e.g., categorizing amounts into buckets).
- **Geographic enrichment**: Resolve IP addresses to locations using a GeoIP database.

When using Claude Code to generate enrichment logic, specify the cache TTL, the fallback behavior when lookups fail, and how to handle null values in key fields. Claude Code consistently generates defensive code for these cases when prompted explicitly.

22.7 Real-Time Data Quality Monitoring

Data quality monitoring in streaming systems requires continuous validation as data flows through the pipeline. Key quality dimensions to monitor in real time:

- **Completeness**: NULL rates per field per time window.
- **Type conformance**: Fields matching expected types.
- **Range validity**: Numeric values within acceptable bounds.
- **Uniqueness**: Duplicate detection within a sliding window.
- **Timeliness**: Delay between event time and processing time.

Implement a windowed quality monitor that accumulates violations per window, computes rates, and generates alerts when thresholds are breached. Use Claude Code to generate the quality rule configurations—describe your schema and business rules, and it will produce comprehensive validation logic including edge cases you might miss.

Note

Track quality scores over time to detect gradual degradation. A sudden drop indicates a source system change (or someone deploying on Friday). A gradual decline suggests data drift—the data equivalent of entropy slowly winning, as it always does.

22.8 Full Example: Fraud Detection Pipeline

This section presents a fraud detection pipeline combining rule-based detection, statistical anomaly detection, and pattern matching.

Code Example

```
"""Real-time fraud detection pipeline."""
import time
from datetime import datetime, timedelta
from dataclasses import dataclass
from collections import defaultdict
from enum import Enum


class FraudSignal(Enum):
    VELOCITY = "velocity"
    AMOUNT = "amount_anomaly"
    GEOGRAPHY = "geo_anomaly"


@dataclass
class FraudScore:
    transaction_id: str
    customer_id: str
    score: float
    signals: list[dict]
    decision: str  # "allow", "review", "block"
    timestamp: datetime


class FraudDetectionPipeline:
    """Real-time fraud detection with multiple signal types."""

```

```
    def __init__(self, config: dict):
        self.config = config
        self.thresholds = config.get("thresholds", {
            "block": 0.8, "review": 0.5,
        })
        self.customer_history: dict[str, list[dict]] = defaultdict(
    list)
        self.amount_detectors: dict[str, StreamingAnomalyDetector] =
    {}
        self.velocity_window = config.get("velocity_window_minutes",
    60)

    def process_transaction(self, transaction: dict) -> FraudScore:
        """Process a transaction and return fraud assessment."""
        start_time = time.time()
        customer_id = transaction["customer_id"]
        amount = float(transaction["amount"])
        timestamp = datetime.fromisoformat(transaction["timestamp"])

        signals = []

        # Velocity check
        cutoff = timestamp - timedelta(minutes=self.velocity_window)
        recent = [t for t in self.customer_history[customer_id]
                  if t["timestamp"] > cutoff]
        velocity_limit = self.config.get("velocity_limit", 10)
        if len(recent) >= velocity_limit:
            signals.append({
                "signal": FraudSignal.VELOCITY.value,
                "weight": 0.3,
                "confidence": min(len(recent) / velocity_limit, 2.0)
    / 2.0,
            })

        # Amount anomaly
        if customer_id not in self.amount_detectors:
            self.amount_detectors[customer_id] =
    StreamingAnomalyDetector(
                window_size=100, z_threshold=2.5, min_samples=5,
            )
        result = self.amount_detectors[customer_id].update(
            amount, timestamp, "amount"
        )
        if result.is_anomaly:
```

```
            signals.append({
                "signal": FraudSignal.AMOUNT.value,
                "weight": 0.25,
                "confidence": min(result.z_score / 5.0, 1.0),
            })

        # Composite score
        if signals:
            weighted_sum = sum(s["weight"] * s["confidence"] for s in signals)
            max_possible = sum(s["weight"] for s in signals)
            score = (weighted_sum / max_possible) + min(len(signals) * 0.1, 0.3)
            score = min(score, 1.0)
        else:
            score = 0.0

        if score >= self.thresholds["block"]:
            decision = "block"
        elif score >= self.thresholds["review"]:
            decision = "review"
        else:
            decision = "allow"

        # Update state
        self.customer_history[customer_id].append({
            **transaction, "timestamp": timestamp,
        })
        self.customer_history[customer_id] = [
            t for t in self.customer_history[customer_id]
            if t["timestamp"] > timestamp - timedelta(hours=24)
        ]

        return FraudScore(
            transaction_id=transaction["transaction_id"],
            customer_id=customer_id, score=round(score, 4),
            signals=signals, decision=decision, timestamp=timestamp,
        )
```

22.9 Latency and Cost Considerations for LLM-in-the-Loop Streaming

A typical Claude Code API call takes 500ms–5s. For streaming pipelines, this has significant implications:

- **Synchronous enrichment**: Acceptable only for near-real-time pipelines with second-scale latency budgets.
- **Asynchronous enrichment**: Fire API calls asynchronously and join results back later. Maintains low primary-path latency but adds complexity.
- **Batch micro-enrichment**: Collect 10–50 events per Claude Code call to amortize per-call overhead, at the cost of batching latency.

Warning

Never place a synchronous Claude Code API call in the critical path of a high-throughput streaming pipeline. At 1,000 events/second, even 1-second latency requires 1,000 concurrent calls—neither cost-effective nor reliable. Your CFO will find you. Your CFO will have questions.

When Not to Use LLMs in Streaming

1. **Sub-100ms latency requirements**: LLM API latency makes this impossible.
2. **Throughput above 10,000 events/second**: Cost and concurrency become prohibitive.
3. **Deterministic processing requirements**: LLMs produce non-deterministic outputs.
4. **Simple rule-based logic**: An LLM adds unnecessary complexity and cost.

The most effective pattern is using Claude Code to *generate* stream processing logic that then runs as deterministic code at full streaming speed. Reserve runtime LLM calls for low-volume, high-value enrichment.

Tip

For cost-sensitive streaming, use CLAUDE CODE Haiku for high-volume classification and reserve Sonnet or Opus for complex reasoning. The cost difference can be an order of magnitude. Think of it as economy vs. first class: both get you there, but one lets you keep your house.

22.10 Exercises

Exercises

1. **Stream Processing Generation**: Use CLAUDE CODE to generate a ksqlDB application that sessionizes web clickstream events with a 30-minute gap, computes page views per session, identifies sessions with 50+ views as potential bots, and computes real-time conversion rates.

2. **Anomaly Detection Tuning**: Process a synthetic dataset through `StreamingAnomalyDetector` with different `z_threshold` (2.0, 2.5, 3.0, 3.5) and `window_size` (100, 500, 1000) values. Report precision, recall, and F1 for each configuration.

3. **Schema Evolution Challenge**: Design an Avro schema for payment events. Simulate three evolution rounds: (a) add optional `currency`, (b) rename `amount` to `payment_amount`, (c) add required `idempotency_key`. Analyze compatibility for each.

4. **Fraud Detection Enhancement**: Extend the fraud pipeline with a "card testing" signal (small transactions followed by a large one) and a "merchant pattern" signal (unfamiliar merchants during unusual hours).

5. **End-to-End Pipeline**: Build a complete streaming pipeline that generates synthetic orders, monitors quality, enriches with customer data, detects anomalies, and writes results to an output topic using Docker Compose with Kafka.

23

Multi-Modal Data Engineering

"Multi-modal data engineering: because regular data wasn't complicated enough, now we're feeding Claude Code pictures of whiteboards."

Data engineering has long been dominated by structured rows and columns. But enterprises generate vast amounts of unstructured, multi-modal data: images, audio recordings, PDFs, video, and scanned documents. With native vision capabilities and sophisticated reasoning, Claude Code enables data engineers to build multi-modal ETL pipelines that treat these sources as first-class data. This chapter provides architectural patterns, production code, and operational guidance for building these pipelines at scale. Finally, a use for all those photos of whiteboard diagrams you took with the sincere intention of "digitizing later."

23.1 Multi-Modal Data in the Modern Data Stack

Architectural Patterns for Multi-Modal Pipelines

Three primary patterns exist for integrating multi-modal data:

Pattern 1: Extract-then-load. Files are processed by Claude Code before entering the warehouse. Only extracted structured data lands in the warehouse; raw files remain in object storage. This is the simplest and most common pattern.

Pattern 2: Load-then-extract. Raw files are loaded into a lakehouse first (e.g., as external tables pointing to S3), and extraction happens as a transformation step.

Pattern 3: Hybrid streaming. Multi-modal data arrives via a streaming platform, is processed in near-real-time by Claude Code, and extracted data is written to both streaming and batch sinks.

Cost Considerations

- **Image tokens**: A 1024×1024 image typically uses around 1,600 tokens. Downscaling reduces costs dramatically.
- **Audio preprocessing**: Claude Code does not natively process audio—transcribe first (Whisper, AWS Transcribe), then analyze the transcript.
- **Caching**: Prompt caching can reduce costs by up to 90% when processing many items with the same instructions.

Tip

Always calculate cost per unit (per image, per document, per audio minute) before scaling to production. Run a sample of 100–500 items and measure actual token consumption. Many a promising multi-modal project has died on the spreadsheet of "wait, it costs HOW much per image?"

23.2 Image Processing Pipelines with Claude Code Vision

Claude Code's vision capabilities allow direct image analysis—reading text, describing content, and extracting structured data without custom computer vision models.

Sending Images to the API

Code Example

```python
import anthropic
import base64
from pathlib import Path


def encode_image(image_path: str) -> tuple[str, str]:
    """Encode an image file to base64 with media type detection."""
```

```
    path = Path(image_path)
    media_types = {
        ".jpg": "image/jpeg", ".jpeg": "image/jpeg",
        ".png": "image/png", ".gif": "image/gif",
        ".webp": "image/webp",
    }
    media_type = media_types.get(path.suffix.lower(), "image/jpeg")
    with open(image_path, "rb") as f:
        image_data = base64.standard_b64encode(f.read()).decode("utf
    -8")
    return image_data, media_type


def extract_from_image(
    client: anthropic.Anthropic,
    image_path: str,
    extraction_prompt: str,
    model: str = "claude-sonnet-4-20250514",
) -> str:
    """Send an image to Claude Code and extract structured data."""
    image_data, media_type = encode_image(image_path)
    message = client.messages.create(
        model=model, max_tokens=4096,
        messages=[{
            "role": "user",
            "content": [
                {"type": "image", "source": {
                    "type": "base64",
                    "media_type": media_type,
                    "data": image_data,
                }},
                {"type": "text", "text": extraction_prompt},
            ],
        }],
    )
    return message.content[0].text
```

Image Preprocessing for Cost Optimization

Code Example

```python
from PIL import Image
import io
import base64


def preprocess_image(
    image_bytes: bytes,
    max_dimension: int = 1024,
    quality: int = 85,
    output_format: str = "JPEG",
) -> tuple[str, str]:
    """Resize and compress an image for optimal API usage."""
    img = Image.open(io.BytesIO(image_bytes))
    if img.mode == "RGBA" and output_format == "JPEG":
        background = Image.new("RGB", img.size, (255, 255, 255))
        background.paste(img, mask=img.split()[3])
        img = background

    width, height = img.size
    if max(width, height) > max_dimension:
        ratio = max_dimension / max(width, height)
        img = img.resize(
            (int(width * ratio), int(height * ratio)),
            Image.LANCZOS,
        )

    buffer = io.BytesIO()
    img.save(buffer, format=output_format, quality=quality)
    encoded = base64.standard_b64encode(buffer.getvalue()).decode("utf-8")
    format_to_media = {"JPEG": "image/jpeg", "PNG": "image/png", "WEBP": "image/webp"}
    return encoded, format_to_media.get(output_format, "image/jpeg")
```

Note

For most extraction tasks (receipts, product attributes, charts), resizing to 1024 pixels on the longest side reduces token consumption by 60–75% with excellent results. Use higher resolutions only for fine-grained details.

23.3 Audio Transcription and Analysis Workflows

The standard pattern is a two-stage pipeline: transcribe audio using a speech-to-text service, then send the transcript to Claude Code for structured extraction.

Code Example

```
import anthropic
import whisper
import json
from dataclasses import dataclass
from typing import Optional


@dataclass
class AudioAnalysis:
    file_path: str
    duration_seconds: float
    transcript: str
    summary: str
    topics: list[str]
    sentiment: str
    action_items: list[str]


AUDIO_ANALYSIS_PROMPT = """Analyze this transcript and return JSON:
{
  "summary": "2-3 sentence summary",
  "topics": ["main", "topics"],
  "sentiment": "positive/negative/neutral/mixed",
  "action_items": ["list of action items mentioned"],
  "language": "detected language"
}

TRANSCRIPT:
```

```
{transcript}

Return ONLY valid JSON."""


class AudioAnalysisPipeline:
    def __init__(self, whisper_model: str = "base"):
        self.whisper_model = whisper.load_model(whisper_model)
        self.claude = anthropic.Anthropic()

    def transcribe(self, audio_path: str) -> dict:
        result = self.whisper_model.transcribe(audio_path)
        return {"text": result["text"], "language": result.get("
    language", "unknown")}

    def analyze_transcript(self, transcript: str) -> dict:
        prompt = AUDIO_ANALYSIS_PROMPT.format(transcript=transcript)
        message = self.claude.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=2048,
            messages=[{"role": "user", "content": prompt}],
        )
        return json.loads(message.content[0].text)
```

Handling Long Audio Files

For transcripts exceeding the context window, split into overlapping chunks, process each independently, then merge:

Code Example

```
def chunk_transcript(
    transcript: str,
    max_chars: int = 150000,
    overlap_chars: int = 5000,
) -> list[str]:
    """Split a long transcript into overlapping chunks."""
    if len(transcript) <= max_chars:
        return [transcript]
    chunks = []
    start = 0
```

```
    while start < len(transcript):
        end = start + max_chars
        if end < len(transcript):
            boundary = transcript.rfind(". ", start + max_chars -
1000, end)
            if boundary > start:
                end = boundary + 1
        chunks.append(transcript[start:end])
        start = end - overlap_chars
    return chunks
```

23.4 Video Metadata Extraction Pipelines

A single hour of 1080p video contains 108,000 frames. The key to practical pipelines is intelligent ***frame sampling***.

Table 23.1: Frame sampling strategies

Strategy	Frames/hr	Best For
Fixed interval (1 fps)	3,600	General indexing
Fixed interval (1/10s)	360	Lectures, presentations
Scene change detection	50–500	Dynamic content
Motion-triggered	Varies	Surveillance

Warning

Video processing can generate enormous API costs. A 1-hour video at 1 fps produces 3,600 calls and ~5.76M input tokens. That is not a typo. That is a "we need to talk about your API bill" meeting. Always start with coarse sampling (one frame every 30–60 seconds) and refine only when necessary.

23.5 Video Frame Analysis with Claude Code

Once frames are extracted, each is analyzed individually. The following pipeline produces a structured timeline:

Code Example

```python
import anthropic
import json
import subprocess
from dataclasses import dataclass
from pathlib import Path


@dataclass
class FrameAnalysis:
    frame_path: str
    timestamp_seconds: float
    description: str
    objects: list[str]
    activity: str


FRAME_PROMPT = """Analyze this video frame and return JSON:
{"description": "what is shown", "objects": ["visible", "objects"],
 "text_visible": "any on-screen text or null",
 "activity": "what action is occurring"}
Return ONLY JSON."""


def extract_frames_interval(
    video_path: str, output_dir: str,
    interval_seconds: float = 10.0,
) -> list[str]:
    """Extract frames at fixed intervals using FFmpeg."""
    Path(output_dir).mkdir(parents=True, exist_ok=True)
    cmd = [
        "ffmpeg", "-i", video_path,
        "-vf", f"fps=1/{interval_seconds},scale=1024:-1",
        "-q:v", "2",
        f"{output_dir}/frame_%05d.jpg",
        "-y", "-loglevel", "quiet",
    ]
    subprocess.run(cmd, check=True)
    return sorted(str(f) for f in Path(output_dir).glob("frame_*.jpg"
    ))


```

```
class VideoAnalysisPipeline:
    def __init__(self):
        self.claude = anthropic.Anthropic()

    def analyze_frame(self, frame_path: str, timestamp: float) ->
    FrameAnalysis:
        image_data, media_type = encode_image(frame_path)
        message = self.claude.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=512,
            messages=[{"role": "user", "content": [
                {"type": "image", "source": {
                    "type": "base64", "media_type": media_type,
                    "data": image_data,
                }},
                {"type": "text", "text": FRAME_PROMPT},
            ]}],
        )
        parsed = json.loads(message.content[0].text)
        return FrameAnalysis(
            frame_path=frame_path, timestamp_seconds=timestamp,
            description=parsed["description"],
            objects=parsed["objects"], activity=parsed["activity"],
        )
```

23.6 Building Multi-Modal ETL Pipelines

A production multi-modal ETL pipeline uses the ***processor registry*** pattern, where each file type has a registered handler:

Code Example

```
import anthropic
import json
import logging
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
```

```python
from typing import Optional

logger = logging.getLogger(__name__)


class ProcessingStatus(Enum):
    PENDING = "pending"
    COMPLETED = "completed"
    FAILED = "failed"
    SKIPPED = "skipped"


@dataclass
class ProcessingResult:
    file_path: str
    status: ProcessingStatus
    extracted_data: Optional[dict] = None
    error_message: Optional[str] = None
    processing_time_ms: float = 0.0


class BaseProcessor(ABC):
    def __init__(self, client: anthropic.Anthropic):
        self.client = client

    @abstractmethod
    def supported_extensions(self) -> set[str]: ...

    @abstractmethod
    def process(self, file_path: str) -> dict: ...

    def can_handle(self, file_path: str) -> bool:
        return Path(file_path).suffix.lower() in self.
    supported_extensions()


class ImageProcessor(BaseProcessor):
    def supported_extensions(self) -> set[str]:
        return {".jpg", ".jpeg", ".png", ".gif", ".webp"}

    def process(self, file_path: str) -> dict:
        preprocessed, prep_type = preprocess_image(
            Path(file_path).read_bytes()
        )
```

```
        message = self.client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=2048,
            messages=[{"role": "user", "content": [
                {"type": "image", "source": {
                    "type": "base64", "media_type": prep_type,
                    "data": preprocessed,
                }},
                {"type": "text", "text": (
                    "Extract all structured data from this image. "
                    "Return JSON with: description, text_content, "
                    "objects, categories, and any tabular data."
                )},
            ]}],
        )
        return json.loads(message.content[0].text)


class MultiModalETLPipeline:
    """Orchestrates multi-modal file processing."""

    def __init__(self):
        self.client = anthropic.Anthropic()
        self.processors: list[BaseProcessor] = [
            ImageProcessor(self.client),
        ]

    def process_file(self, file_path: str) -> ProcessingResult:
        import time
        processor = next(
            (p for p in self.processors if p.can_handle(file_path)),
            None,
        )
        if processor is None:
            return ProcessingResult(
                file_path=file_path, status=ProcessingStatus.SKIPPED,
                error_message="No processor for file type",
            )
        start = time.monotonic()
        try:
            extracted = processor.process(file_path)
            return ProcessingResult(
                file_path=file_path, status=ProcessingStatus.
    COMPLETED,
                extracted_data=extracted,
```

```
                processing_time_ms=(time.monotonic() - start) * 1000,
            )
        except Exception as e:
            logger.error(f"Failed to process {file_path}: {e}")
            return ProcessingResult(
                file_path=file_path, status=ProcessingStatus.FAILED,
                error_message=str(e),
                processing_time_ms=(time.monotonic() - start) * 1000,
            )
```

Tip

The processor registry pattern makes it easy to add new file types without modifying existing code. Simply implement a new `BaseProcessor` subclass and register it. Open/closed principle: open for extension, closed for "who touched the image processor and broke PDFs."

23.7 Document Digitization with Claude Code Vision

Traditional OCR systems produce raw text but struggle with layout understanding and contextual interpretation. They see pixels; Claude Code's vision understands document *semantics*. It's the difference between a scanner and a human who can read—except this human processes invoices at 3 AM without complaining.

Invoice Processing Example

Code Example

```python
import anthropic
import json
from pydantic import BaseModel
from typing import Optional


class LineItem(BaseModel):
    description: str
    quantity: float
```

```
    unit_price: float
    total: float


class InvoiceData(BaseModel):
    invoice_number: str
    invoice_date: str
    vendor_name: str
    line_items: list[LineItem]
    subtotal: float
    tax_amount: Optional[float] = None
    total_amount: float
    currency: str = "USD"


INVOICE_PROMPT = """Extract all data from this invoice image.
Return a JSON object with: invoice_number, invoice_date (YYYY-MM-DD),
vendor_name, line_items (description, quantity, unit_price, total),
subtotal, tax_amount, total_amount, currency.
All monetary values as numbers. Return ONLY JSON."""


def extract_invoice(
    client: anthropic.Anthropic, image_path: str,
) -> InvoiceData:
    image_data, media_type = encode_image(image_path)
    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=4096,
        messages=[{"role": "user", "content": [
            {"type": "image", "source": {
                "type": "base64", "media_type": media_type,
                "data": image_data,
            }},
            {"type": "text", "text": INVOICE_PROMPT},
        ]}],
    )
    return InvoiceData(**json.loads(message.content[0].text))
```

Warning

When processing financial documents, always implement a validation layer that cross-checks extracted values—verify line item totals, check that items sum to the subtotal, and that subtotal plus tax equals the total. Flag discrepancies for human review. "The AI said the invoice was $47 million" is not a sentence your accounting department wants to hear.

Validation

Code Example

```python
from dataclasses import dataclass


@dataclass
class ValidationResult:
    is_valid: bool
    errors: list[str]
    warnings: list[str]


def validate_invoice(invoice: InvoiceData) -> ValidationResult:
    errors, warnings = [], []
    for i, item in enumerate(invoice.line_items):
        expected = round(item.quantity * item.unit_price, 2)
        if abs(item.total - expected) > 0.01:
            errors.append(f"Line {i+1}: total != qty * price")

    items_sum = round(sum(it.total for it in invoice.line_items), 2)
    if abs(invoice.subtotal - items_sum) > 0.01:
        errors.append(f"Subtotal {invoice.subtotal} != line items sum
     {items_sum}")

    if not invoice.invoice_number:
        errors.append("Missing invoice number")
    if not invoice.line_items:
        errors.append("No line items extracted")

    return ValidationResult(is_valid=len(errors) == 0, errors=errors,
     warnings=warnings)
```

23.8 PDF Processing Pipeline

PDFs require special handling: text-based pages can use direct text extraction, while scanned pages must be rendered as images and processed through Claude Code's vision:

Code Example

```
import base64

class PDFProcessor(BaseProcessor):
    """Processor for PDF documents."""

    def supported_extensions(self) -> set[str]:
        return {".pdf"}

    def process(self, file_path: str) -> dict:
        import fitz  # PyMuPDF

        doc = fitz.open(file_path)
        pages_data = []
        for page_num in range(len(doc)):
            page = doc[page_num]
            text = page.get_text()
            if len(text.strip()) < 50:
                # Scanned page: render as image
                pix = page.get_pixmap(matrix=fitz.Matrix(2, 2))
                img_b64 = base64.standard_b64encode(
                    pix.tobytes("jpeg")
                ).decode("utf-8")
                message = self.client.messages.create(
                    model="claude-sonnet-4-20250514", max_tokens
    =2048,
                    messages=[{"role": "user", "content": [
                        {"type": "image", "source": {
                            "type": "base64", "media_type": "image/
    jpeg",
                            "data": img_b64,
```

```
                        }},
                        {"type": "text", "text": (
                            "Extract text, tables, and form fields. "
                            "Return JSON with: text, tables,
    form_fields."
                        )},
                    ]}],
                )
                page_data = json.loads(message.content[0].text)
            else:
                page_data = {"text": text, "tables": [], "form_fields
    ": {}}
            pages_data.append({"page_number": page_num + 1, "content"
    : page_data})
        doc.close()
        return {"total_pages": len(pages_data), "pages": pages_data}
```

23.9 Document Classification System

For large-scale digitization, classify documents by type first, then apply type-specific extraction prompts:

Code Example

```
import anthropic
import json
from dataclasses import dataclass
from enum import Enum


class DocumentType(Enum):
    INVOICE = "invoice"
    RECEIPT = "receipt"
    CONTRACT = "contract"
    FORM = "form"
    UNKNOWN = "unknown"


@dataclass
```

```python
class ClassificationResult:
    document_type: DocumentType
    confidence: float
    reasoning: str


CLASSIFICATION_PROMPT = """Classify this document into one of:
invoice, receipt, contract, form, unknown.
Return JSON: {"document_type": "...", "confidence": 0.0-1.0,
"reasoning": "brief explanation"}
Return ONLY JSON."""


class DocumentClassifier:
    def __init__(self, client: anthropic.Anthropic):
        self.client = client

    def classify(self, image_data: str, media_type: str) ->
    ClassificationResult:
        message = self.client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=256,
            messages=[{"role": "user", "content": [
                {"type": "image", "source": {
                    "type": "base64", "media_type": media_type,
                    "data": image_data,
                }},
                {"type": "text", "text": CLASSIFICATION_PROMPT},
            ]}],
        )
        parsed = json.loads(message.content[0].text)
        return ClassificationResult(
            document_type=DocumentType(parsed["document_type"]),
            confidence=parsed["confidence"],
            reasoning=parsed["reasoning"],
        )
```

Tip

For high-volume digitization (10,000+ documents/day), use a two-pass strategy: classify with Haiku (faster, cheaper), then extract with Sonnet (more accurate). This reduces costs by 40–60%. Think of Haiku as the bouncer deciding who gets in, and Sonnet as the bartender taking the actual order.

23.10 Storage and Indexing Strategies

Separate raw assets from extracted metadata:

- **Object storage**: Raw media files, organized by date and type, with lifecycle policies for cold storage.
- **Relational database**: Extracted metadata, classification results, and processing state.
- **Vector database**: Embeddings of extracted text for semantic search.
- **Search index**: Full-text search across extracted content.

Code Example

```
-- Core asset table
CREATE TABLE media_assets (
    asset_id        UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    s3_key          TEXT NOT NULL UNIQUE,
    asset_type      TEXT NOT NULL,
    file_size_bytes BIGINT NOT NULL,
    content_type    TEXT NOT NULL,
    uploaded_at     TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    status          TEXT DEFAULT 'pending'
);

-- Extracted text with full-text search
CREATE TABLE extracted_text (
    id          SERIAL PRIMARY KEY,
    asset_id    UUID REFERENCES media_assets(asset_id),
    content     TEXT NOT NULL,
    content_tsv TSVECTOR GENERATED ALWAYS AS (
                    to_tsvector('english', content)
                ) STORED
);
CREATE INDEX idx_extracted_text_tsv
    ON extracted_text USING GIN (content_tsv);

-- Extracted structured fields (JSONB for flexibility)
CREATE TABLE extracted_fields (
    id          SERIAL PRIMARY KEY,
    asset_id    UUID REFERENCES media_assets(asset_id),
```

```
    doc_type    TEXT NOT NULL,
    fields      JSONB NOT NULL,
    confidence  REAL DEFAULT 1.0
);
CREATE INDEX idx_extracted_fields_jsonb
    ON extracted_fields USING GIN (fields jsonb_path_ops);
```

Note

For semantic search, generate embeddings using a dedicated embedding model (Voyage AI, OpenAI's `text-embedding-3-large`), not Claude Code itself. Using Claude Code for embeddings is like hiring a brain surgeon to take your temperature. Embedding models are orders of magnitude cheaper and faster for indexing.

23.11 Advanced Techniques

Multi-Image Analysis

Some tasks require analyzing multiple related images together. Claude Code supports multiple images in a single API call:

Code Example

```python
def analyze_multi_page_document(
    client: anthropic.Anthropic,
    image_paths: list[str],
    extraction_prompt: str,
) -> dict:
    """Send multiple page images for unified analysis."""
    content = []
    for i, path in enumerate(image_paths):
        data, media = encode_image(path)
        content.append({"type": "image", "source": {
            "type": "base64", "media_type": media, "data": data,
        }})
        content.append({"type": "text", "text": f"(Page {i+1} of {len
    (image_paths)})"})
```

```
    content.append({"type": "text", "text": extraction_prompt})

    message = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=8192,
        messages=[{"role": "user", "content": content}],
    )
    return json.loads(message.content[0].text)
```

Error Handling and Retry Strategies

Code Example

```
import time
import random
from functools import wraps

def retry_with_backoff(
    max_retries: int = 3, base_delay: float = 1.0,
    max_delay: float = 60.0,
):
    """Decorator for retrying API calls with exponential backoff."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries + 1):
                try:
                    return func(*args, **kwargs)
                except (anthropic.RateLimitError,
                        anthropic.InternalServerError) as e:
                    if attempt == max_retries:
                        raise
                    delay = min(
                        base_delay * (2 ** attempt) + random.uniform
    (0, 1),
                        max_delay,
                    )
                    time.sleep(delay)
                except anthropic.BadRequestError:
                    raise  # Non-retryable
        return wrapper
```

```
    return decorator
```

23.12 Exercises

1. **Receipt Processing Pipeline.** Build a pipeline that processes receipt photographs, extracts store name, date, items with prices, tax, and total. Write results to SQLite. Validate that item prices plus tax equal total.

2. **Audio Meeting Summarizer.** Build a pipeline using Whisper for transcription and Claude Code for extraction of summaries, action items, key decisions, and topics. Handle files over 2 hours with chunking.

3. **Multi-Modal ETL with Error Handling.** Extend `MultiModalETLPipeline` to support images and PDFs with: exponential backoff retry, a dead-letter queue for failures, Prometheus metrics, and idempotent processing.

4. **Cost Calculator.** Build a tool that estimates the cost of processing a directory of mixed media through the pipeline, based on file types, image dimensions, and audio durations.

Tip

For exercises involving image or document processing, find free sample datasets on Kaggle—search for "receipt OCR dataset" or "invoice dataset." For audio, use freely available podcast episodes. Do not use your company's actual financial documents for testing. Your compliance team will not find it as educational as you do. Always respect licensing terms.

23.13 Monitoring and Observability

Production multi-modal pipelines require comprehensive monitoring. Track these key metrics:

Data Lifecycle Management

Multi-modal data grows quickly. Implement lifecycle policies from day one:

Table 23.2: Key metrics for multi-modal pipeline monitoring

Metric	Type	Alert Threshold	Purpose
Processing latency (p99)	Histogram	>60s per item	Performance regression
API error rate	Counter	>5%	Claude Code API issues
Validation failure rate	Gauge	>20%	Extraction quality
Token usage per item	Histogram	>2× baseline	Cost anomaly
Queue depth	Gauge	>10,000 items	Throughput bottleneck

- **Object storage**: Transition processed files to infrequent access after 30 days, Glacier after 90 days, deep archive after 365 days.
- **Temporary frames**: Delete extracted video frames after 7 days.
- **Database**: Archive old extractions annually. Partition tables by date for efficient pruning.

Prompt Caching for Batch Processing

When processing many documents of the same type, the system prompt and extraction instructions remain constant. Use prompt caching to avoid reprocessing these tokens:

Code Example

```python
def process_batch_with_caching(
    client: anthropic.Anthropic,
    image_data_list: list[tuple[str, str]],
    system_prompt: str,
    extraction_prompt: str,
) -> list[dict]:
    results = []
    for image_data, media_type in image_data_list:
        message = client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=2048,
            system=[{
                "type": "text", "text": system_prompt,
                "cache_control": {"type": "ephemeral"},
            }],
            messages=[{"role": "user", "content": [
                {"type": "image", "source": {
```

```
                    "type": "base64", "media_type": media_type,
                    "data": image_data,
                }},
                {"type": "text", "text": extraction_prompt},
            ]}],
        )
        results.append(json.loads(message.content[0].text))
    return results
```

Part VIII

The Future

24

Building a Claude Code-First Data Platform

"Every company will eventually become a Claude Code wrapper. It's wrappers all the way down."

Tip

Building a Claude Code-first data platform is not about replacing your infrastructure overnight. It is about systematically identifying where AI augmentation delivers the highest ROI and incrementally transforming your architecture, team, and processes. Rome wasn't built in a day, and neither is a data platform—but with Claude Code, you can at least get the architecture diagram done before lunch.

This chapter provides a blueprint for building and operating a Claude Code-augmented data platform, including honest assessments of where it should *not* be used, how to measure ROI, and how to manage the human side of this transformation. Whether you are a solo data engineer at a startup or leading a platform team at a Fortune 500, the patterns here scale to your context.

24.1 Architecture Patterns for AI-Augmented Data Platforms

A modern Claude Code-augmented platform consists of five layers, each with defined AI integration points:

Ingestion Layer Connectors, CDC pipelines, and API extractors. Claude Code assists with connector configuration, schema mapping, and error handling logic.

Processing Layer Spark, Flink, dbt, and custom Python. Claude Code generates transformation code, optimizes queries, and reviews data quality rules.

Storage Layer Data lakes, warehouses, and lakehouses. Claude Code advises on partitioning strategies, manages schema evolution, and generates documentation.

Serving Layer APIs, dashboards, and feature stores. Claude Code powers natural language query interfaces and generates API boilerplate.

Orchestration Layer Airflow, Dagster, and Prefect. Claude Code monitors pipeline health, diagnoses failures, and automates incident response.

The key architectural decision is *where* Claude Code integrates. Three patterns have emerged as the most effective.

The Sidecar Pattern

The ***sidecar pattern*** attaches a Claude Code agent to each critical data service. The sidecar monitors the service's events, provides advisory analysis, and optionally takes remediation actions within a constrained scope. This pattern is effective because each sidecar has deep context about its specific service without needing global visibility.

Code Example

Claude Code sidecar pattern

```
import anthropic
from collections import deque

class ClaudeSidecar:
```

```
    """A sidecar agent that monitors a data service."""

    def __init__(self, service_name: str,
                 model: str = "claude-sonnet-4-20250514",
                 window_size: int = 100):
        self.service_name = service_name
        self.client = anthropic.Anthropic()
        self.model = model
        self.context_window = deque(maxlen=window_size)
        self.alert_history: list[dict] = []

    async def observe(self, event: dict) -> None:
        """Record an event from the monitored service."""
        self.context_window.append(event)

    async def analyze(self, question: str) -> str:
        """Ask the sidecar to analyze recent events."""
        context = "\n".join(
            f"[{e['timestamp']}] {e['type']}: "
            f"{e['message']}"
            for e in list(self.context_window)[-50:]
        )
        response = self.client.messages.create(
            model=self.model, max_tokens=2048,
            system=(
                f"You are a sidecar agent for "
                f"{self.service_name}. Analyze events, "
                f"detect anomalies, recommend actions. "
                f"Be specific and cite event timestamps."),
            messages=[{
                "role": "user",
                "content": (
                    f"Recent events:\n{context}\n\n"
                    f"Question: {question}")
            }],
        )
        return response.content[0].text

    async def should_alert(self) -> dict | None:
        """Proactively check if conditions warrant alert."""
        if len(self.context_window) < 10:
            return None
        recent_errors = sum(
            1 for e in list(self.context_window)[-20:]
```

```
            if e.get("level") == "ERROR")
        if recent_errors < 3:
            return None
        analysis = await self.analyze(
            "Are recent errors concerning? "
            "Should the on-call engineer be alerted?")
        return {"service": self.service_name,
                "error_count": recent_errors,
                "analysis": analysis}
```

The Gateway Pattern

The ***gateway pattern*** routes all Claude Code interactions through a centralized API gateway. This gateway handles authentication, rate limiting, prompt management, cost tracking, and audit logging. It provides a single control plane for all AI usage across the platform.

Note

The gateway pattern is especially valuable in regulated industries where every AI interaction must be logged, every prompt must be versioned, and usage must be attributed to specific users and cost centers. It also enables A/B testing of different prompts and models without changing application code.

Code Example

Centralized AI gateway

```
from dataclasses import dataclass, field
from datetime import datetime, timezone

@dataclass
class GatewayRequest:
    user_id: str
    team: str
    use_case: str  # "code_review", "incident", etc.
    prompt: str
    model_override: str | None = None

@dataclass
```

```python
class GatewayConfig:
    default_model: str = "claude-sonnet-4-20250514"
    model_overrides: dict[str, str] = field(
        default_factory=lambda: {
            "classification": "claude-haiku-4-20250514",
            "architecture": "claude-opus-4-20250514",
        })
    rate_limits: dict[str, int] = field(
        default_factory=lambda: {
            "default": 100,      # per hour
            "batch": 1000,
            "interactive": 50,
        })

class AIGateway:
    """Centralized gateway for all Claude Code calls."""

    def __init__(self, config: GatewayConfig):
        self.config = config
        self.client = anthropic.Anthropic()
        self.usage_log: list[dict] = []

    def route(self, request: GatewayRequest) -> str:
        # Select model based on use case
        model = (request.model_override
                 or self.config.model_overrides.get(
                     request.use_case,
                     self.config.default_model))

        # Check rate limits
        self._check_rate_limit(
            request.user_id, request.use_case)

        # Execute request
        response = self.client.messages.create(
            model=model, max_tokens=4096,
            messages=[{"role": "user",
                       "content": request.prompt}])

        # Log usage for cost tracking
        self.usage_log.append({
            "timestamp": datetime.now(
                timezone.utc).isoformat(),
            "user_id": request.user_id,
```

```
            "team": request.team,
            "use_case": request.use_case,
            "model": model,
            "input_tokens": response.usage.input_tokens,
            "output_tokens": response.usage.output_tokens,
        })
        return response.content[0].text

    def _check_rate_limit(self, user_id, use_case):
        # Implementation: count recent requests,
        # raise if over limit
        pass

    def cost_report(self, team=None) -> dict:
        """Generate cost report, optionally by team."""
        filtered = self.usage_log
        if team:
            filtered = [
                r for r in filtered
                if r["team"] == team]
        # Pricing per 1M tokens (approximate)
        pricing = {
            "claude-haiku-4-20250514": (0.25, 1.25),
            "claude-sonnet-4-20250514": (3.0, 15.0),
            "claude-opus-4-20250514": (15.0, 75.0),
        }
        total = 0.0
        by_use_case: dict[str, float] = {}
        for r in filtered:
            inp, out = pricing.get(
                r["model"], (3.0, 15.0))
            cost = (r["input_tokens"] * inp
                    + r["output_tokens"] * out) / 1_000_000
            total += cost
            by_use_case[r["use_case"]] = \
                by_use_case.get(r["use_case"], 0) + cost
        return {"total_cost": total,
                "by_use_case": by_use_case}
```

The Mesh Pattern

In larger organizations, a ***mesh pattern*** combines sidecars and gateways: each domain team runs its own sidecars and prompt libraries, but all traffic routes through a shared gateway for governance, cost tracking, and cross-domain coordination. This mirrors the data mesh philosophy of domain-oriented ownership with federated governance.

24.2 Team Structure and Cost Management

The Evolving Role Map

Table 24.1: Role evolution in a Claude Code-augmented platform

Traditional Role	**Evolved Role**	**Key Changes**
Data Engineer	AI-Augmented Data Engineer	Writes prompts alongside code; reviews AI output; focuses on architecture and system design
Analytics Engineer	Semantic Layer Architect	Designs natural language interfaces; curates prompt templates; owns business context encoding
Data Platform Engineer	AI Platform Engineer	Manages AI gateway infrastructure; optimizes token budgets; maintains MCP servers
(New)	Prompt Engineer / AI Ops	Develops and versions prompt libraries; monitors AI output quality; tunes retrieval systems

Use a *hub and spoke* model for team organization: a small AI Platform Hub (3–5 people) maintains the gateway, prompt libraries, and shared tooling. Domain spokes (each data team) each designate an "AI champion" who adapts shared tooling to their domain. A governance overlay sets policies for data classification, human review requirements, and cost allocation.

Warning

Do not create a centralized "AI team" that becomes a bottleneck for every AI-related request. We tried centralized everything in the 2010s; it was called "the data team that has a 6-month backlog." This pattern creates queues, slows iteration, and prevents domain teams from developing AI expertise. Instead, distribute AI capabilities across existing teams with a small platform team providing infrastructure, standards, and enablement—not gatekeeping.

Cost Optimization Strategies

AI API costs can grow rapidly without deliberate management. Five strategies keep costs under control:

1. **Model tiering**: Use Haiku for high-volume, low-complexity tasks (log classification, alert triage); Sonnet for code generation and analysis; Opus for architecture decisions and complex reasoning. A single pipeline monitoring system might use all three tiers depending on the task.
2. **Prompt caching**: Cache system prompts and schema context using Claude's prompt caching feature for up to 90% input cost reduction on repeated calls with the same system prompt.
3. **Batch API**: Use the Batch API for non-time-sensitive workloads (documentation generation, bulk code review, nightly quality checks) at 50% cost savings.
4. **Output management**: Constrain `max_tokens` to what each task actually needs. A classification task needs 50 tokens, not 4096.
5. **Smart context management**: Use embeddings or search to select only relevant files and schema definitions, rather than dumping entire codebases or full warehouse schemas into the context window.

Tip

Track cost-per-pipeline-run and cost-per-incident-resolved metrics from day one. These "unit economics" metrics demonstrate that Claude Code costs are offset by reduced engineering hours and faster resolution times. A $0.50 AI-assisted incident resolution that saves 45 minutes of engineer time is a clear win at any engineering salary. Your CFO speaks fluent ROI—learn the language.

Code Example

Cost tracking decorator

```
import functools, time

class CostTracker:
    """Track AI API costs per operation category."""

    # Approximate pricing per 1M tokens
    PRICING = {
        "claude-haiku-4-20250514": (0.25, 1.25),
        "claude-sonnet-4-20250514": (3.0, 15.0),
        "claude-opus-4-20250514": (15.0, 75.0),
    }

    def __init__(self):
        self.records: list[dict] = []

    def track(self, category: str):
        """Decorator to track costs of API calls."""
        def decorator(func):
            @functools.wraps(func)
            def wrapper(*args, **kwargs):
                start = time.time()
                result = func(*args, **kwargs)
                duration = time.time() - start
                # Extract usage from response
                if hasattr(result, 'usage'):
                    model = getattr(result, 'model', '')
                    inp, out = self.PRICING.get(
                        model, (3.0, 15.0))
                    cost = (
                        result.usage.input_tokens * inp
                        + result.usage.output_tokens * out
                    ) / 1_000_000
                    self.records.append({
                        "category": category,
                        "cost": cost,
                        "duration": duration,
                        "input_tokens":
                            result.usage.input_tokens,
                        "output_tokens":
                            result.usage.output_tokens,
```

```
                    })
                return result
            return wrapper
        return decorator
```

24.3 When NOT to Use Claude Code

Knowing when *not* to use AI is as important as knowing when to use it. Over-applying Claude Code adds latency, cost, and non-determinism without benefit.

Hard constraints—do not use for:

- **Deterministic computations**: Financial calculations, checksums, cryptographic operations. These require exact, reproducible results that LLMs cannot guarantee.
- **Sub-millisecond real-time decisions**: Fraud detection scoring, real-time bidding. LLM inference latency (100ms–5s) is orders of magnitude too slow.
- **Bulk data transformation**: Do not send millions of rows through an LLM. Use Claude Code to *generate* the transformation code, then execute that code directly.
- **Secrets and credentials management**: Never pass API keys, passwords, or tokens through an LLM. Use dedicated secrets managers.
- **Regulatory-prohibited contexts**: Some jurisdictions and industries prohibit AI decision-making in specific contexts. Know your regulatory landscape.

Soft constraints—exercise caution:

- **Highly novel domains**: If your domain has limited representation in training data, AI outputs will be less reliable. Validate more aggressively.
- **Safety-critical systems**: Use advisory mode with mandatory human review. Never let an agent autonomously modify production systems that could cause financial harm or data loss without approval gates.
- **Large-scale production schema migrations**: Plan with AI, but execute with tested, deterministic migration tooling (Flyway, Alembic, dbt). AI should generate and review the migration; automated tooling should apply it.

Warning

The most common mistake is applying Claude Code to every problem indiscriminately. Not every nail needs an AI-powered hammer. If a regex, a SQL `CASE` statement, or a simple Python script solves the problem reliably, use that. Using Claude Code to add two numbers together is like hiring a NASA engineer to change a lightbulb. Adding AI to simple deterministic tasks introduces latency, cost, and non-determinism without benefit. Reserve AI for tasks that genuinely require reasoning, language understanding, or pattern recognition across unstructured data.

24.4 Platform Maturity Model

Organizations adopt AI at different paces. This maturity model helps you assess where you are and plan where to go.

Level 1 — Ad Hoc Individual engineers use Claude Code via the web interface or CLI. No shared infrastructure, no cost tracking, no governance. Value is real but uncoordinated.

Level 2 — Standardized Shared API access via organization account. Basic prompt libraries for common tasks. Usage guidelines and cost awareness. This is the minimum viable level for a team.

Level 3 — Integrated Claude Code is embedded in CI/CD pipelines, quality checks, and incident response workflows. Centralized gateway provides cost tracking and audit logging. Prompt libraries are versioned and reviewed.

Level 4 — Optimized Model tiering routes tasks to the most cost-effective model. Prompt caching and batch processing reduce costs. A/B testing of prompts and models drives continuous improvement. RAG systems provide organizational knowledge grounding.

Level 5 — Autonomous Self-healing pipelines diagnose and fix common failures without human intervention. Schema evolution is AI-mediated for safe, backward-compatible changes. Agents coordinate across services to resolve complex incidents.

Note

Aim for Level 3–4 within the first year. Level 5 autonomy should be pursued selectively for well-understood, low-risk workflows only after you have robust monitoring, audit logging, and rollback mechanisms in place. Premature autonomy erodes trust when things go wrong—and they will go wrong, because entropy is undefeated.

24.5 Build vs. Buy Decisions

Every AI integration presents a build-versus-buy decision. The right answer depends on your team's size, domain complexity, and competitive landscape.

Build custom integrations when:

- Your domain has unique semantics that generic tools cannot capture (proprietary data models, custom pipeline frameworks).
- You need deep integration with proprietary internal systems.
- Prompt engineering and AI-powered workflows provide competitive advantage.
- Data sensitivity or regulatory requirements preclude third-party SaaS.

Buy (or use open-source) when:

- The vendor covers 80%+ of your requirements out of the box.
- The problem is commoditized (text-to-SQL, code review, documentation generation).
- Your team is small and cannot maintain custom AI infrastructure.
- Time-to-value is critical and the vendor offers fast onboarding.
- The vendor provides enterprise features (SSO, audit logging, compliance certifications) that you would otherwise need to build.

Tip

A useful heuristic: if the AI integration is *about* your data (schema-aware querying, domain-specific quality checks), build it. If the AI integration is *generic* (code formatting, documentation spell-checking, standard CI checks),

buy it. Your engineering time is better spent on integrations where domain knowledge creates defensible value.

24.6 Migration and Change Management

The Strangler Fig Pattern

Named after the strangler fig tree that gradually envelops its host, this pattern replaces manual processes with AI-augmented ones incrementally, never cutting over all at once.

1. **Shadow Mode (Weeks 1–4)**: Claude Code processes inputs in parallel with existing workflows. Outputs are logged and compared but never used in production. This builds confidence and reveals edge cases.
2. **Advisory Mode (Weeks 5–12)**: Claude Code recommendations are surfaced via Slack notifications or PR comments. Engineers choose whether to follow them. Track acceptance rates—they are your leading indicator of trust and quality.
3. **Supervised Mode (Months 3–6)**: Claude Code acts on pre-approved task categories with full logging and a human override window. If no human objects within the window (e.g., 15 minutes), the action proceeds.
4. **Autonomous Mode (Month 6+)**: Proven, high-confidence workflows run autonomously with monitoring, alerting, and automatic rollback on anomalies.

Every integration point needs four safety mechanisms: **feature flags** (disable any AI integration instantly), **fallback paths** (revert to pre-AI workflows), **provenance tagging** (mark AI-generated artifacts), and **circuit breakers** (auto-disable on excessive error rates).

Warning

Resist the temptation to skip Shadow Mode. "It worked in dev" is the data engineering equivalent of "hold my beer." Even if you are confident in the AI's capabilities, Shadow Mode reveals integration issues, edge cases, and trust concerns that only surface when the system runs against real production data and workflows. Two weeks of shadow running has prevented many costly production incidents.

24.7 Organizational Change Management

Technology transformation is 20% technology and 80% people. The most sophisticated AI platform will fail if the team does not trust it, does not understand it, or feels threatened by it. You can have the best Claude Code integration in the world, but if your team treats it like a haunted Ouija board, adoption will be zero. Address common concerns honestly:

"Will Claude Code replace my job?" Frame AI as a force multiplier, not a replacement. Engineers who leverage AI are *more* valuable because they produce higher quality work faster. The role evolves from writing boilerplate to designing systems, reviewing AI output, and solving novel problems that AI cannot handle alone.

"I don't trust AI-generated code." This is a healthy instinct. Channel it into building robust validation processes: automated tests, code review checklists for AI-generated code, and metrics that track AI output quality over time.

"This adds complexity." Acknowledge it honestly. AI integration *does* add complexity—another dependency, another failure mode, another cost center. Show concrete metrics proving that the value (faster development, fewer incidents, better documentation) offsets the complexity cost.

Build champions by giving early adopters dedicated experimentation time (10–20% of sprint capacity), creating shared channels for wins and patterns, and having them present at team meetings. Champions who speak from experience are far more persuasive than mandates from leadership.

Note

Track and share "AI wins" publicly. When an agent resolves a 3 AM incident without waking anyone, when AI-generated documentation passes review without changes, when a text-to-SQL query saves an analyst two hours—celebrate these moments. Concrete examples build organic adoption faster than any top-down mandate.

24.8 ROI Measurement and Governance

Table 24.2: ROI metrics for Claude Code-augmented platforms

Metric	Measurement	Target
Developer velocity	Tasks completed per sprint	30–50% increase in 6 months
Time to first pipeline	Days from request to production	50–70% reduction
Incident MTTR	Minutes from alert to resolution	40–60% reduction
Code review turnaround	Hours from PR to approval	60–80% reduction
Documentation coverage	% of pipelines with up-to-date docs	80%+ (from typical 20–30%)
On-call burden	After-hours pages requiring human action	50–70% reduction

Governance Framework

Establish policies for:

- **Data classification and AI eligibility**: Which data can be sent to AI APIs? PII, financial data, and health data may require on-premises models or special handling.
- **Human review requirements**: Which AI-generated outputs require human review before production use? All code changes? Only schema modifications?
- **Audit trail requirements**: Every AI interaction should log the prompt, response, model used, timestamp, and requesting user.
- **Incident response**: What happens when an AI-generated artifact causes a production issue? Who is responsible? How is it triaged differently from human-caused issues?
- **Prompt version control**: Treat prompts like code. Version them, review changes, and test before deployment.

Compliance considerations vary by industry: GDPR and CCPA require data processing agreements with AI providers; SOX and SOC 2 demand audit controls for AI in financial pipelines; HIPAA requires Business Associate Agreements with any provider processing health data.

Warning

Be honest in ROI calculations. We are data engineers—we of all people should not be making up numbers. Use conservative numbers validated with actual time-tracking data, not optimistic estimates. Overpromised ROI undermines trust in the entire initiative and makes it harder to secure budget for future improvements. A 25% improvement that you can prove is more valuable than a claimed 60% improvement that nobody believes.

12-Month Roadmap

Months 1–2: Foundation Set up API access, deploy the AI gateway, establish usage policies, and run the first pilot project (recommend starting with incident response or documentation generation).

Months 3–4: Expand Add 2–3 additional workflows (code review, data quality checks). Build prompt libraries. Implement cost tracking dashboards.

Months 5–6: Integrate Embed Claude Code in CI/CD pipelines. Deploy RAG for runbook automation. Set up quality monitoring. Reach Maturity Level 3.

Months 7–9: Optimize Implement model tiering, prompt caching, and batch processing. Run A/B tests on prompts and models. Publish the first ROI report with real data.

Months 10–12: Scale Roll out to all data teams. Enable supervised autonomy for proven workflows. Plan Level 4–5 initiatives for the following year.

Tip

This roadmap assumes a team of 8–15 engineers. Larger organizations may run phases in parallel across different teams. Smaller teams (3–5 engineers) should focus on fewer integration points per phase—start with the single highest-impact workflow and expand from there.

24.9 Exercises

1. **Architecture Design**: Design a Claude Code-augmented data platform architecture for a mid-size e-commerce company. Draw the five layers, identify

three specific AI integration points per layer, and specify which pattern (sidecar, gateway, or mesh) you would use and why.

2. **Cost Model**: Build a spreadsheet model for AI API costs based on your team's current workflow. Estimate daily query volume for five use cases (incident response, code review, documentation, data quality, ad hoc queries). Calculate monthly costs with and without model tiering, prompt caching, and batch processing.

3. **Maturity Assessment**: Assess your current organization against the five-level maturity model. For each gap between your current level and your target level, identify the specific actions, tools, and timeline needed to close it.

4. **Build vs. Buy Analysis**: Pick three AI-powered features you want to add to your platform (e.g., text-to-SQL, automated documentation, incident triage). For each, evaluate build vs. buy using the criteria in this chapter. Present your recommendation with estimated cost, timeline, and risk for each option.

5. **Change Management Plan**: Draft a change management plan for introducing Claude Code to a skeptical team of 10 data engineers. Include your Shadow Mode design, success metrics for each phase, a communication plan, and how you would handle the three common objections discussed in this chapter.

25

The Future of Data Engineering with AI

"The future of data engineering is bright. Mostly because Claude Code doesn't need office lighting."

Tip

The future of data engineering is not about replacement—it is about amplification. The data engineers who thrive will be those who wield AI as a force multiplier, treating Claude Code and similar tools as extensions of their expertise rather than threats to their livelihood. The robots are not coming for your job. They are coming for the boring parts of your job. The interesting parts are safe—for now.

Throughout this book, we have explored how Claude Code transforms every stage of the data engineering lifecycle. In this final chapter, we look toward the horizon: how will the profession evolve, what new patterns will emerge, and what should you do today to prepare for the next decade of data engineering?

25.1 The Evolving Role of the Data Engineer

From Builder to Orchestrator

The data engineer's primary activity is shifting from *writing* code to *directing*, *reviewing*, and *orchestrating* code generation. This does not mean data engineers

write less code—it means they write *different* code. The boilerplate shrinks while the architecture, validation, and integration logic grows.

Note

This shift does not mean less technical depth. Paradoxically, you need *more* expertise to review AI-generated code, identify subtle bugs that pass tests, and architect systems that AI tools can enhance effectively. A data engineer who does not understand query optimization cannot verify that an AI-generated query is efficient. You wouldn't trust a GPS that says "drive into the lake" just because it sounds confident. The AI raises the floor but does not lower the ceiling.

Dimension	Pre-Cloud (2005–15)	Cloud-Native (2015–24)	AI-Augmented (2024+)
Primary Activity	Managing infrastructure, writing MapReduce jobs	Building pipelines in SQL and Python	Orchestrating AI-generated pipelines, designing systems
Key Skill	Java, Hadoop administration	SQL, Python, cloud services, dbt	Prompt engineering, system design, AI output validation
Code Origin	100% human-written	95% human, 5% generated	40–70% AI-generated, human-reviewed
Iteration Speed	Days to weeks	Hours to days	Minutes to hours
Failure Mode	Hardware failures, cluster issues	Configuration drift, cloud misconfig	AI hallucination, prompt regression, context window limits

New Responsibilities

The AI-augmented data engineer takes on responsibilities that did not exist five years ago:

- **AI Output Quality Assurance**: Reviewing and validating AI-generated

code, SQL, and configurations before they reach production. This requires deeper domain expertise, not less, because you must catch errors that automated tests might miss.

- **Prompt Library Curation**: Maintaining versioned, tested prompt templates for common tasks—pipeline generation, schema review, incident investigation. These prompts become organizational assets, iterated and improved over time.
- **Cost Governance**: Managing AI API costs alongside traditional compute and storage costs. This includes model tiering decisions, prompt caching strategies, and usage budgets per team and use case.
- **Trust Calibration**: Developing an intuition for when to trust AI output at face value versus when to demand independent verification. This calibration is context-dependent and comes from experience.
- **AI-Human Interface Design**: Designing workflows where AI and human responsibilities are clearly delineated, handoffs are smooth, and neither side becomes a bottleneck.

Warning

The most dangerous position is uncritical acceptance of AI output. "Claude Code said it, I believe it, that settles it" is not an engineering methodology. Engineers who blindly copy-paste AI-generated code without review will introduce bugs, security vulnerabilities, and performance problems. The AI is a powerful first draft generator, not an infallible oracle. Always review, always test, always think critically about the output.

25.2 Emerging Patterns: AI-Native Data Platforms

While most organizations today are *retrofitting* AI onto existing platforms, a new generation of ***AI-native data platforms*** is being designed from the ground up with AI as a core component, not an add-on.

1. **Intent-Driven Configuration**: Engineers express what they want in natural language or high-level declarative specs. The platform translates intent into executable infrastructure, pipeline code, and monitoring rules. Instead of writing 500 lines of Airflow DAG code, the engineer writes a 20-line manifest describing the desired data flow.

2. **Self-Healing Pipelines**: When a pipeline fails, an agent automatically diagnoses the issue, generates a fix, tests it in a sandbox, and applies it—escalating to a human only when confidence is low or the fix involves a high-risk operation.

3. **Continuous Schema Evolution**: Rather than treating schema changes as disruptive migrations, AI mediates schema evolution continuously. It detects upstream changes, generates backward-compatible adaptations, notifies affected consumers, and applies changes incrementally.

4. **Semantic Metadata**: Every table, column, and pipeline has rich, AI-generated and human-reviewed metadata that enables natural language querying. Users ask "What percentage of our European customers made a purchase last month?" and the platform translates this to the correct SQL against the correct tables.

Code Example

AI-native platform manifest

```
platform:
  name: analytics-platform-v3
  intent: |
    Real-time analytics platform ingesting clickstream
    data, enriching with user profiles from PostgreSQL,
    serving aggregated metrics to the dashboard team.
    Maintain 30-second freshness for clickstream and
    5-minute freshness for user profiles.
  data_sources:
    - name: clickstream
      type: kafka
      topic: user.clicks
      freshness_sla: 30s
      volume: 50k_events_per_second
    - name: user_profiles
      source_type: postgresql
      connection: profiles_db
      change_capture: cdc
      freshness_sla: 5m
  transformations:
    - name: enriched_clicks
      intent: |
        Join clickstream events with user profiles.
        Add geographic region based on IP address.
        Filter bot traffic using the ua_parser library.
```

```
      output_schema: auto_evolve_with_review
  quality_expectations:
    completeness: 99.5%
    freshness_alert: 2x_sla
    schema_drift_policy: auto_evolve_with_review
    anomaly_detection: enabled
  serving:
    - name: dashboard_metrics
      type: materialized_view
      refresh: continuous
      consumers: [dashboard_team, exec_reports]
```

Note

Intent-driven configuration does not eliminate the need to understand the underlying systems. When the platform mis-translates intent (and it will), you need the expertise to diagnose and correct the generated implementation. Think of it as a dramatically faster starting point, not a finished product.

The Semantic Layer Revolution

The semantic layer—a business-friendly abstraction over raw data—is being transformed by AI. Traditional semantic layers require manual definition of metrics, dimensions, and relationships. AI-native semantic layers can:

- **Auto-discover relationships** between tables by analyzing column names, data types, foreign keys, and actual data distributions.
- **Generate metric definitions** from business documentation, Slack conversations, and existing dashboard queries.
- **Resolve ambiguity** in natural language queries by asking clarifying questions or presenting multiple interpretations.
- **Learn from corrections**: When a user says "That is not what I meant by revenue—I mean net revenue after refunds," the system updates its understanding for future queries.

25.3 The Convergence of Data Engineering and ML Engineering

One of the most significant trends is the dissolving boundary between data engineering and ML engineering. As AI becomes central to data platforms, the skill sets converge:

- **Data engineers embed ML models inside pipelines**: Anomaly detection on streaming data, entity resolution during ingestion, data quality scoring at each transformation step.
- **ML engineers need deep data engineering skills**: Feature stores, training data pipelines, model serving infrastructure, and data versioning are all data engineering problems.
- **Both roles require prompt engineering**: Working with LLMs is a core skill for both disciplines, whether for code generation, data analysis, or building AI-powered features.

The result is a new hybrid role—the ***AI data engineer***—who combines pipeline expertise with ML literacy and prompt engineering fluency. This does not mean everyone needs to be an expert in all three areas, but baseline competence across all three is becoming table stakes.

Tip

If you are a data engineer, invest in learning ML fundamentals: how models are trained, what feature engineering means, how evaluation metrics work, and how models are deployed and monitored. You do not need to derive gradient descent from first principles, but you need to have an informed conversation with ML engineers and understand what their systems need from yours.

25.4 Natural Language as the New Interface for Data

The Text-to-SQL Revolution

Text-to-SQL has been a research topic for decades, but Claude Code's capabilities make it practical for production use for the first time. Modern text-to-SQL systems provide full schema awareness, business context understanding, dialect-specific SQL generation, query safety guardrails, and multi-turn refinement.

Code Example

Text-to-SQL service with guardrails

```
from anthropic import Anthropic
import sqlglot

class TextToSQLService:
    """Convert natural language to safe SQL queries."""

    def __init__(self, schema_context: str,
                 dialect: str = 'snowflake'):
        self.client = Anthropic()
        self.schema_context = schema_context
        self.dialect = dialect
        self.blocked_ops = [
            'DROP', 'DELETE', 'TRUNCATE',
            'INSERT', 'UPDATE', 'ALTER', 'CREATE']

    def generate_sql(self, question: str) -> str | None:
        """Generate SQL from a natural language question."""
        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=2048,
            system=(
                f'You are a SQL expert for {self.dialect}. '
                f'Schema:\n{self.schema_context}\n'
                f'Rules:\n'
                f'- Only generate SELECT statements\n'
                f'- Use CTEs for complex logic\n'
                f'- Include LIMIT unless aggregating\n'
                f'- Add column aliases for clarity\n'
                f'- Return only the SQL, no explanation'),
            messages=[{
                "role": "user",
                "content": question
            }],
        )
        sql = self._extract_sql(response.content[0].text)
        if self._validate_sql(sql):
            return sql
        return None

    def _validate_sql(self, sql: str) -> bool:
```

```
        """Validate SQL is safe to execute."""
        try:
            parsed = sqlglot.parse_one(sql)
            # Check for prohibited operations
            for node in parsed.walk():
                if hasattr(node, 'key'):
                    if node.key.upper() in self.blocked_ops:
                        return False
            return True
        except sqlglot.errors.ParseError:
            return False

    def _extract_sql(self, text: str) -> str:
        """Extract SQL from model response."""
        if '```sql' in text:
            return text.split('```sql')[1] \
                       .split('```')[0].strip()
        if '```' in text:
            return text.split('```')[1] \
                       .split('```')[0].strip()
        return text.strip()
```

Beyond Text-to-SQL: Conversational Analytics

Text-to-SQL is just the beginning. ***Conversational analytics*** extends this into multi-turn interactions where users explore data through dialogue:

- "Show me revenue by region for Q1." (Initial query)
- "Break that down by product category." (Refinement—system adds GROUP BY)
- "Why did Europe drop compared to last quarter?" (Analysis—system generates comparison query and interprets results)
- "Create a weekly alert if any region drops more than 15%." (Action—system generates monitoring rule)

Each turn builds on the previous context, and the system maintains state across the conversation. This transforms data exploration from a technical skill into a natural conversation.

Note

Conversational analytics does not replace dashboards or BI tools. It complements them by handling ad hoc questions that do not warrant a new dashboard panel. The sweet spot is questions that are too specific for a pre-built dashboard but too common to justify a data engineering ticket every time—a.k.a. the "Can you just pull a quick number for me?" category that has consumed approximately 40% of all data engineering hours since the dawn of the profession.

25.5 The AI-Augmented Data Engineer's Day

What does a typical day look like for an AI-augmented data engineer? The contrast with the traditional workflow is striking:

Time	Traditional Workflow	AI-Augmented Workflow
9:00 AM	Check dashboards, read through logs manually	Claude Code has triaged overnight failures, proposed fixes, and auto-resolved two routine issues
10:00 AM	Write dbt models from scratch	Describe the business logic; review and refine generated models
11:00 AM	Code review: read every line carefully	AI pre-reviews PRs with detailed comments; engineer focuses on architecture and business logic
1:00 PM	Debug a slow query by reading EXPLAIN plans	Feed query plan to Claude Code for optimization suggestions with rationale
2:00 PM	Write documentation from memory	Claude Code generates docs from code and commit history; engineer reviews for accuracy
3:00 PM	Handle ad hoc data requests from analysts	Direct analysts to the natural language query interface for self-service

Time	Traditional Workflow	AI-Augmented Workflow
4:00 PM	Sprint planning: estimate story points	Use Claude Code to analyze ticket descriptions, estimate complexity, identify risks

The Pair Programming Paradigm

The most productive data engineers treat Claude Code as a pair programming partner rather than an autocomplete engine. This means engaging in genuine collaboration:

- **Think out loud**: Explain your reasoning before asking for code. "I need to deduplicate events, but some events are legitimately repeated. The key should be user_id + event_type + timestamp within a 5-second window." This context dramatically improves generated code quality.
- **Iterate, don't perfect**: Start with a rough description and refine through conversation rather than spending 20 minutes crafting the perfect prompt. Three rounds of iteration typically produce better results than one perfect prompt.
- **Challenge**: Ask Claude Code to identify weaknesses in your approach. "What edge cases am I missing?" and "How would this break at 10x scale?" often surface issues you had not considered.
- **Learn**: Use responses as learning opportunities. When Claude Code suggests an approach you have not seen before, ask it to explain the tradeoffs. This accelerates skill development.
- **Verify**: Always run the code. Always check the output. Trust but verify, especially for SQL queries that touch production data.

Tip

Keep a personal "prompt journal"—a file where you save prompts that worked exceptionally well, along with notes on why they worked. Over time, this becomes a powerful reference that captures your personal prompt engineering expertise. Share the best entries with your team.

25.6 Career Advice for Data Engineers in the AI Era

The data engineering job market is evolving rapidly. Here is practical guidance for navigating the transition.

Skills That Become More Valuable

System Design and Architecture Understanding how systems fit together, how data flows end-to-end, and how to design for reliability, scalability, and maintainability. AI cannot replace the architect who understands the business and its constraints.

Data Modeling Dimensional modeling, graph modeling, and schema design require deep understanding of the business domain. AI can suggest schemas, but a human must validate them against business reality.

Business Domain Expertise The engineer who understands healthcare billing, financial compliance, or supply chain logistics brings irreplaceable context that makes AI outputs more accurate and relevant.

Quality Engineering Designing comprehensive data quality frameworks, defining expectations, and building monitoring systems. As more code is AI-generated, quality assurance becomes more critical, not less.

Communication and Leadership Explaining technical decisions to stakeholders, mentoring junior engineers on AI collaboration, and building the case for AI investment require human skills that AI amplifies but does not replace.

Skills That Become Table Stakes

These skills remain necessary but are no longer sufficient to differentiate you:

- Boilerplate SQL and Python writing (AI handles first drafts)
- Configuration syntax memorization (AI generates configs from descriptions)
- API documentation lookup (AI synthesizes API usage from docs)
- Routine debugging of common errors (agents handle known patterns)
- Manual documentation writing (AI generates from code; you review and refine)

Warning

AI is a leverage multiplier: the more expertise you bring, the more leverage you gain. A chainsaw is far more dangerous in the hands of someone who has never used one. A senior engineer who masters AI collaboration can produce the output of an entire team while maintaining quality standards that only experience provides. But a junior engineer using AI without foundational knowledge will produce plausible-looking code that fails in subtle, expensive ways. Invest in fundamentals first, then amplify with AI.

Career Development Strategies

1. **Go deep on architecture**: Invest in distributed systems, streaming architecture, and data modeling. These are the skills AI amplifies most.

2. **Build a portfolio of AI-augmented projects**: Show that you can use AI tools effectively, not just that you can code. Employers increasingly value this.

3. **Develop business domain expertise**: Specialize in an industry (finance, healthcare, e-commerce) where your domain knowledge makes AI outputs more valuable.

4. **Learn to evaluate AI output critically**: This is a meta-skill that applies across all tasks. Practice by asking Claude Code to solve problems you already know the answer to, then analyzing where it gets things right and wrong.

5. **Contribute to the community**: Write about your experiences with AI-augmented data engineering. The field is young enough that practitioners who share knowledge become recognized experts.

25.7 Predictions for 2027–2030

Predicting the future is inherently uncertain, but the following trajectories are supported by current trends and investment patterns.

2027: Integration Becomes Standard

- AI-native orchestrators ship with built-in LLM integration for pipeline generation, monitoring, and remediation. Dagster, Prefect, and next-generation tools offer "describe your pipeline" interfaces.
- Conversational data catalogs become the default way business users discover and understand data. Keyword search in catalogs feels as dated as command-line-only interfaces.
- 50%+ of new pipeline code in progressive organizations is AI-generated, human-reviewed. The review process itself is partially AI-assisted.
- MCP becomes the dominant standard for AI-to-tool integration, with community-maintained servers for every major data tool.

2028: Platform Convergence

- The distinction between data lake, data warehouse, and streaming platform blurs further. Unified platforms handle batch, streaming, and AI workloads through a single interface.
- Self-tuning pipelines adjust parallelism, resource allocation, and scheduling automatically based on workload patterns. Manual performance tuning becomes the exception, not the rule.
- Governments establish AI-specific data governance regulations. Data engineers who understand both AI capabilities and regulatory requirements become highly sought after.
- Agent-to-agent coordination matures. Specialized agents for different domains (ingestion, transformation, serving, quality) collaborate through shared protocols to resolve complex incidents.

2029–2030: The Autonomous Horizon

- Intent-driven platforms become the default for new projects. Engineers describe what they want; the platform figures out how to build it.
- Level 4 autonomous data operations are common in mature organizations. Agents handle 80%+ of routine operational tasks without human intervention.

- A single senior AI-augmented data engineer manages infrastructure that previously required 5–10 people. This does not mean fewer data engineers—it means more data infrastructure per engineer and more ambitious projects becoming feasible.
- The "AI data engineer" role is mainstream, combining pipeline expertise, ML literacy, prompt engineering, and domain knowledge.

Note

These predictions assume continued rapid progress in AI capabilities. If progress plateaus or hits diminishing returns, the timeline stretches, but the direction remains the same. The question is not *whether* these changes happen, but *when* and *how fast*. Of course, predicting the future of AI is itself the kind of task you probably should not trust an AI to do. The irony is not lost on us.

25.8 Building a Learning Culture Around AI

Organizations that successfully integrate AI into their data engineering practice share common cultural traits:

- **Psychological safety**: No blame for AI-related mistakes. When an AI-generated query causes an incident, the response is "How do we improve our review process?" not "Why did you trust the AI?"
- **Experimentation time**: 10–20% of sprint capacity dedicated to AI experimentation. This is not optional—teams that cannot experiment cannot learn, and teams that cannot learn cannot improve.
- **Shared learning**: Dedicated channels (Slack, Teams) for effective prompts, surprising interactions, lessons learned, and tool recommendations. Knowledge sharing accelerates adoption across the team.
- **AI champions**: Senior engineers who curate prompt libraries, run workshops, mentor on AI collaboration, and stay current with the rapidly evolving landscape. Every team should have at least one.
- **Structured learning cadence**: Weekly AI digests (curated links and tips), monthly deep dives (hands-on workshops on specific techniques), quarterly hackathons (build something new with AI), and annual strategy reviews (where are we, where are we going).

Tip

Start a "Prompt of the Week" practice where team members share their most useful prompt discovery. This low-effort, high-impact practice builds collective prompt engineering expertise faster than any training program. Plus, it's the one Slack channel people actually read.

25.9 Ethical Considerations

As AI becomes deeply embedded in data engineering, ethical considerations demand explicit attention:

- **Bias propagation**: AI-generated data transformations can inadvertently encode or amplify biases present in training data. Review AI-generated business logic for fairness implications, especially in customer-facing analytics.
- **Transparency**: Stakeholders have a right to know when data products are AI-generated or AI-assisted. Label AI-generated artifacts clearly.
- **Environmental impact**: Large language models consume significant compute resources. Use model tiering and prompt caching not just for cost savings but as a responsible resource usage practice.
- **Skill atrophy**: If engineers over-rely on AI, foundational skills can atrophy. Maintain deliberate practice of core skills (writing SQL from scratch, manual debugging) alongside AI-augmented workflows.

Warning

Do not let AI-generated code become a "black box" in your pipeline. "I don't know what it does, but it works" was already a bad excuse for human-written code; it does not get better with AI-generated code. Every AI-generated transformation should be reviewable, testable, and explainable. If you cannot explain what a piece of AI-generated code does and why, it should not be in production.

25.10 Final Thoughts and Call to Action

The AI-augmented data engineer rethinks their approach to every aspect of the profession:

- Instead of "How do I write this pipeline?" they ask "What should this pipeline accomplish, and how do I verify it works correctly?"
- Instead of being the bottleneck for every data request, they build natural language interfaces that enable self-service.
- Instead of guarding knowledge in their heads, they encode it in AI-augmented workflows, RAG systems, and prompt libraries that scale beyond any individual.
- Instead of fearing obsolescence, they embrace the force multiplier that makes their expertise more valuable and impactful than ever.

The transition is not always smooth. There will be failed experiments, AI-generated bugs in production, and moments of frustration when the model does not understand what seems obvious. But the trajectory is clear: AI-augmented data engineering is not the future—it is the present, and the gap between teams that embrace it and those that resist it will only widen.

Your Call to Action

Start today:

1. Pick one repetitive task you do weekly and automate it with Claude Code.
2. Share what you learn—the wins and the failures—with your team.
3. Build a personal prompt library and iterate on it.
4. Experiment with agentic workflows for a low-risk operational pipeline.
5. Measure the impact: time saved, quality improved, incidents prevented.
6. Advocate for broader adoption based on your real results, not hype.

The data engineers who start now will be the architects and leaders of tomorrow's AI-native data platforms. The rest will be writing "we should have adopted AI sooner" in their retrospectives. Written by Claude Code, of course.

25.11 Exercises

1. **Self-Assessment**: Assess your current skills across four dimensions: technical depth (SQL, Python, distributed systems), AI collaboration (prompt engineering, output validation), breadth (cloud platforms, streaming, ML fundamentals), and leadership (communication, mentoring, architecture). Rate yourself 1–5 on each, identify your three largest gaps, and create a six-month development plan with specific actions and milestones.

2. **AI-Native Pipeline Design**: Redesign a pipeline you currently maintain using the intent-driven pattern. Write the natural language manifest (similar to the example in this chapter), then compare the manifest to your existing implementation. What information is captured in your code that the manifest misses? What does the manifest express more clearly?

3. **Text-to-SQL Prototype**: Build a text-to-SQL service for one of your databases using the pattern in this chapter. Test it with ten real questions from your stakeholders. Measure accuracy (correct SQL generated), safety (no prohibited operations), and usability (would the stakeholder accept this answer?). Identify the failure modes and propose improvements.

4. **Daily Workflow Audit**: Track every activity you perform for one full work week. For each activity, estimate how much time Claude Code could save (0%, 25%, 50%, 75%, or 90%). Calculate total potential savings. Identify the top three candidates for AI augmentation and implement one of them.

5. **Future Architecture**: Design a complete AI-native data platform for an e-commerce company processing 100M events per day. Include: the platform intent manifest, data product specifications for three key products (customer 360, real-time recommendations, executive dashboard), the team structure and roles required, the maturity model progression plan for the first 18 months, and cost projections for both compute and AI API usage.

Claude API Reference Quick Guide

Claude Code API Reference Quick Guide

"In case of emergency, consult this appendix. In case of existential crisis about whether you should have used Claude Code for this, also consult this appendix."

Tip

This appendix provides a condensed reference for the Claude Code API. For current information, consult `docs.anthropic.com`. This book was written by a human and may go stale. The docs are updated by—well, probably also Claude Code at this point.

Messages API Endpoint Reference

Property	Value
URL	`POST https://api.anthropic.com/v1/messages`
Authentication	`x-api-key` header
Content Type	`application/json`
API Version	`anthropic-version: 2023-06-01`

Key parameters: `model` (required), `messages` (required), `max_tokens` (required), `system`, `temperature` (0.0–1.0), `stream`, `tools`, `tool_choice`, `stop_sequences`.

```
curl https://api.anthropic.com/v1/messages \
```

```
  -H 'Content-Type: application/json' \
  -H 'x-api-key: $ANTHROPIC_API_KEY' \
  -H 'anthropic-version: 2023-06-01' \
  -d '{
    "model": "claude-sonnet-4-20250514",
    "max_tokens": 1024,
    "system": "You are a SQL expert for Snowflake.",
    "messages": [{"role": "user", "content": "Write a query to find
    duplicates"}]
  }'
```

Listing 1: Basic Messages API request

Model IDs and Selection Guide

Model ID	Context	Speed	Cost	Best For
`claude-opus-4-20250514`	200K	Slowest	Highest	Complex architecture, nuanced review
`claude-sonnet-4-20250514`	200K	Medium	Medium	General coding, SQL, most DE tasks
`claude-haiku-3-5-20241022`	200K	Fastest	Lowest	Classification, extraction, high-volume

Note

Pin versioned model IDs in configuration files for production to avoid unexpected behavior changes. "Our pipeline broke because the model got smarter overnight" is a real sentence that real engineers have said. Pin your versions.

Rate Limits and Error Codes

Rate limits vary by tier (Tier 1: 50 req/min, Tier 4: 4,000 req/min). Key error codes: 400 (invalid request), 401 (auth error), 429 (rate limit—implement exponential backoff with jitter), 500 (server error—retry), 529 (overloaded—retry with backoff).

Warning

Always implement exponential backoff with jitter for 429 responses. Do not retry immediately—hammering a rate-limited endpoint is the API equivalent of pressing the elevator button faster to make it arrive sooner. It does not work and annoys everyone involved.

Tool Use Schema Reference

Tool use enables Claude Code to invoke external functions. The flow: (1) send tools in the request, (2) Claude Code responds with `tool_use` blocks, (3) execute the tool and return results, (4) Claude Code incorporates results.

```python
from anthropic import Anthropic
client = Anthropic()

def run_tool_loop(user_message, tools, tool_handlers):
    messages = [{"role": "user", "content": user_message}]
    while True:
        response = client.messages.create(
            model="claude-sonnet-4-20250514", max_tokens=4096,
            tools=tools, messages=messages,
        )
        tool_results = []
        final_text = []
        for block in response.content:
            if block.type == "text":
                final_text.append(block.text)
            elif block.type == "tool_use":
                result = tool_handlers[block.name](block.input)
                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": str(result),
                })
        if not tool_results:
            return "\n".join(final_text)
        messages.append({"role": "assistant", "content": response.content})
        messages.append({"role": "user", "content": tool_results})
```

Listing 2: Tool use flow

Streaming and Batch API

Streaming: Use `client.messages.stream()` for incremental results in interactive tools.

Batch API: Process large volumes asynchronously within 24 hours at 50% cost reduction—ideal for bulk documentation, quality scans, and batch SQL validation.

Tip

Use the Batch API for any task that does not require immediate results—nightly quality scans, weekly documentation updates, or bulk schema analysis. Patience saves 50%. If only that worked at restaurants.

Python SDK Quick Reference

```python
from anthropic import Anthropic, AsyncAnthropic

client = Anthropic()  # Reads ANTHROPIC_API_KEY from env

# Simple message
response = client.messages.create(
    model="claude-sonnet-4-20250514", max_tokens=1024,
    messages=[{"role": "user", "content": "Hello, Claude Code"}],
)
print(response.content[0].text)

# With prompt caching
response = client.messages.create(
    model="claude-sonnet-4-20250514", max_tokens=1024,
    system=[{
        "type": "text", "text": large_schema_context,
        "cache_control": {"type": "ephemeral"}
    }],
    messages=[{"role": "user", "content": "List all tables"}],
)
```

Listing 3: Python SDK essentials

Note

Both Python and TypeScript SDKs read `ANTHROPIC_API_KEY` from the environment. In production, store keys in a secrets manager and inject at runtime. If you hardcode your API key, you will end up on a very expensive and very public list of "things found in GitHub repos."

Prompt Engineering Patterns for Data Engineers

> *"A prompt engineer is just a regular engineer who has learned to say 'please' and 'be specific' to a computer."*

This appendix provides a catalog of reusable ***prompt engineering patterns*** for data engineering workflows. Think of these as copy-paste recipes for talking to Claude Code without it going off on a creative tangent about the philosophy of NULL values. Each pattern includes: the problem it solves, a parameterized template, and guidance on when to apply it.

The Anatomy of an Effective Data Engineering Prompt

Note

Every effective data engineering prompt contains four layers: **Context** (system and schema), **Intent** (what to produce), **Constraints** (rules to obey), and **Format** (output structure). We call this the ***CICF framework***.

The single most impactful improvement is including actual DDL for relevant tables. Claude Code reasons about column types, foreign keys, and constraints far more accurately given real schemas rather than prose descriptions. Telling Claude Code "I have a table with some columns" and expecting correct SQL is like telling a chef "I have some ingredients" and expecting a Michelin-star meal.

Tip

Use `SHOW CREATE TABLE` or your ORM's schema dump. For large schemas, include only relevant tables plus their foreign key references.

Pattern 1: Schema-Aware SQL Generation

Problem: Generate SQL queries correct with respect to a specific schema.

Code Example

Template

```
You are a data engineer working with [DATABASE_PLATFORM].

## Schema
```sql
[PASTE DDL HERE]
```

Task
Write a SQL query that [OBJECTIVE].

Requirements
- Use CTEs for readability. Handle NULLs explicitly.
- Optimize for [PERFORMANCE_GOAL].
- Target [SQL DIALECT] syntax.
```

Pattern 2: Documentation Generator

Problem: Generate documentation from existing code or schemas.

Code Example

Template

```
You are a technical writer for data engineering.

## Source Material
```

```
```[LANGUAGE]
[PASTE CODE, DDL, OR CONFIGURATION]
```

## Task
Generate [table / pipeline / API] documentation.
- Audience: [junior engineers / analysts / stakeholders]
- Include: purpose, inputs, outputs, dependencies, limitations.
- Format: [Markdown / YAML / Confluence]
```

Pattern 3: Code Review Assistant

Problem: Review data engineering code for correctness, performance, security, and maintainability.

Code Example

Template

```
Senior data engineer code review.

## Code
```[LANGUAGE]
[PASTE CODE]
```

## Context
- Environment: [ENV]. Volume: [VOLUME]. Frequency: [FREQ].

## Evaluate
1. Correctness (logic errors, edge cases, NULL handling)
2. Performance (joins, indexes, full scans)
3. Security (injection, credentials, permissions)
4. Resilience (error handling, retry, idempotency)

## Output per finding
- Severity: Critical / Major / Minor
- Location, Issue, Recommendation, Example fix
```

Warning

Never blindly apply code review suggestions. CLAUDE CODE may flag idiomatic patterns as issues—it once suggested "fixing" a perfectly valid Python walrus operator because it looked like a typo. Always validate against your specific context.

Pattern 4: Data Quality Analyzer

Problem: Generate data quality checks for a dataset or table.

Code Example

Data Quality Analyzer Template

```
You are a data quality engineer.

## Table Schema
```sql
[PASTE DDL HERE]
```

Business Context
[DESCRIBE WHAT THIS TABLE REPRESENTS]

Task
Generate quality checks covering:
1. Completeness (NULL rates for each column)
2. Uniqueness (PK/NK uniqueness)
3. Validity (ranges, formats, referential integrity)
4. Consistency (cross-column logic)
5. Timeliness (data freshness)
6. Statistical (volume anomalies vs. historical)

Return SQL queries for [PLATFORM], one per check.
```

Worked Example

Code Example

Generated Data Quality Checks

```sql
-- CHECK 1: Primary key uniqueness (expect 0 duplicates)
SELECT order_id, COUNT(*) AS cnt
FROM orders GROUP BY order_id HAVING COUNT(*) > 1;

-- CHECK 2: Referential integrity (expect 0 orphans)
SELECT o.order_id, o.customer_id
FROM orders o
LEFT JOIN customers c ON o.customer_id = c.customer_id
WHERE c.customer_id IS NULL;

-- CHECK 3: Temporal consistency (ship never before order)
SELECT order_id, order_date, ship_date
FROM orders WHERE ship_date < order_date;

-- CHECK 4: Volume anomaly (daily counts within 2 std devs)
WITH daily_counts AS (
    SELECT DATE_TRUNC('day', created_at) AS dt, COUNT(*) AS cnt
    FROM orders
    WHERE created_at >= DATEADD('day', -30, CURRENT_DATE)
    GROUP BY 1
), stats AS (
    SELECT AVG(cnt) AS avg_cnt, STDDEV(cnt) AS std_cnt
    FROM daily_counts
)
SELECT dc.dt, dc.cnt
FROM daily_counts dc CROSS JOIN stats s
WHERE ABS(dc.cnt - s.avg_cnt) > 2 * s.std_cnt;
```

Pattern 5: Migration Script Generator

Problem: Generate safe, reversible database migration scripts.

Provide current DDL, desired DDL, platform, table size. Request UP and DOWN scripts with pre-migration validation, transactions, and post-migration checks. Specify zero-downtime requirements.

Pattern 6: Incident Root Cause Analyzer

Problem: Systematically diagnose a pipeline failure from logs and system state.

Provide: incident summary, error logs (with timestamps), system context (pipeline, orchestrator, platform), recent changes, and steps already tried. Request: error categorization, top 3 causes ranked by probability, diagnostic queries for each, and immediate mitigation plus long-term fix.

Tip

Include timestamps in log entries and preserve chronological order. Redact secrets but keep connection strings (with masked passwords) for diagnostic clues.

Pattern 7: Cost Optimization Advisor

Problem: Identify optimization opportunities in queries, storage, or compute.

Provide: platform, monthly spend, expensive query with execution plan, and optimization goals. Request: line-by-line cost driver analysis, recommendations ranked by savings, rewritten queries, and trade-off assessment.

Pattern 8: Data Contract Generator

Problem: Define formal data contracts between producers and consumers.

Code Example

Data Contract Generator Template

````
You are a data governance specialist.

## Dataset
- Name: [NAME]. Owner: [TEAM]. Description: [WHAT IT IS].

## Schema
```sql
[PASTE DDL]
```

````

```
## Consumers
[LIST TEAMS AND HOW THEY USE THE DATA]

## Task
Generate YAML data contract with:
1. Schema: every field with type, description, PII flag
2. Quality: completeness, uniqueness, freshness rules
3. SLA: latency, availability, incident response time
4. Ownership: producer, consumers, escalation contacts
5. Change management: how changes are proposed/reviewed
```

Worked Example

Code Example

Generated Data Contract

```yaml
dataContract:
  name: user_transactions
  version: "2.1.0"
  owner:
    team: payments-data-eng
    slack: "#payments-data"
  schema:
    fields:
      - name: transaction_id
        type: STRING
        nullable: false
        pii: false
        primaryKey: true
      - name: user_id
        type: STRING
        nullable: false
        pii: true
      - name: amount_cents
        type: INTEGER
        nullable: false
        validRange: {min: 1, max: 100000000}
  quality:
    completeness: {transaction_id: 100%, user_id: 100%}
    uniqueness: [transaction_id]
```

```
    freshness: {maxDelayMinutes: 30}
  sla:
    availability: 99.9%
    maxLatencyMinutes: 30
```

Pattern 9: dbt Model Generator

Problem: Create dbt models following project conventions.

Provide: naming conventions, materialization defaults, testing strategy, source DDL, model requirements (name, layer, grain, business logic, incremental strategy). Generate: SQL model with Jinja/ref(), schema YAML with tests, and source definitions.

Pattern 10: Terraform Module Generator

Problem: Generate IaC for data infrastructure components.

Code Example

Template

```
Cloud infrastructure engineer for data platforms.

## Requirements
- Provider: [AWS / GCP / Azure]. Component: [DESCRIBE].
- Security: [encryption, network, IAM, compliance]
- Operations: [scaling, monitoring, backup]

## Task
Generate: main.tf, variables.tf, outputs.tf, usage example.
```

Worked Example

Code Example

Generated Terraform – S3 Data Lake

```
resource "aws_s3_bucket" "data_lake" {
  bucket = "${var.project}-${var.environment}-data-lake"
  tags = merge(var.common_tags, {
    Layer = "bronze"
    ManagedBy = "terraform"
  })
}

resource "aws_s3_bucket_versioning" "data_lake" {
  bucket = aws_s3_bucket.data_lake.id
  versioning_configuration { status = "Enabled" }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "
    data_lake" {
  bucket = aws_s3_bucket.data_lake.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm     = "aws:kms"
      kms_master_key_id = var.kms_key_arn
    }
  }
}

resource "aws_s3_bucket_public_access_block" "data_lake" {
  bucket                  = aws_s3_bucket.data_lake.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}
```

Anti-Patterns to Avoid

Warning

Vague requests: "Write SQL for analytics" with no schema or metrics. CLAUDE CODE is good, but it cannot read your mind. Yet. Fix: provide DDL and specific objectives.
Kitchen sink: Asking for schema design, ETL, tests, docs, and monitoring in one prompt. This is the prompt equivalent of a 47-item grocery list scrawled on a napkin. Fix: break into sequential prompts.
No platform specified: Optimization strategies differ radically between Snowflake, BigQuery, and Spark SQL. "SQL is SQL" said no one who has ever debugged a dialect mismatch.
Ignoring data volume: A correct solution for 10K rows may fail at 10B rows. "It worked in dev" does not count.
Pasting credentials: Security risk and biases output. Fix: sanitize with synthetic data. Your API key does not need to be in a book.
No output examples: "Generate a report" without showing format. Fix: include 2–3 rows of desired output.
Trusting without validation: Never run AI-generated SQL against production without review. We cannot stress this enough. We have stressed it in every chapter. We will stress it again.

Composing Patterns for Complex Workflows

New source onboarding: Pattern 1 (SQL gen) → Pattern 9 (dbt) → Pattern 4 (quality) → Pattern 2 (docs) → Pattern 8 (contract).

Schema migration: Schema Evolution Advisor → Pattern 5 (migration) → Pattern 9 (dbt update) → Pattern 4 (validation) → Pattern 3 (code review).

Incident response: Pattern 6 (RCA) → Pipeline Debug → Pattern 1 (fix) → Pattern 4 (prevention) → Pattern 2 (runbook update).

Use patterns when the task is repeatable, high-stakes, involves complex context, or needs consistent quality across team members. For quick one-off questions, freeform prompts work fine.

Note

Save customized templates, successful outputs, and lessons learned in a shared repository. Over time, your team builds a library of battle-tested prompts encoding your organization's standards. Think of it as institutional memory that does not leave when someone accepts an offer from a competitor.

Cost Optimization Strategies

"The cloud was supposed to save us money. Then AI was supposed to save us money. At some point, we should probably stop 'saving' money."

Deploying Claude Code at scale requires careful cost management. This appendix covers understanding, monitoring, and optimizing API spending—because the only thing worse than a broken pipeline is a working pipeline that costs more than the data it produces.

Understanding Claude Code Pricing

Token Basics

A ***token*** is roughly 3–4 characters of English text. Code tokenizes less efficiently—SQL averages about 2.5 characters per token.

Table 3: Approximate token counts for common artifacts.

Artifact	Chars	Tokens
Simple SELECT (5 cols)	200	60
Complex CTE query (100 lines)	3,000	900
Full schema (50 tables)	25,000	7,500
Python ETL script (200 lines)	6,000	1,800

Table 4: Claude Code model pricing (per million tokens, USD).

Model	Input ($/1M)	Output ($/1M)	Context
Haiku	$0.25	$1.25	200K
Sonnet	$3.00	$15.00	200K
Opus	$15.00	$75.00	200K

Per-Model Pricing

Warning

Output tokens are 5× more expensive than input tokens. Claude Code is like a consultant: listening is cheap, but the moment it starts talking, the meter is running hard. Controlling output length is one of the most effective cost levers. Including output format examples reduces verbosity.

Token Counting and Estimation

Code Example

Token Counting and Cost Estimation

```python
import anthropic

client = anthropic.Anthropic()

def count_tokens(text, model="claude-sonnet-4-20250514"):
    response = client.messages.count_tokens(
        model=model,
        messages=[{"role": "user", "content": text}]
    )
    return response.input_tokens

def estimate_cost(input_tokens, output_tokens, model="sonnet"):
    pricing = {
        "haiku":  {"input": 0.25, "output": 1.25},
        "sonnet": {"input": 3.00, "output": 15.00},
        "opus":   {"input": 15.00, "output": 75.00},
    }
```

```
    p = pricing[model]
    return round(
        (input_tokens / 1e6) * p["input"]
        + (output_tokens / 1e6) * p["output"], 6
    )
```

Prompt Caching

Prompt caching reduces costs when you repeatedly send the same prompt prefix (system prompt, schema context). First request caches at 1.25× base cost; subsequent reads at 0.10× base.

Code Example

Prompt Caching Implementation

```
def query_with_caching(client, schema_context, question):
    response = client.messages.create(
        model="claude-sonnet-4-20250514", max_tokens=2048,
        system=[{
            "type": "text", "text": schema_context,
            "cache_control": {"type": "ephemeral"}
        }],
        messages=[{"role": "user", "content": question}]
    )
    usage = response.usage
    print(f"Cache read: {getattr(usage, 'cache_read_input_tokens', 0)
    }")
    return response.content[0].text
```

A 5,000-token system prompt making 100 calls/hour saves ~88% on cached tokens—from $264/month to $31/month with Sonnet.

Tip

Prompt caching is especially valuable for data engineering because schema context is large and stable. A 50-table schema (7,000–10,000 tokens) saves significantly with dozens of queries per hour.

Model Selection Optimization

Table 5: Model selection guide.

Task	Haiku	Sonnet	Opus
Simple SQL, validation, docs, log parsing	✓		
Complex SQL, dbt, code review, migration		✓	
Architecture, complex debug, novel design			✓

Note

Start with Haiku for every new workload. Step up to Sonnet if quality is insufficient. Reserve Opus for tasks where Sonnet demonstrably falls short. This is the "try the free sample before buying the bottle" principle of AI cost management.

Batch API for Bulk Processing

The ***Batch API*** processes requests within 24 hours at 50% discount. Ideal for: bulk documentation, warehouse-wide quality analysis, migration code review, and dataset enrichment.

Cost Monitoring

Code Example

Cost Tracking Module

```
import sqlite3
from datetime import datetime, timedelta
from dataclasses import dataclass

@dataclass
class APICallRecord:
    timestamp: str
    model: str
    workload: str
```

```
    input_tokens: int
    output_tokens: int
    cost_usd: float

class CostTracker:
    PRICING = {
        "claude-haiku-4-20250514": {"input": 0.25, "output": 1.25},
        "claude-sonnet-4-20250514": {"input": 3.00, "output": 15.00},
        "claude-opus-4-20250514": {"input": 15.00, "output": 75.00},
    }

    def __init__(self, db_path="cost_tracking.db"):
        self.conn = sqlite3.connect(db_path)
        self.conn.execute("""
            CREATE TABLE IF NOT EXISTS api_calls (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp TEXT, model TEXT, workload TEXT,
                input_tokens INTEGER, output_tokens INTEGER,
                cost_usd REAL
            )""")

    def calculate_cost(self, model, input_tokens, output_tokens):
        p = self.PRICING.get(model, {"input": 3.0, "output": 15.0})
        return round(
            (input_tokens / 1e6) * p["input"]
            + (output_tokens / 1e6) * p["output"], 8
        )

    def record(self, model, workload, usage):
        cost = self.calculate_cost(
            model, usage.input_tokens, usage.output_tokens
        )
        self.conn.execute(
            "INSERT INTO api_calls VALUES (NULL,?,?,?,?,?,?)",
            (datetime.utcnow().isoformat(), model, workload,
             usage.input_tokens, usage.output_tokens, cost)
        )
        self.conn.commit()

    def daily_summary(self, days=30):
        cutoff = (datetime.utcnow() - timedelta(days=days)).isoformat
    ()
        return self.conn.execute("""
            SELECT DATE(timestamp), model, COUNT(*),
```

```
                    SUM(cost_usd) FROM api_calls
                WHERE timestamp >= ? GROUP BY DATE(timestamp), model
                ORDER BY 1 DESC""", (cutoff,)).fetchall()
```

Budget Allocation

Table 6: Example monthly budget for a 10-person team.

Category	Model	Calls/mo	Cost/mo
Ad hoc SQL	Sonnet	3,000	$315
Quality checks	Haiku	12,000	$66
Code review (CI/CD)	Sonnet	2,000	$240
Complex analysis	Opus	300	$360
Buffer (16%)	Mixed	—	$237
Total			**$1,440**

Tip

Always include a 15–20% buffer. Data engineering workloads are spiky—incidents, migrations, and new source onboarding cause temporary increases. "We didn't budget for the production fire" is not a line item anyone wants to explain.

ROI Calculation Framework

Code Example

ROI Calculation

```python
def calculate_roi(
    engineer_hourly_rate: float,
    tasks_per_month: int,
    avg_manual_hours: float,
    avg_assisted_hours: float,
    monthly_api_cost: float,
```

```
) -> dict:
    manual_cost = tasks_per_month * avg_manual_hours *
    engineer_hourly_rate
    assisted_cost = (
        tasks_per_month * avg_assisted_hours * engineer_hourly_rate
        + monthly_api_cost
    )
    savings = manual_cost - assisted_cost
    hours_saved = tasks_per_month * (avg_manual_hours -
    avg_assisted_hours)
    return {
        "monthly_savings": round(savings, 2),
        "hours_saved": round(hours_saved, 1),
        "roi_pct": round((savings / monthly_api_cost) * 100, 1),
    }

# Example: SQL generation workload
result = calculate_roi(
    engineer_hourly_rate=95.0, tasks_per_month=150,
    avg_manual_hours=0.5, avg_assisted_hours=0.15,
    monthly_api_cost=315.0,
)
# savings: $4,672, hours saved: 52.5, ROI: 1483%
```

Beyond time savings, consider: quality improvement (fewer production bugs), knowledge democratization (junior engineers producing senior-level work), faster onboarding, and reduced context switching. Also consider the intangible value of not being woken up at 3 AM because Claude Code handled the incident while you were sleeping. You cannot put a price on sleep. Actually, you can—it's roughly $0.50 per incident resolution.

Cost Per Task Reference

Note

Costs assume no prompt caching. With caching and stable system prompts, input costs for repeated workloads drop by up to 90%.

Table 7: Cost per task (Sonnet, typical token volumes).

Use Case	In Tok	Out Tok	Cost/Call	Cost/1000
Simple SQL query	800	400	$0.008	$8.40
Complex SQL (CTEs)	3,000	1,500	$0.032	$31.50
Code review	5,000	2,000	$0.045	$45.00
dbt model generation	2,500	1,800	$0.035	$34.50
Data quality suite	2,000	3,000	$0.051	$51.00
Migration script	4,000	3,500	$0.065	$64.50
Terraform module	3,000	4,000	$0.069	$69.00

Tips for Reducing Token Usage

1. **Concise system prompts**: Every token is charged on every call. Cut verbose instructions.
2. **Relevant schema only**: Do not paste 50 tables when only 3 are needed.
3. **Structured output**: Request JSON/SQL code blocks rather than prose (40–60% shorter).
4. **Appropriate `max_tokens`**: 512 for simple queries, 2048 for complex generation.
5. **Stop sequences**: Halt generation at predictable boundaries.
6. **Compressed schema**: Use shorthand notation (~40 tokens/table vs. ~160 for full DDL).
7. **Batch related questions**: One call for 5 tables beats 5 separate calls.
8. **Cache aggressively**: Any constant text across calls should be cached.

Note

Cost optimization is ongoing. Like flossing, everyone agrees it should be done regularly, and almost nobody does it until something hurts. Set a monthly reminder to review your cost dashboard, identify trends, and adjust model routing and prompts. Small per-call savings compound at scale.

Glossary

"Every glossary is just a graveyard of terms the author had to explain more than twice."

This glossary defines key terms used throughout this book, organized alphabetically. Some definitions are more serious than others. All are technically accurate. We think.

Agentic AI
An AI system that autonomously plans, uses tools, and takes multi-step actions to accomplish a goal, as opposed to single-shot Q&A. Essentially, an AI that has learned to procrastinate less than you.

Airflow
Apache Airflow. Open-source workflow orchestration platform for scheduling and monitoring data pipelines, defined as DAGs in Python.

API (Application Programming Interface)
Defined protocols and endpoints for software communication. In this book, usually the Anthropic Messages API.

Backfill
Retroactively processing historical data through a pipeline after a bug fix or new transformation. The data engineering equivalent of "we need to redo all of this." Always takes longer than estimated.

Batch API
An Anthropic API feature that processes requests asynchronously within 24 hours at 50% discount.

Batch Processing
Processing data in scheduled chunks (hourly, daily) rather than continuously. Contrast with *Stream Processing*.

CDC (Change Data Capture)
Technique for capturing row-level changes from source databases via log reading (Debezium) or timestamp polling.

CI/CD (Continuous Integration / Continuous Deployment)
Practices automating building, testing, and deploying code—including pipeline code, dbt models, and schema migrations.

Claude Code
Anthropic's official CLI for using Claude Code in software development workflows, including code generation, review, and debugging.

Columnar Storage
Storage format storing column values contiguously, optimized for analytical queries. Examples: Parquet, ORC.

Constitutional AI (CAI)
Anthropic's AI alignment approach where a model follows a set of principles guiding its behavior.

Context Window
Maximum tokens (input plus output) a model processes per request. Claude Code supports up to 200K tokens. Still smaller than your average Jira ticket's comment thread.

CTE (Common Table Expression)
A temporary named result set in SQL defined with `WITH`. Preferred pattern for complex transformations.

DAG (Directed Acyclic Graph)
Graph with directed edges and no cycles. Used in orchestrators for task dependencies and in dbt for lineage. The "acyclic" part is aspirational in some codebases.

Data Contract
Formal agreement between producer and consumers specifying schema, quality expectations, SLAs, and change procedures.

Data Governance
Policies, processes, and standards ensuring data is managed consistently, securely, and in regulatory compliance.

Data Lakehouse
Architecture combining data lake flexibility with warehouse management features (ACID, schema enforcement). Examples: Delta Lake, Iceberg, Hudi.

Data Lineage
Tracing data from origin through all transformations to final destination. Critical for debugging and compliance. Also critical for answering "where did this number come from?" in the board meeting.

Data Observability
Monitoring and understanding data pipeline health in real time—freshness, volume, schema, distribution, lineage.

dbt (data build tool)
Open-source transformation framework using SQL SELECT statements with built-in testing, documentation, and lineage.

Delta Lake
Open-source storage layer bringing ACID transactions and time travel to data lakes. Commonly used with Spark/Databricks.

Dimensional Modeling
Data modeling technique organizing data into fact tables (measurements) and dimension tables (context). Popularized by Claude Code's training data, originally by Kimball.

ELT (Extract, Load, Transform)
Pattern where raw data is loaded into a target system first, then transformed in place. Contrast with *ETL*.

Embeddings
Dense vector representations capturing semantic meaning, enabling similarity search and RAG.

Extended Thinking
A Claude Code feature allowing additional internal reasoning steps before generating a response.

Few-Shot Prompting
Including examples of desired input-output behavior in the prompt to guide the model.

Hallucination
When an AI model generates plausible but factually incorrect information—invented column names, wrong SQL syntax, fictional functions. Like a confident intern on their first day, but with better grammar.

Iceberg
Apache Iceberg. Open table format supporting ACID transactions, schema evolution, and time travel for large analytic datasets.

Idempotency
Property where executing an operation multiple times produces the same result as once. Critical for pipeline retries. If only meetings had this property.

IaC (Infrastructure as Code)
Managing infrastructure through configuration files. Tools: Terraform, Pulumi, CloudFormation.

Kafka
Apache Kafka. Distributed event streaming platform for real-time data pipelines with durable pub-sub messaging.

Human
A biological entity occasionally consulted for coffee preferences and final sign-off on production deployments. Requires sleep, food, and encouragement. See also: *Legacy System*.

Large Language Model (LLM)
Deep learning model trained on vast text, capable of understanding and generating language. Claude Code is an LLM. Not to be confused with "Large Language Mistake," which is what happens when you skip the review step.

Legacy Code
Code written by a human. Identifiable by its bugs, inconsistent variable naming, and comments that say `// TODO: fix this later` from 2017.

Materialization
In dbt, strategy for persisting results: `view`, `table`, `incremental`, or `ephemeral`.

MCP (Model Context Protocol)
Open protocol by Anthropic standardizing how AI models connect to external tools, data sources, and services.

Medallion Architecture
Data lake/lakehouse pattern: Bronze (raw), Silver (cleaned), Gold (business aggregations).

Model Routing
Directing tasks to different model tiers (Haiku, Sonnet, Opus) based on complexity, latency, or cost.

Parquet
: Apache Parquet. Columnar file format optimized for analytics, the de facto standard for data lake storage.

PII (Personally Identifiable Information)
: Data identifying a specific individual (name, email, SSN). Must be handled per privacy regulations.

Pipeline
: Sequence of processing steps moving data from sources through transformations to a destination. Will break at 3 AM. Always at 3 AM.

Prompt Caching
: API feature caching repeated prompt prefixes. Cached tokens charged at 10% of normal input rate.

Prompt Engineering
: Crafting effective prompts: system prompt design, few-shot examples, constraints, and output formatting. The art of asking a computer to do something in exactly the right way, which is somehow now a six-figure job.

RAG (Retrieval-Augmented Generation)
: Combining information retrieval with generation—retrieved documents ground the model's response in facts.

Rate Limiting
: Restricting API calls per time period. Anthropic enforces limits on requests/min and tokens/min.

SCD (Slowly Changing Dimension)
: Techniques for handling dimension changes over time. Type 1: overwrite. Type 2: new rows with date ranges. Type 3: add a column. Type 4: deep existential regret about schema design choices.

Sleep
: A biological process required by humans but not by Claude Code. Historically interrupted by pipeline failures. See *Pipeline*.

Schema Evolution
: Modifying schemas over time while maintaining compatibility with existing data and consumers.

Snowflake
: Cloud-native data warehouse separating storage and compute, supporting multi-cloud deployment.

Spark
Apache Spark. Distributed computing framework for batch/stream processing, SQL, and ML.

Stream Processing
Processing data continuously as it arrives. Tools: Kafka Streams, Flink, Spark Structured Streaming.

System Prompt
Special prompt setting model behavior and constraints for an entire conversation, passed separately from user messages.

Technical Debt
The accumulated cost of shortcuts taken during development. Like real debt, except the interest rate is measured in on-call pages.

Temperature
Parameter controlling output randomness. Lower (0.0) = deterministic; higher (1.0) = creative. For data engineering, keep it low unless you enjoy surprises in your SQL. You do not enjoy surprises in your SQL.

Terraform
Open-source IaC tool using HCL to provision cloud infrastructure across providers.

Token
Fundamental text processing unit in LLMs, typically 3–4 characters. Billing is based on token counts. The modern equivalent of "the meter is running."

Tool Use (Function Calling)
Claude Code capability to invoke external functions during conversations via structured tool call requests.

Transformer
Neural network architecture based on attention mechanisms. Foundation of all modern LLMs including Claude Code.

Upsert
Insert-or-update operation based on key match. Implemented via `MERGE` in most SQL dialects.

Vector Database
Database optimized for storing and querying high-dimensional embeddings. Examples: Pinecone, Weaviate, pgvector.

Watermark
: In stream processing, a timestamp indicating all data up to that point has been received.

Window Function
: SQL function calculating across related rows without collapsing them. Defined with `OVER()`. Examples: `ROW_NUMBER()`, `LAG()`.

YAML (YAML Ain't Markup Language)
: Human-readable serialization format for configuration. Used by dbt, Kubernetes, GitHub Actions, and data contracts. "Human-readable" is doing a lot of heavy lifting in that definition, especially when indentation is off by one space.

Z-Order
: Multi-dimensional clustering in Delta Lake/Databricks to co-locate related data, improving multi-column filter performance.

Zero-Shot Prompting
: Giving a task description without examples, relying on pre-trained knowledge. Contrast with *Few-Shot Prompting*. The AI equivalent of "just figure it out."