CLAUDE COWORK

Data Plugin Handbook

The Complete Guide for Data Analysts, BI Developers, and Analytics Managers

VERSION 1.0.0

PLUGIN data

RationalEyes.ai

contact@rationaleyes.ai

February 2026

Table of Contents

The Complete Guide to SQL, Analytics, and Data-Driven Decision Making with Claude Cowork

Version 1.0.0 | For data analysts, BI teams, and analytics managers

Part I: Getting Started

What the Data Plugin Does

The Data plugin transforms Claude into a collaborative data analyst. It enables end-to-end analytical workflows: writing SQL queries, exploring datasets, generating visualizations, building interactive dashboards, and validating analyses before sharing with stakeholders.

Core capabilities:

- **Query your data warehouse** directly through MCP connections to Snowflake, BigQuery, Databricks, PostgreSQL, and other SQL databases
- **Write optimized SQL** for any dialect with dialect-specific best practices
- **Profile and explore datasets** to understand structure, quality, and patterns
- **Generate publication-quality visualizations** using Python (matplotlib, seaborn, plotly)
- **Build interactive HTML dashboards** with filters, charts, and tables that work without a server
- **Validate analyses** for methodology errors, calculation mistakes, and bias before sharing

Two modes of operation:

1. **Connected mode** (recommended): With a data warehouse MCP server connected, Claude queries your data directly, iterates on queries based on results, and runs complete analyses end-to-end
2. **Disconnected mode**: Without a warehouse connection, paste SQL results or upload CSV/Excel files. Claude writes queries for you to run manually, then analyzes the results you provide

Who Should Use This Plugin

ROLE	PRIMARY USE CASES
Data Analyst	Ad-hoc queries, exploratory analysis, dashboard creation, stakeholder reporting
BI Developer	SQL optimization, data quality checks, metric definitions, dashboard prototyping
Analytics Manager	QA team analyses, standardize reporting methods, document data assets
Product Manager	Self-service metric queries, trend analysis, experiment result interpretation

ROLE	PRIMARY USE CASES
Business Analyst	Dataset exploration, insight generation, presentation-ready visualizations
Data Engineer	Query debugging, schema documentation, data profiling for pipeline validation

Installation

Cowork (recommended): Open **Plugin Settings** in the Cowork desktop app, find **Data**, and click **Install**. The plugin activates immediately — no CLI required.

Claude Code CLI (alternative): If you are using Claude Code in the terminal rather than Cowork, install via:

```
claude plugins add knowledge-work-plugins/data
```

> **Note:** *All standard Cowork plugins, including Data, are available from Plugin Settings with a single click. The CLI command above is only needed for Claude Code terminal users.*

The plugin installs with six commands and seven skills. Pre-configured MCP servers for BigQuery, Hex, Amplitude, and Atlassian are included in `.mcp.json` but require authentication on first use.

Quick Start

Scenario 1: You have a data warehouse connected

```
You: /analyze How many new users signed up in the last 30 days, broken down by signup

Claude: [Explores schema to find users table]
       → [Writes SQL query with date filter and grouping]
       → [Executes query against your warehouse]
       → [Returns results with a bar chart visualization]

Results: 2,847 new signups in last 30 days
- Organic search: 1,203 (42.3%)
- Paid social: 891 (31.3%)
- Direct: 523 (18.4%)
- Referral: 230 (8.1%)
```

Scenario 2: No warehouse connected

```
You: /write-query I need a retention cohort analysis. Group users by signup month,
     show what percentage are still active 1, 3, 6, 12 months later. Snowflake syntax.

Claude: [Generates SQL with CTEs, window functions, and Snowflake date functions]
       → [Explains each CTE's purpose]
       → [Provides modification suggestions for different time ranges]

You can copy the query, run it in your warehouse, then paste results back for visualiz
```

Part II: The Six Commands

Commands are actions you invoke with a slash. Each is designed for a specific stage of the analysis lifecycle.

/analyze — Answer Data Questions

Purpose: Answer a data question ranging from a quick metric lookup to a formal report.

Usage:

```
/analyze <natural language question>
```

When to use:

- Quick factual lookups: "How many orders were placed yesterday?"
- Exploratory questions: "What's driving the increase in support tickets?"
- Formal reports: "Prepare a quarterly business review of subscription metrics"

How it works:

1. **Parses the question** to determine complexity (quick answer, full analysis, or formal report)
2. **Gathers data** by querying the warehouse or asking you to provide data
3. **Analyzes** with relevant aggregations, comparisons, and pattern detection
4. **Validates** results with sanity checks (row counts, nulls, magnitude, trend continuity)
5. **Presents findings** with the appropriate level of detail and supporting evidence

Examples:

Quick answer:

```
/analyze What was our total revenue in Q4 2024?
```

Full analysis:

```
/analyze Why did conversion rate drop 5% between August and September?
         Break down by channel and device type.
```

Formal report:

```
/analyze Prepare a data quality assessment of our customer table covering
         completeness, consistency, accuracy, and timeliness.
```

Behind the scenes:

For connected warehouses, `/analyze` uses the `sql-queries` skill to write dialect-specific SQL, the `statistical-analysis` skill to identify trends and outliers, the `data-visualization` skill to select appropriate chart types, and the `data-validation` skill to sanity-check results before presenting them.

/explore-data — Profile and Explore a Dataset

Purpose: Generate a comprehensive data profile showing structure, quality, and patterns before diving into analysis.

Usage:

```
/explore-data <table_name or file>
```

When to use:

- First time working with a new table or dataset
- Data quality assessment before building a report
- Understanding column distributions and cardinality
- Discovering which dimensions are worth analyzing

What you get:

Table-level metrics:

- Row count, column count, date range coverage
- Column type breakdown (dimensions, metrics, dates, IDs)

Column-level metrics:

- Null counts and rates
- Distinct counts and cardinality ratios
- For numeric columns: min, max, mean, median, percentiles (p25, p50, p75, p95, p99)
- For string columns: length statistics, most/least common values

- For date columns: date range, distribution by period, gap detection

Data quality flags:

- High null rates (>5% warning, >20% alert)
- Suspicious values (negatives where unexpected, placeholder values, future dates)
- Cardinality surprises (high-cardinality fields that should be categorical)
- Duplicate detection on natural keys

Recommendations:

- Best dimension columns for slicing (3-50 distinct values)
- Key metric columns for measurement
- Suggested follow-up analyses

Example:

```
/explore-data users

→ Data Profile: users

Overview
- Rows: 2,340,891
- Columns: 23 (8 dimensions, 6 metrics, 4 dates, 5 IDs)
- Date range: 2021-03-15 to 2024-01-22

Column Details
[Summary table with nulls, cardinality, distributions]

Data Quality Issues
⚠️ email: 0.2% nulls (4,682 records)
🔴 status: Unexpected value "UNKNOWN" in 340 rows (0.01%)
⚠️ last_login_at: 23% nulls (538,405 records) — expected for dormant users

Recommended Explorations
1. Trend analysis: signups by created_at grouped by signup_source
2. Retention analysis: cohort by signup month, activity by last_login_at
3. Segmentation: compare metrics across plan_type dimension
```

Pro tip: For very large tables (100M+ rows), profiling uses sampling by default. Mention if exact counts are needed.

/write-query — Write Optimized SQL

Purpose: Generate SQL from a natural language description, optimized for your specific dialect.

Usage:

```
/write-query <description of what data you need>
```

When to use:

- Translating requirements into SQL
- Learning SQL patterns for complex operations (window functions, CTEs, pivots)
- Optimizing slow queries
- Translating queries between dialects

Supported dialects:

- PostgreSQL (including Aurora, RDS, Supabase, Neon)
- Snowflake
- BigQuery (Google Cloud)
- Redshift (Amazon)
- Databricks SQL
- MySQL (including Aurora MySQL, PlanetScale)
- SQL Server (Microsoft)
- DuckDB, SQLite

What you get:

1. **Complete query** with syntax highlighting
2. **CTE structure** when queries have multiple logical steps
3. **Inline comments** explaining non-obvious logic
4. **Performance notes** about partition usage, indexing, or potential bottlenecks
5. **Modification suggestions** for common variations (different time ranges, granularity)

Example:

```
/write-query Cohort retention analysis. Group users by signup month, show what
             percentage are still active (had at least one event) at 1, 3, 6,
             and 12 months after signup. BigQuery syntax.
```

Returns:

```
-- Cohort Retention Analysis
-- Users grouped by signup month with retention at 1, 3, 6, 12 months

WITH cohorts AS (
  -- Define cohort membership based on first activity
  SELECT
    user_id,
    DATE_TRUNC(first_activity_date, MONTH) as cohort_month
  FROM users
),

activity AS (
  -- Capture all user activity months
  SELECT
    user_id,
    DATE_TRUNC(activity_date, MONTH) as activity_month
  FROM user_activity
)

SELECT
  c.cohort_month,
  COUNT(DISTINCT c.user_id) as cohort_size,

  -- Month 0 (signup month)
  COUNT(DISTINCT CASE
    WHEN a.activity_month = c.cohort_month THEN a.user_id
  END) as month_0,

  -- Month 1
  COUNT(DISTINCT CASE
    WHEN a.activity_month = DATE_ADD(c.cohort_month, INTERVAL 1 MONTH)
    THEN a.user_id
  END) as month_1,

  -- Month 3
  COUNT(DISTINCT CASE
    WHEN a.activity_month = DATE_ADD(c.cohort_month, INTERVAL 3 MONTH)
    THEN a.user_id
  END) as month_3,

  -- Month 6
  COUNT(DISTINCT CASE
    WHEN a.activity_month = DATE_ADD(c.cohort_month, INTERVAL 6 MONTH)
    THEN a.user_id
  END) as month_6,

  -- Month 12
  COUNT(DISTINCT CASE
    WHEN a.activity_month = DATE_ADD(c.cohort_month, INTERVAL 12 MONTH)
```

```
    THEN a.user_id
  END) as month_12

FROM cohorts c
LEFT JOIN activity a ON c.user_id = a.user_id
GROUP BY c.cohort_month
ORDER BY c.cohort_month;
```

Performance note: This query uses DATE_TRUNC for partition-aware filtering in BigQuery. Ensure user_activity table is partitioned by activity_date for optimal performance.

Best practices applied:

- CTEs for readability (one per logical step)
- Descriptive CTE names (cohorts, activity)
- LEFT JOIN to preserve cohorts with zero retention
- Comments explaining each section
- No SELECT * — only needed columns
- Partition-friendly date functions

/create-viz — Create Visualizations

Purpose: Generate publication-quality charts from data using Python.

Usage:

```
/create-viz <data source> [chart type] [additional instructions]
```

When to use:

- Communicating trends, comparisons, or distributions visually
- Creating charts for presentations or reports
- Exploring data patterns interactively (with plotly)
- Generating standardized chart templates

Data sources:

- Query results from connected warehouse
- Pasted data (tables, CSV format)
- Uploaded CSV or Excel files
- Data from previous analysis in the conversation

Chart type selection:

If you don't specify a chart type, Claude recommends one based on the data relationship:

DATA RELATIONSHIP	RECOMMENDED CHART
Trend over time	Line chart
Comparison across categories	Bar chart (horizontal for many categories)
Part-to-whole composition	Stacked bar or area chart
Distribution of values	Histogram or box plot
Correlation between variables	Scatter plot
Ranking	Horizontal bar chart
Geographic patterns	Choropleth map
Matrix of relationships	Heatmap

What you get:

- Python code (matplotlib/seaborn or plotly)
- Saved PNG file (default) or interactive HTML (if plotly)
- Professional styling (colorblind-friendly palettes, readable fonts, clean layout)
- Proper number formatting (currency, percentages, large numbers)

Example:

```
/create-viz Show monthly revenue for the last 12 months as a line chart
            with trend highlighted

→ [Queries revenue data or asks for it]
→ [Generates Python code with matplotlib]
→ [Saves monthly_revenue_trend.png]
→ [Displays chart]
```

The code includes:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(df['month'], df['revenue'], linewidth=2, color='#4C72B0',
        label='Monthly Revenue')

# Format y-axis as currency
ax.yaxis.set_major_formatter(mticker.FuncFormatter(
    lambda x, p: f'${x/1e6:.1f}M' if x ⩾ 1e6 else f'${x/1e3:.1f}K'
))

ax.set_title('Revenue Grew 23% YoY', fontweight='bold', fontsize=14)
ax.set_xlabel('Month')
ax.set_ylabel('Revenue')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.tight_layout()
plt.savefig('monthly_revenue_trend.png', dpi=150, bbox_inches='tight')
```

Design principles applied:

- Descriptive title stating the insight ("Revenue grew 23% YoY" not "Revenue by Month")
- Color used purposefully (accent color for key series, grey for reference)
- Readable number formatting ($1.2M, not 1200000)
- Minimal chart junk (no unnecessary gridlines or borders)
- Bar charts always start at zero
- Colorblind-friendly palette

For interactive charts: Add "interactive" to your request and Claude uses plotly instead, generating an HTML file you can open in a browser with hover tooltips and zoom.

/build-dashboard — Build Interactive Dashboards

Purpose: Create self-contained HTML dashboards with charts, filters, and tables that work in any browser without a server.

Usage:

```
/build-dashboard <description of dashboard> [data source]
```

When to use:

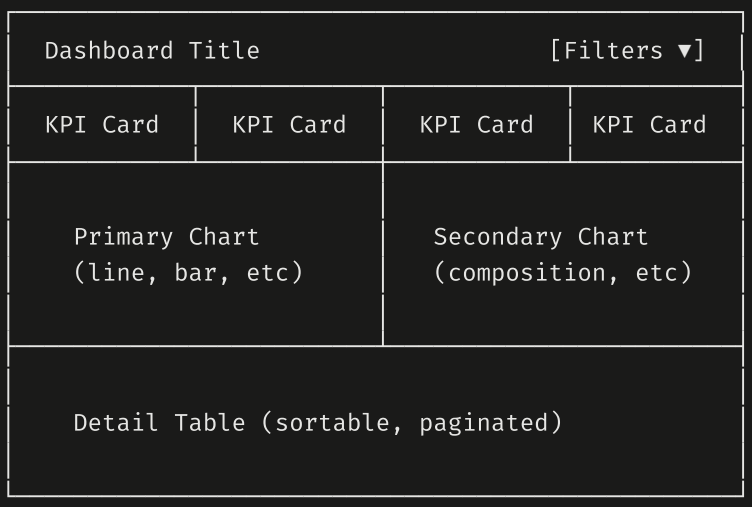
- Executive reporting (high-level KPIs with drill-down details)
- Operational monitoring (team metrics, funnel performance)
- Stakeholder sharing (send a single HTML file via email, no login required)
- Quick dashboard prototypes before building in a BI tool

What you get:

A single HTML file containing:

- **KPI cards** at the top showing headline metrics with period-over-period changes
- **Interactive charts** (line, bar, doughnut, stacked area) using Chart.js
- **Dropdown filters** that update all visualizations simultaneously
- **Sortable data table** for detailed drill-down
- **Professional styling** (clean, responsive, print-friendly)
- **Embedded data** (no external dependencies, works offline)

Dashboard layout pattern:



Example:

```
/build-dashboard Monthly sales dashboard with revenue trend, top 10 products,
                 and regional breakdown. Query the orders table for last 90 days.
```

Returns `sales_dashboard.html` containing:

KPI Cards:

- Total Revenue: $847K (+12% vs prior period)
- Orders: 2,341 (+8%)
- Average Order Value: $362 (+4%)
- Conversion Rate: 3.2% (+0.3pp)

Charts:

- Line chart: Daily revenue trend (90 days)
- Horizontal bar: Top 10 products by revenue
- Doughnut: Revenue by region

Filters:

- Date range picker
- Region dropdown
- Product category dropdown

Table:

- All orders with: Date, Product, Region, Amount, Status
- Sortable by any column
- Shows top 100, with "showing X of Y" pagination

Technical details:

The generated HTML:

- Uses Chart.js 4.5.1 from CDN (integrity hash verified)
- Includes all data as embedded JSON (no external data fetches)
- Responsive CSS Grid layout (works on desktop, tablet, mobile)
- Print-friendly styles (filters hidden, charts preserved)
- No server required — works by double-clicking the file

Dashboard data size guidelines:

DATA SIZE	APPROACH
<1,000 rows	Embed directly in HTML
1,000-10,000 rows	Embed, may pre-aggregate for charts

DATA SIZE	APPROACH
10,000-100,000 rows	Pre-aggregate server-side before embedding
>100,000 rows	Not suitable for client-side dashboard. Use a BI tool

Customization:

Request specific styling:

- "Dark mode dashboard" for presentations
- "Print-optimized dashboard" for PDF export
- "Brand colors: #FF5733 and #33A1FF" for company palette

/validate — Validate Analysis Before Sharing

Purpose: QA an analysis for accuracy, methodology, and bias before sharing with stakeholders.

Usage:

```
/validate <analysis to review>
```

The analysis can be:

- A document or report
- SQL queries and results
- Charts and underlying data
- A description of methodology and findings

When to use:

- Before any high-stakes presentation or decision
- After completing an analysis, before sending to stakeholders
- When results seem surprising and you want a sanity check
- To train new analysts on common pitfalls

What gets checked:

Methodology review:

- Is the question framed correctly?
- Are the right tables and time ranges used?

- Is the analysis population correctly defined?
- Are metric definitions clear and consistent?
- Is the comparison fair (apples to apples)?

Common analytical errors:

- Missing data that could skew results
- Survivorship bias (only looking at entities that "survived")
- Simpson's paradox (aggregated trend reverses when segmented)
- Double-counting from bad joins (many-to-many explosions)
- Incorrect denominators in rate calculations
- Seasonality not accounted for
- Look-ahead bias (using future data to explain past)

Calculation spot-checks:

- Recalculate key numbers independently
- Verify subtotals sum to totals
- Check that percentages sum to ~100%
- Validate YoY/MoM comparison periods

Visualization review:

- Do axes start appropriately (zero for bars)?
- Are scales consistent across comparison charts?
- Could the chart mislead a quick reader?

Narrative evaluation:

- Are conclusions supported by the data?
- Are alternative explanations acknowledged?
- Is uncertainty communicated appropriately?

Output:

A validation report with three-level assessment:

Ready to share — Analysis is sound, calculations verified, caveats noted. Minor suggestions but nothing blocking.

Share with noted caveats — Largely correct but has specific limitations that must be communicated. Lists required caveats.

Needs revision — Specific errors, methodology issues, or missing analyses must be addressed. Lists required changes with priority.

Example:

```
/validate Review this churn analysis before I present to the exec team:
          [pastes analysis]

→ Validation Report

Overall Assessment: Share with noted caveats

Methodology Review
✓ Time range appropriate (full quarters, no partial months)
✓ Churn definition matches finance team standard
⚠ Analysis excludes trial users — note this in presentation

Issues Found
1. [Medium] Denominator calculation: Using beginning-of-period customers,
   but some churned customers signed up mid-period. This slightly
   overstates churn rate (estimated 0.2-0.3pp impact).

2. [Low] Q4 includes holiday seasonality — YoY comparison better than QoQ.

Calculation Spot-Checks
✓ Total churned customers: 147 (verified)
✓ Churn rate: 4.7% (verified with noted caveat)
✓ Segmentation: Enterprise 2.1%, SMB 6.8% (subtotals sum correctly)

Required Caveats for Stakeholders
- "Churn rate excludes trial users, includes only paying customers"
- "Q4 seasonality may affect QoQ comparison; YoY shows clearer trend"
- "Denominator uses beginning-of-period customers; methodology note attached"

Suggested Improvements
1. Add cohort view: churn by tenure (0-3mo, 3-6mo, 6-12mo, 12mo+)
2. Calculate both beginning-of-period and average-customer churn rates
3. Show churn $ value alongside customer count
```

Pro tip: Run /validate on your own work, even for quick analyses. It takes a minute and catches errors that could damage your credibility.

Part III: The Seven Skills

Skills are domain knowledge that Claude loads automatically when the conversation touches relevant topics. You don't invoke them explicitly — they activate in the background based on what you're discussing.

sql-queries — SQL Best Practices Across Dialects

Triggers: Writing queries, optimizing SQL, translating between dialects, debugging query errors

What it provides:

Dialect-specific reference for date/time functions, string operations, JSON handling, and window functions across:

- PostgreSQL, Snowflake, BigQuery, Redshift, Databricks, MySQL, SQL Server, DuckDB, SQLite

Common SQL patterns:

- Window functions (ranking, running totals, moving averages, lag/lead)
- CTEs for readability
- Cohort retention analysis
- Funnel analysis
- Deduplication

Performance optimization:

- Partition pruning (filter on partition keys early)
- Index usage vs table scans
- EXISTS vs IN for subqueries
- Avoiding correlated subqueries
- Dialect-specific features (Snowflake clustering, BigQuery partitioning, Redshift distribution keys)

Example — dialect differences for date arithmetic:

```
-- PostgreSQL
created_at + INTERVAL '7 days'
DATE_TRUNC('month', created_at)
EXTRACT(DOW FROM created_at)  -- 0=Sunday

-- Snowflake
DATEADD(day, 7, created_at)
DATE_TRUNC('month', created_at)
DAYOFWEEK(created_at)

-- BigQuery
DATE_ADD(created_at, INTERVAL 7 DAY)
DATE_TRUNC(created_at, MONTH)
EXTRACT(DAYOFWEEK FROM created_at)  -- 1=Sunday
```

Error handling and debugging: When a query fails, the skill guides troubleshooting:

1. Syntax errors — check dialect-specific syntax
2. Column not found — verify column names (case sensitivity in PostgreSQL)
3. Type mismatches — explicit casting
4. Division by zero — use NULLIF or safe division
5. Ambiguous columns — qualify with table aliases

---

data-exploration — Data Profiling Methodology

Triggers: Encountering new datasets, assessing data quality, discovering column distributions, identifying nulls and outliers

What it provides:

Three-phase profiling methodology:

Phase 1: Structural understanding

- Table-level questions (row count, grain, primary key, update frequency)
- Column classification (ID, dimension, metric, temporal, text, boolean, structural)

Phase 2: Column-level profiling

- Nulls, distinct counts, cardinality
- For numerics: min, max, mean, median, percentiles, zero/negative counts
- For strings: length stats, common values, pattern analysis
- For dates: range, gaps, future dates

- For booleans: true/false/null distribution

Phase 3: Relationship discovery

- Foreign key candidates
- Hierarchies (country > state > city)
- Correlations between numeric columns
- Derived columns (computed from others)

Quality assessment framework:

Completeness scoring:

- Complete (>99% non-null): Green
- Mostly complete (95-99%): Yellow — investigate nulls
- Incomplete (80-95%): Orange
- Sparse (<80%): Red — may not be usable

Consistency checks:

- Value format inconsistency ("USA", "US", "United States")
- Type inconsistency (numbers stored as strings)
- Referential integrity (foreign keys without matching parent)
- Business rule violations (negative quantities, end dates before start dates)

Accuracy indicators (red flags):

- Placeholder values (0, -1, 999999, "N/A", "test")
- Default values with suspiciously high frequency
- Stale data (no recent updates)
- Impossible values (ages > 150, far-future dates)

Pattern discovery techniques:

Distribution characterization:

- Normal, right-skewed, left-skewed, bimodal, uniform, power law

Temporal patterns:

- Trend, seasonality, day-of-week effects, holiday effects, change points, anomalies

Segmentation discovery:

- Natural segments (categorical columns with 3-20 values)
- Significantly different behavior across segments

data-visualization — Chart Selection and Design

Triggers: Building charts, choosing chart types, creating publication-quality figures, applying design principles

What it provides:

Chart selection guide by data relationship (see /create-viz section above)

When NOT to use certain charts:

- Pie charts (humans bad at comparing angles — use bar charts)
- 3D charts (distort perception, add no information)
- Dual-axis charts (misleading — use cautiously, label clearly)
- Stacked bars with many categories (hard to compare middle segments)

Python code patterns for matplotlib, seaborn, and plotly with:

- Professional styling setup
- Colorblind-friendly palettes
- Number formatting helpers (currency, percentages, large numbers)
- Small multiples for comparing groups
- Interactive charts with plotly

Design principles:

Color:

- Use purposefully (encode data, not decorate)
- Highlight key insight with accent color, grey out reference data
- Sequential gradients for ordered values
- Diverging gradients for data with meaningful center
- Avoid red/green only (8% of men are colorblind)

Typography:

- Title states the insight: "Revenue grew 23% YoY" beats "Revenue by Month"
- Subtitle adds context (date range, filters, data source)
- Readable axis labels (never rotated 90 degrees)
- Data labels on key points only

Layout:

- Reduce chart junk (remove decorative gridlines, borders, backgrounds)
- Sort by value, not alphabetically (unless natural order)
- Appropriate aspect ratio (time series wider than tall)
- White space is good

Accuracy:

- Bar charts start at zero (always)
- Line charts can have non-zero baselines when variation is meaningful
- Consistent scales across comparison panels
- Show uncertainty (error bars, confidence intervals)
- Label axes with units

Accessibility:

- Never rely on color alone (add patterns, line styles, labels)
- Test with colorblind simulator
- Sufficient contrast
- Minimum 10pt labels, 12pt titles
- Works in black and white for printing

---

statistical-analysis — Descriptive Stats and Hypothesis Testing

Triggers: Analyzing distributions, testing for significance, detecting anomalies, computing correlations, interpreting statistical results

What it provides:

Descriptive statistics methodology:

Choosing measures of center:

- Symmetric distribution, no outliers → Mean
- Skewed distribution → Median
- Highly skewed with outliers → Report both mean and median; the gap shows skew

Spread and variability:

- Standard deviation (normally distributed data)
- IQR (skewed data, robust to outliers)
- Coefficient of variation (compare variability across different scales)

Percentiles for business context:

- p1: Floor/minimum typical value
- p25, p50, p75: Quartiles
- p90, p95: High end of normal range
- p99: Extreme users/values

Trend analysis and forecasting:

Moving averages for smoothing:

- 7-day MA (daily data with weekly seasonality)
- 28-day MA (smooths weekly and monthly patterns)

Period-over-period comparison:

- WoW, MoM, YoY, same-day-last-year

Growth rate calculations:

- Simple growth: (current - previous) / previous
- CAGR: (ending / beginning) ^ (1 / years) - 1
- Log growth for volatile series

Simple forecasting methods:

- Naive (tomorrow = today)
- Seasonal naive (tomorrow = same day last week/year)
- Linear trend (for clearly linear trends only)
- Moving average forecast

Always communicate uncertainty: Provide ranges, not point estimates.

Outlier and anomaly detection:

Statistical methods:

- Z-score (for normal distributions): >3 standard deviations
- IQR method (robust): below Q1 - 1.5×IQR or above Q3 + 1.5×IQR
- Percentile method: below p1 or above p99

Handling outliers:

1. Investigate (data error vs genuine extreme vs different population)
2. Fix or remove errors
3. Keep genuine extremes, use robust statistics (median instead of mean)

4. Segment out different populations

Hypothesis testing basics:

Common tests:

- t-test: Compare two group means (normal data)
- z-test: Compare two proportions (conversion rates, binary outcomes)
- Mann-Whitney U: Compare two groups (non-normal data)
- ANOVA: Compare 3+ group means
- Chi-squared: Association between categorical variables

Practical vs statistical significance:

- Statistical significance = unlikely due to chance
- Practical significance = large enough to matter for business

Always report:

- Effect size (how big is the difference?)
- Confidence interval (range of plausible true effects)
- Business impact (translate to revenue, users, etc.)

When to be cautious:

Correlation is not causation — consider:

- Reverse causation (B causes A, not A causes B)
- Confounding variables (C causes both)
- Coincidence

Multiple comparisons problem:

- Testing 20 metrics at p=0.05 means ~1 false positive
- Adjust alpha or report how many tests were run

Simpson's paradox:

- Aggregate trend can reverse when segmented
- Always check key segments

Survivorship bias:

- Can only analyze entities that "survived" to be in dataset
- Ask: "Who is missing and would their inclusion change conclusions?"

---

data-validation — Pre-Delivery QA Checklist

Triggers: Reviewing analyses for errors, checking for survivorship bias, validating aggregation logic, preparing documentation

What it provides:

Comprehensive QA checklist covering:

Data quality:

- Source verification, freshness, completeness, null handling, deduplication, filter verification

Calculation checks:

- Aggregation logic, denominator correctness, date alignment, join correctness, metric definitions, subtotal verification

Reasonableness:

- Magnitude plausibility, trend continuity, cross-reference with known sources, order-of-magnitude checks

Presentation:

- Chart accuracy, number formatting, title clarity, caveat transparency, reproducibility

Common data analysis pitfalls:

Join explosion: Many-to-many joins silently multiply rows. Always check row counts before and after joins.

Survivorship bias: Analyzing only current users misses churned users. Ask "who is NOT in this dataset?"

Incomplete period comparison: Comparing partial period to full period (January MTD vs December full month).

Denominator shifting: The denominator definition changes between periods, making rates incomparable.

Average of averages: Averaging pre-computed averages gives wrong results when group sizes differ. Always aggregate from raw data.

Timezone mismatches: Different sources use different timezones. Standardize to UTC.

Selection bias: Segments defined by the outcome being measured ("users who completed onboarding have higher retention" — they self-selected).

Result sanity checking:

Magnitude checks:

- User counts match known MAU/DAU?
- Revenue in right order of magnitude vs ARR?
- Conversion rates between 0-100%?
- Percentages sum to ~100%?

Cross-validation:

- Calculate same metric two ways, verify they match
- Spot-check individual records manually
- Compare to benchmarks (dashboards, prior analyses)
- Boundary checks (single day, single user)

Red flags:

- > *50% change period-over-period without obvious cause*
- Exact round numbers (suggests filter issue)
- Rates exactly 0% or 100% (incomplete data)
- Results perfectly confirm hypothesis (reality is messier)

Documentation standards:

Every non-trivial analysis should include:

- Question being answered
- Data sources with "as of" dates
- Metric definitions (exactly how calculated)
- Methodology steps
- Assumptions and limitations
- Key findings with evidence
- SQL queries with comments
- Caveats for stakeholders

---

interactive-dashboard-builder — HTML/JS Dashboard Patterns

Triggers: Creating dashboards, building interactive reports, generating shareable HTML files

What it provides:

Complete HTML/JS dashboard templates using Chart.js for visualizations.

Base structure:

- Responsive layout (CSS Grid)
- KPI cards with period-over-period changes
- Chart containers with Chart.js integration
- Sortable data tables
- Dropdown filters that update all visualizations
- Professional styling (clean, colorblind-friendly, print-optimized)
- Embedded data (no server required)

Chart.js integration patterns:

- Line charts (trends)
- Bar charts (comparisons, horizontal for many categories)
- Doughnut charts (composition, <6 categories)
- Stacked charts (composition over time)

Interactivity implementation:

- Filter dropdowns
- Date range pickers
- Sortable tables (click headers)
- Hover tooltips
- Number formatting (commas, currency, percentages)

CSS styling system:

- Color variables (backgrounds, text, data colors, status colors)
- Responsive grid layout
- KPI card styling with change indicators
- Chart container styling
- Professional data table
- Print-friendly styles

Performance considerations:

Data size guidelines:

- <1K rows: Full interactivity
- 1K-10K: Pre-aggregate for charts
- 10K-100K: Server-side aggregation required
- > *100K: Use BI tool instead*

Chart performance:

- Limit line charts to <500 points per series
- Limit bar charts to <50 categories
- Scatter plots capped at 1K points
- Disable animations: `animation: false`
- Use `Chart.update('none')` for filter updates

DOM performance:

- Table pagination (50-100 rows per page)
- `requestAnimationFrame` for coordinated updates

---

data-context-extractor — Generate Company-Specific Data Skills

Triggers: "Create a data context skill", "Set up data analysis for our warehouse", "Generate a data skill for [company]"

What it provides:

A meta-skill that extracts company-specific data knowledge and generates tailored skills.

Two modes:

Bootstrap Mode — Create new skill from scratch:

1. Identify database type (BigQuery, Snowflake, PostgreSQL, Databricks)
2. Explore schema (list tables, identify most-used tables)
3. Ask core questions:
   - Entity disambiguation ("What does 'user' mean here? Are there different types?")
   - Primary identifiers ("What's the main ID? Multiple IDs for same entity?")
   - Key metrics ("Top 2-3 metrics? How calculated?")

- Data hygiene ("What should always be filtered out?")
- Common gotchas ("What mistakes do new analysts make?")

4. Generate skill with structure:
   - `SKILL.md` with entity definitions, metric formulas, best practices
   - `references/` with detailed table documentation
   - `references/entities.md`, `metrics.md`, domain-specific files

Iteration Mode — Improve existing skill:

1. Load existing skill files
2. Identify gap (new domain, missing metrics, undocumented tables)
3. Targeted discovery (explore relevant tables, ask domain questions)
4. Generate new reference file
5. Update skill navigation

Generated skill structure:

```
company-data-analyst/
├── SKILL.md
└── references/
    ├── entities.md          # Entity definitions and relationships
    ├── metrics.md           # KPI calculations
    ├── tables/
    │   ├── product.md       # Product analytics domain
    │   ├── finance.md       # Financial reporting domain
    │   └── marketing.md     # Marketing attribution domain
    └── dashboards.json      # Catalog of existing dashboards
```

Quality standards:

- Entity disambiguation is clear
- Key terminology defined
- Standard filters documented
- 2-3 sample queries per domain
- SQL uses correct dialect
- Reference files linked from SKILL.md

Example use case:

A BI team at Acme Corp wants Claude to understand their Snowflake warehouse:

```
You: Create a data context skill for our Snowflake warehouse

Claude: I'll help you create a custom data analysis skill. Let me connect
        to your warehouse and explore the schema.

        [Connects to Snowflake via MCP]
        [Lists schemas: PROD, ANALYTICS, STAGING]

        Which 3-5 tables do analysts query most often?

You: Most common are ANALYTICS.USERS, ANALYTICS.EVENTS, ANALYTICS.SUBSCRIPTIONS,
     and PROD.TRANSACTIONS

Claude: [Pulls schema for each table]

        When people at Acme say "user", what exactly do they mean?
        Are there different types of users?

You: We distinguish between "users" (individuals with accounts) and "organizations"
     (companies that have subscriptions). One organization can have many users.

Claude: [Continues asking questions about IDs, metrics, filters, gotchas]
        [Generates acme-data-analyst skill with 4 reference files]

        I've created your Acme data skill with:
        - Entity definitions (users vs organizations, relationship mapping)
        - 3 key metrics (MRR, active users, churn rate) with exact formulas
        - 4 table reference files with column details and common queries
        - Standard exclusion filters (is_test = false, is_deleted = false)
        - Common gotchas (timezone handling, subscription state changes)

        Install with: claude plugins add ./acme-data-analyst.plugin
```

The generated skill is immediately usable and improves over time as analysts add more domain knowledge.

Part IV: Connecting Your Data Stack

MCP Servers and Tool Categories

The data plugin uses the placeholder system to remain tool-agnostic. Plugin files reference tools by category, not by specific product:

CATEGORY	PLACEHOLDER	INCLUDED SERVERS	OTHER OPTIONS
Data warehouse	`~data warehouse`	Snowflake*, Databricks*, BigQuery	Redshift, PostgreSQL, MySQL
Notebook	`~notebook`	Hex	Jupyter, Deepnote, Observable
Product analytics	`~product analytics`	Amplitude	Mixpanel, Heap, Google Analytics
Project tracker	`~project tracker`	Atlassian (Jira/Confluence)	Linear, Asana, Monday

*Placeholder — MCP URL not yet configured in the included `.mcp.json`

Connecting a Data Warehouse

Pre-configured (HTTP/SSE servers):

BigQuery, Hex, Amplitude, and Atlassian are pre-configured in `.mcp.json` with their MCP endpoints. On first use, Claude prompts for authentication.

Not yet configured (Snowflake, Databricks):

Edit `.mcp.json` to add connection details:

Snowflake:

```
{
  "mcpServers": {
    "snowflake": {
      "type": "http",
      "url": "https://your-account.snowflakecomputing.com/mcp",
      "headers": {
        "Authorization": "Bearer ${SNOWFLAKE_TOKEN}"
      }
    }
  }
}
```

Set `SNOWFLAKE_TOKEN` environment variable with your access token.

Databricks:

```
{
  "mcpServers": {
    "databricks": {
      "type": "http",
      "url": "https://your-workspace.cloud.databricks.com/mcp",
      "headers": {
        "Authorization": "Bearer ${DATABRICKS_TOKEN}"
      }
    }
  }
}
```

PostgreSQL, Redshift, MySQL (via stdio MCP server):

For local or self-hosted databases, use the appropriate MCP server implementation:

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres",
               "postgresql://user:pass@localhost:5432/dbname"]
    }
  }
}
```

Connecting Analytics and BI Tools

Amplitude (included, requires auth):

```
{
  "mcpServers": {
    "amplitude": {
      "type": "http",
      "url": "https://mcp.amplitude.com/mcp"
    }
  }
}
```

Other product analytics tools:

For tools without official MCP servers, you can:

1. Use REST API via HTTP MCP server
2. Export data to CSV and upload to Claude
3. Query via your data warehouse if the tool syncs there

Connecting Notebooks

Hex (included):

```
{
  "mcpServers": {
    "hex": {
      "type": "http",
      "url": "https://app.hex.tech/mcp"
    }
  }
}
```

Jupyter, Deepnote, Observable:

These don't have native MCP servers but you can:

- Reference local Jupyter notebooks by file path
- Export notebooks to `.ipynb` files for Claude to read
- Use Claude to generate notebook code, then paste into your environment

What Changes When Warehouse is Connected

With connection:

- `/analyze` queries data directly, iterates on results
- `/explore-data` profiles live tables with exact statistics
- `/write-query` can explore schema to find correct table/column names
- End-to-end workflows (question → query → results → visualization)

Without connection:

- `/analyze` asks you to provide data (paste or upload)
- `/explore-data` works on uploaded files only
- `/write-query` writes queries for you to run manually
- You copy queries, run in your tool, paste results back

Both modes work. Connected mode is faster and more iterative.

Part V: Real-World Workflows

Workflow 1: Ad-Hoc Exploratory Analysis

Scenario: Product manager asks "Why did signups drop 15% last week?"

Process:

```
You: /analyze Why did signups drop 15% last week? Break down by source, device,
     and compare to previous 4 weeks.

Claude: [Phase 1: Data gathering]
        → Explores schema, finds users table with signup_date, source, device
        → Writes query for last 5 weeks of signups, grouped by week, source, device
        → Executes query

        [Phase 2: Analysis]
        → Identifies: Mobile web signups down 42% week-over-week
        → Desktop and mobile app signups flat
        → Specific source: organic search on mobile web dropped significantly

        [Phase 3: Validation]
        → Sanity check: Total signups match known dashboard figure
        → Checks for partial week data — last week is complete
        → Verifies no test users included in count

        [Phase 4: Presentation]
        → Line chart: 5-week signup trend by source × device
        → Bar chart: WoW change by segment
        → Key finding: "Mobile web organic signups dropped 42% (from 450 to 260)"
        → Follow-up questions suggested:
          - Was there a Google algorithm update last week?
          - Did mobile site performance degrade?
          - Are there technical errors in mobile signup flow?
```

Result: Root cause identified in 2 minutes. Product team investigates mobile site performance, discovers a JavaScript error introduced in last deployment.

Workflow 2: Building a Recurring Executive Report

Scenario: Finance team needs a monthly revenue dashboard for board meetings.

Process:

```
You: /build-dashboard Monthly revenue dashboard for board presentation.
     KPIs: MRR, new MRR, churned MRR, net new MRR, logo retention.
     Charts: MRR trend (12 months), new vs churned MRR (stacked),
     top 10 accounts by MRR.
     Query subscriptions and transactions tables for last 12 months.

Claude: [Queries data warehouse]
        → Pulls subscription and transaction history
        → Calculates MRR metrics with month-over-month changes
        → Generates revenue_dashboard.html

        [Dashboard includes]
        KPIs (top row):
        - MRR: $847K (+5.2% MoM)
        - New MRR: $89K
        - Churned MRR: $23K
        - Net New MRR: $66K
        - Logo Retention: 96.2%

        Charts:
        - Line chart: MRR trend (12 months) with annotations for major events
        - Stacked bar: New vs Churned MRR by month
        - Horizontal bar: Top 10 accounts

        Filters:
        - Date range picker
        - Segment dropdown (Enterprise / SMB / Startup)

        Table: All accounts with MRR > $10K, sortable

You: Can you make it presentation-mode with larger fonts and high contrast?

Claude: [Updates CSS for presentation]
        → Increases font sizes (KPI values 36px, chart titles 16px)
        → Higher contrast colors
        → Removes gridlines for cleaner look
        → Saves revenue_dashboard_presentation.html

You: Perfect. Can you also generate the SQL queries so our data team can
     automate this monthly?

Claude: [Provides all queries as documented SQL files]
        → mrr_calculation.sql (with CTEs explaining each step)
        → new_churned_mrr.sql
        → top_accounts.sql
        → Includes scheduling notes: "Run on 1st of month for prior month"
```

Result: Board-ready dashboard in 5 minutes. SQL queries handed to data engineering for automation. Next month's report generated automatically.

Workflow 3: Data Quality Assessment

Scenario: Data team launches new customer table and needs a quality audit before analysts start using it.

Process:

```
You: /explore-data ANALYTICS.CUSTOMERS_V2

Claude: [Profiles table]

        Data Profile: ANALYTICS.CUSTOMERS_V2

        Overview
        - Rows: 45,829
        - Columns: 31 (12 dimensions, 8 metrics, 6 dates, 5 IDs)
        - Date range: 2019-01-01 to 2024-02-14

        [Detailed column statistics]

        Data Quality Issues
        🔴 email: 12.3% nulls (5,637 records) — CRITICAL for marketing ops
        🔴 country: Inconsistent values ("US", "USA", "United States",
           "us", "United States of America") — needs standardization
        ⚠️ customer_since: 340 records with future dates (likely timezone issue)
        ⚠️ lifetime_value: 15 records with negative values — data error
        🔴 account_id: 2,341 customers with NULL account_id — orphaned records
        ⚠️ last_activity_at: 38% nulls (17,415 records) — expected for dormant

You: Let me validate that email null issue before we alert the team.
     /analyze What percentage of customers with null email were created
     in the last 30 days vs older?

Claude: [Queries data]
        → Of 5,637 null emails:
          - 4,891 (87%) created before 2022 (legacy data migration)
          - 746 (13%) created in last 30 days — ACTIVE ISSUE

        → Breakdown of recent nulls by signup source:
          - API integration: 512 (68% of recent nulls)
          - Manual import: 234 (31%)

        Finding: Email null rate is 1.6% for recent signups, concentrated
```

```
         in API integration path. Legacy nulls are expected.

You: /validate Review this analysis before I send to engineering

Claude: Validation Report

        Overall Assessment: Ready to share

        ✓ Data source verified (CUSTOMERS_V2, current as of today)
        ✓ Null calculation correct (5,637 / 45,829 = 12.3%)
        ✓ Date range logic correct (last 30 days)
        ✓ Breakdown percentages sum to 100%

        Suggested addition: Include ticket link for API integration team
```

Result: Quality issues identified and prioritized. Legacy nulls flagged as "expected," recent nulls flagged as "active bug in API integration." Engineering ticket created with specific reproduction steps.

Workflow 4: Self-Service Metrics for Non-Analysts

Scenario: Marketing manager (non-technical) needs campaign performance metrics.

Process:

```
Marketing Manager: Can you show me how our paid social campaigns performed
                   last month compared to the previous month?

Claude: [Uses /analyze command automatically]
        → Explores schema, finds campaigns and conversions tables
        → Identifies last month as January 2024, prior month as December 2023
        → Writes query:
          - Campaign-level metrics (impressions, clicks, conversions, spend)
          - Month-over-month comparison
          - Calculated fields (CTR, CPC, conversion rate, ROAS)
        → Executes query

        Results: January 2024 Paid Social Performance

        Summary:
        - Spend: $28,450 (+12% vs Dec)
        - Conversions: 342 (+8% vs Dec)
        - CPA: $83.19 (-4% vs Dec — improving)
        - ROAS: 4.2x (+0.3x vs Dec)

        [Horizontal bar chart: Top 5 campaigns by ROAS]
```

```
        [Line chart: Daily conversion trend]

        Key Insights:
        1. "Product Launch" campaign (Jan 15-31) drove 40% of conversions
           at $67 CPA, 15% below average
        2. "Retargeting" campaign maintained best ROAS (6.1x) with
           smallest spend ($3.2K)
        3. "Brand Awareness" campaign has improving trend but low conversions

Marketing Manager: Can you make this into a dashboard I can share with my team?

Claude: [Uses /build-dashboard]
        → Creates paid_social_performance.html with:
          - KPI cards (spend, conversions, CPA, ROAS)
          - Monthly trend chart
          - Campaign comparison table
          - Filters: date range, campaign type
        → Saves and opens in browser

Marketing Manager: Perfect! Can I get this same report next month automatically?

Claude: I can provide the SQL query for your data team to schedule:

        [Provides parameterized SQL with @month variable]
        [Includes scheduling instructions]

        Alternative: If you want to run this yourself next month, just ask
        me the same question and I'll generate an updated report.
```

Result: Marketing manager gets monthly report without waiting for data team. Report is shareable (single HTML file). Optional automation path provided.

Workflow 5: Query Optimization

Scenario: Analyst has a slow-running query (12 minutes) and needs to optimize it.

Process:

```
You: This query takes 12 minutes in BigQuery. Can you optimize it?
     [Pastes 80-line query with multiple CTEs and joins]

Claude: [Analyzes query structure]

        Performance Issues Identified:

        1. CRITICAL: Missing partition filter
```

```
            → events table is partitioned by event_date but query has no date filter
            → BigQuery scanning full table (500M rows)
            → Fix: Add WHERE event_date ≥ '2024-01-01'

         2. HIGH: Unnecessary SELECT *
            → First CTE selects all columns, only 5 are used downstream
            → Fix: Select only needed columns

         3. MEDIUM: Correlated subquery in WHERE clause
            → Subquery runs for each row
            → Fix: Convert to LEFT JOIN with filtering

         4. LOW: Multiple passes over same table
            → events table accessed in 3 separate CTEs
            → Fix: Combine into single CTE with CASE statements

         Optimized Query:
         [Provides refactored version with:]
         - Partition filter at top CTE: `WHERE event_date ≥ '2024-01-01'`
         - Explicit column selection: removes 15 unused columns
         - Correlated subquery converted to LEFT JOIN
         - Combined CTEs reduce table scans from 3 to 1

         Expected Impact:
         - Scanned data: 500M rows → 8M rows (98% reduction)
         - Estimated runtime: 12 min → 15-30 sec
         - Cost reduction: ~95%

You: [Runs optimized query]
     It's now 18 seconds. Huge improvement!

Claude: Great! A few more tips for this query pattern:

         1. Consider materializing the base events CTE as a table if you
            run this daily — costs amortize over multiple uses

         2. The users table join is on user_id — ensure users table is
            clustered by user_id for best performance

         3. If you need historical trends, don't expand the date range too far
            — better to query month-by-month and union the results
```

Result: Query runtime reduced from 12 minutes to 18 seconds. Analyst learns BigQuery partitioning best practices for future queries.

Part VI: Customization and Extension

Adding Company-Specific Metric Definitions

Your company likely has standard metric calculations that everyone should use consistently. Add these to a custom skill.

Approach: Create a complementary skill in `.claude/skills/`

File: `.claude/skills/acme-metrics/SKILL.md`

```
---
name: acme-metrics
description: >
  Acme Corp standard metric definitions and calculation methods.
  Use when calculating or reporting on ARR, MRR, NRR, CAC, LTV,
  churn rate, or any KPI referenced in board reports or OKRs.
---

# Acme Corp Standard Metrics

## Revenue Metrics

### Annual Recurring Revenue (ARR)
**Definition:** Annualized value of all active subscriptions.

**Formula:**
```sql
SELECT
  SUM(monthly_amount * 12) as arr
FROM subscriptions
WHERE status = 'active'
  AND is_trial = false
  AND is_test = false
```

Caveats:

- Excludes trials (even if paid)
- Includes annual subscriptions at full value
- Snapshot metric (point-in-time, not averaged)
```

Monthly Recurring Revenue (MRR)

Formula:

```
SELECT
  SUM(monthly_amount) as mrr
FROM subscriptions
WHERE status = 'active'
  AND is_trial = false
  AND is_test = false
```

Segments:

- New MRR: From subscriptions created this month
- Expansion MRR: Upgrades within existing accounts
- Churned MRR: From subscriptions cancelled this month
- Net New MRR: New + Expansion - Churned

Net Revenue Retention (NRR)

Definition: Revenue retained from a cohort including expansions, contractions, and churn.

Formula:

```
WITH cohort AS (
  SELECT
    account_id,
    SUM(monthly_amount) as starting_mrr
  FROM subscriptions
  WHERE DATE_TRUNC(created_at, MONTH) = DATE '2023-01-01'
  GROUP BY account_id
),
current AS (
  SELECT
    account_id,
    SUM(monthly_amount) as current_mrr
  FROM subscriptions
  WHERE status = 'active'
  GROUP BY account_id
)
SELECT
  SUM(COALESCE(c.current_mrr, 0)) / SUM(cohort.starting_mrr) as nrr
FROM cohort
LEFT JOIN current c USING (account_id)
```

Target: >100% (indicates expansion exceeds churn)

Customer Acquisition Metrics

Customer Acquisition Cost (CAC)

Definition: Fully loaded cost to acquire one new customer.

Formula:

```
CAC = (Sales & Marketing Spend) / (New Customers Acquired)
```

Acme-specific inclusions:

- Marketing spend (ads, events, tools, agencies)
- Sales compensation (base + commission + benefits)
- SDR team compensation
- Marketing team compensation
- EXCLUDE: Customer success, support, product marketing

Time period: Typically monthly or quarterly

Lifetime Value (LTV)

Definition: Expected total gross margin from a customer over their lifetime.

Formula:

```
LTV = (Average Monthly Revenue per Account) × (Average Lifetime Months) × (Gross Margi
```

Acme calculation:

- Average Monthly Revenue: $850 (from last 12 months cohorts)
- Average Lifetime: 28 months (from cohorts with 18+ months maturity)
- Gross Margin: 82%
- **LTV = $850 × 28 × 0.82 = $19,516**

LTV:CAC Ratio: Target >3:1 (ideally 4:1 to 5:1)

Churn Metrics

Logo Churn Rate

Definition: Percentage of customers who cancelled in the period.

Formula:

```
SELECT
  COUNT(DISTINCT churned.account_id) /
  COUNT(DISTINCT start_of_period.account_id) as logo_churn_rate
FROM
  (SELECT account_id FROM subscriptions
   WHERE status = 'active'
   AND DATE(status_updated_at) = '2024-01-01') start_of_period
LEFT JOIN
  (SELECT account_id FROM subscriptions
   WHERE status = 'cancelled'
   AND DATE(status_updated_at) BETWEEN '2024-01-01' AND '2024-01-31') churned
  USING (account_id)
```

Acme target: <3% monthly (enterprise), <5% (SMB)

MRR Churn Rate

Definition: Percentage of MRR lost in the period.

Formula: `Churned MRR / Starting MRR`

Acme calculation notes:

- Use beginning-of-period MRR as denominator (not average)
- Include downgrades as partial churn
- Exclude expansions from churn calculation (count separately)

Standard Filters

Always exclude unless explicitly analyzing test data:

```
WHERE is_test = false
  AND is_internal = false
  AND is_deleted = false
```

User activity: "Active user" = at least one event in last 30 days

Subscription states:

- Active: status = 'active'
- Trial: status = 'trial' OR is_trial = true
- Churned: status IN ('cancelled', 'expired')

````
**File: `CLAUDE.md` additions**

```markdown
## Acme Metrics Standards

When calculating or reporting any revenue, customer, or churn metrics,
always use the definitions from the acme-metrics skill. These match
our board reports and finance team calculations.

If a metric is requested that isn't defined in acme-metrics, flag it
and ask for the calculation method before proceeding.
````

Result: All analysts (and Claude) calculate metrics consistently. New analysts learn standard definitions automatically.

Adding Industry-Specific Knowledge

The data plugin is general-purpose. Add domain-specific knowledge for your industry.

Example: Healthcare Analytics

File: `.claude/skills/healthcare-analytics/SKILL.md`

```
---
name: healthcare-analytics
description: >
  Healthcare and clinical data analysis patterns, regulatory requirements,
  and common healthcare metrics. Use when analyzing patient data, claims data,
  clinical outcomes, HEDIS measures, or healthcare quality metrics.
---

# Healthcare Analytics

## Regulatory Compliance
```
```

````
### HIPAA Requirements for Analysis

When working with PHI (Protected Health Information):

1. **Limited datasets:** Remove direct identifiers before analysis
   - Names, addresses, SSN, MRN
   - Dates can be shifted or truncated to year
   - Geographic areas smaller than state must be aggregated

2. **Minimum necessary:** Only query columns needed for the analysis

3. **De-identification safe harbor:** 18 identifier types must be removed
   for data to be considered de-identified

4. **Audit logging:** All queries against PHI tables should be logged
   (happens automatically in most warehouses)

## Common Healthcare Metrics

### HEDIS Measures
[Healthcare Effectiveness Data and Information Set]

**Diabetes Care - HbA1c Control (CDC):**
- Population: Members 18-75 with Type 1 or Type 2 diabetes
- Measure: % with most recent HbA1c <8.0% or <7.0% for comprehensive control
- Data sources: Claims (diagnosis codes), lab results

**Breast Cancer Screening (BCS):**
- Population: Women 50-74 years old
- Measure: % who had mammogram in last 24 months
- Exclusions: Bilateral mastectomy, advanced illness

### Readmission Rates

**30-Day All-Cause Readmission:**
```sql
WITH index_admits AS (
  SELECT
    patient_id,
    admission_date,
    discharge_date
  FROM admissions
  WHERE discharge_disposition = 'home'
),
readmissions AS (
  SELECT
    i.patient_id,
    i.discharge_date as index_discharge,
    r.admission_date as readmit_date,
    DATE_DIFF(r.admission_date, i.discharge_date, DAY) as days_to_readmit
````
````

```
  FROM index_admits i
  JOIN admissions r ON i.patient_id = r.patient_id
  WHERE r.admission_date > i.discharge_date
    AND DATE_DIFF(r.admission_date, i.discharge_date, DAY) ⩽ 30
)
SELECT
  COUNT(DISTINCT patient_id || index_discharge) /
  COUNT(DISTINCT patient_id || discharge_date) as readmit_rate_30d
FROM index_admits
LEFT JOIN readmissions USING (patient_id, index_discharge)
```

Risk Adjustment

HCC (Hierarchical Condition Category) Scores:

- Derived from ICD-10 diagnosis codes
- Predicts future healthcare costs
- Used for Medicare Advantage risk adjustment

Common risk score interpretation:

- 1.0 = average Medicare beneficiary cost
- <1.0 = healthier/lower cost
- > *1.0 = sicker/higher cost*

Claims Data Patterns

Claims Table Structure

Typical tables:

- **Medical claims:** Inpatient, outpatient, professional
- **Pharmacy claims:** Drug dispensing records
- **Enrollment:** Eligibility and coverage periods
- **Providers:** NPI, specialty, affiliations

Common Joins

```
-- Patient with diagnosis
FROM medical_claims mc
JOIN diagnosis_codes dc ON mc.claim_id = dc.claim_id
WHERE dc.icd10_code LIKE 'E11%'  -- Type 2 diabetes

-- Patient with filled medication
FROM pharmacy_claims pc
JOIN ndc_codes nc ON pc.ndc = nc.ndc
WHERE nc.therapeutic_class = 'Antidiabetic'

-- Active enrollment check
FROM enrollment e
WHERE '2024-01-15' BETWEEN e.coverage_start AND e.coverage_end
```

Lookback Periods

Standard measurement periods:

- Quality measures: Typically 12-month measurement period
- Readmissions: 30, 60, or 90 days post-discharge
- Medication adherence: PDC (Proportion of Days Covered) over 12 months
- Chronic condition identification: 2 outpatient or 1 inpatient claim in measurement period

Terminology

HEALTHCARE TERM	DEFINITION	SQL REPRESENTATION
Member	Individual enrolled in health plan	patient_id, member_id
Enrollee	Active member during time period	Join to enrollment table
Lives	Count of covered members	COUNT(DISTINCT member_id)
PMPM	Per member per month cost	Total cost / member-months
Member-months	Sum of months each member was enrolled	SUM(months_enrolled)
PDC	Proportion of days covered (medication adherence)	Days with med / days in period

HEALTHCARE TERM	DEFINITION	SQL REPRESENTATION
NPI	National Provider Identifier	provider_id in provider table
NDC	National Drug Code	drug identifier in pharmacy claims
ICD-10	Diagnosis code system	diagnosis_code (alphanumeric, 3-7 chars)
CPT	Procedure code system	procedure_code in claims
DRG	Diagnosis Related Group (inpatient grouping)	drg_code for hospital stays

Data Quality Checks

Healthcare data common issues:

- **Duplicate claims:** Same claim submitted multiple times (check by claim_id + line_number)
- **Overlapping enrollment:** Member enrolled in multiple plans simultaneously
- **Future service dates:** Claims with dates after submission
- **Invalid codes:** ICD-10/CPT codes not in current code set
- **Age conflicts:** Birth date inconsistent with age at service

```
**Result:** Healthcare analysts can work with claims data, HEDIS measures, and regulat

---

### Customizing for Your SQL Dialect

If everyone at your company uses the same warehouse, replace dialect-agnostic patterns

**Approach: Modify the `sql-queries` skill directly (Approach A)**

**File: `skills/sql-queries/SKILL.md` (add section at top)**

```markdown
## Acme Corp Standard: Snowflake SQL

All Acme queries use Snowflake syntax. Key patterns:

### Date Functions
```sql
-- Always use these Snowflake functions
DATEADD(day, 7, created_at)       -- not DATE_ADD
DATEDIFF(day, start, end)          -- not DATE_DIFF
DATE_TRUNC('month', created_at)    -- works in Snowflake
TO_CHAR(created_at, 'YYYY-MM-DD')  -- for formatting
```

String Functions

```
-- Case-insensitive by default in Snowflake
column LIKE '%pattern%'            -- already case-insensitive
REGEXP_LIKE(column, 'pattern')     -- regex matching
column:key::STRING                 -- JSON access (VARIANT type)
```

Tables and Schemas

- Production: `PROD.ANALYTICS.*`
- Staging: `STAGING.RAW_*`
- Sandbox: `SANDBOX.USERNAME_*`

Performance

- Use clustering keys (not indexes): `CLUSTER BY (created_date, user_id)`
- Filter on clustered columns first for partition pruning

- Use RESULT_SCAN(LAST_QUERY_ID()) to avoid re-running expensive queries
- Set warehouse size based on query: XS (dev), S (standard), M (heavy agg), L (large scans)

```
**Result:** All generated queries use Snowflake syntax by default. Analysts don't need

---

## Part VII: Advanced Topics

### Working with Large Datasets (100M+ Rows)

**Challenge:** Profiling and analyzing massive tables is slow and expensive.

**Solutions:**

**1. Sampling for exploration**

```sql
-- BigQuery: Sample 1% of table
SELECT * FROM `project.dataset.huge_table` TABLESAMPLE SYSTEM (1 PERCENT)

-- Snowflake: Sample 10,000 rows
SELECT * FROM huge_table SAMPLE (10000 ROWS)

-- PostgreSQL: Random sample
SELECT * FROM huge_table TABLESAMPLE BERNOULLI (1)  -- 1%
```

Use sampling for `/explore-data` on very large tables, then note in results: "Statistics based on 1% sample (approx 1M rows)."

2. Pre-aggregation

For dashboards and recurring reports, pre-aggregate to daily or hourly level:

```
-- Instead of querying 500M raw events
SELECT DATE_TRUNC('day', event_date) as day,
       event_type,
       COUNT(*) as event_count
FROM events
GROUP BY 1, 2

-- Creates a 100K row summary table
-- Dashboard queries this summary, not raw events
```

3. Partitioning awareness
```

Always filter on partition keys:

```
-- BAD: Scans entire 500M row table
SELECT * FROM events WHERE user_id = 'abc123'

-- GOOD: Scans only 1 day partition (~1.5M rows)
SELECT * FROM events
WHERE event_date = '2024-02-14'
  AND user_id = 'abc123'
```

BigQuery, Snowflake, and most warehouses partition by date. Include date filters in every query.

4. Materialized views

For expensive aggregations used repeatedly:

```
-- Snowflake materialized view
CREATE MATERIALIZED VIEW daily_user_activity AS
SELECT
  DATE_TRUNC('day', event_date) as day,
  user_id,
  COUNT(*) as event_count,
  COUNT(DISTINCT session_id) as session_count
FROM events
GROUP BY 1, 2;

-- Queries hit the materialized view, not raw events
```

5. Incremental queries

For time series, query recent data only:

```
-- Last 7 days (fast)
WHERE event_date ⩾ CURRENT_DATE - 7

-- All history (slow)
WHERE event_date ⩾ '2020-01-01'
```

Build historical trends incrementally (query month-by-month, union results) rather than all at once.

Debugging Incorrect Results

When `/analyze` returns unexpected results, use `/validate` to diagnose:

```
/validate [paste the analysis and query]
```

Common issues `/validate` catches:

Join explosions:

```
-- WRONG: Many-to-many join doubles counts
SELECT COUNT(*) FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id

-- CORRECT: Count distinct orders
SELECT COUNT(DISTINCT o.order_id) FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
```

Wrong aggregation level:

```
-- WRONG: Averaging pre-computed averages
SELECT channel, AVG(avg_order_value) FROM daily_stats GROUP BY channel

-- CORRECT: Aggregate from raw data
SELECT channel, AVG(order_value) FROM orders GROUP BY channel
```

Survivorship bias:

```
-- WRONG: Only current users (misses churned users)
SELECT AVG(lifetime_value) FROM users WHERE status = 'active'

-- CORRECT: All users including churned
SELECT AVG(lifetime_value) FROM users
```

Timezone issues:

```
-- WRONG: Mixing UTC timestamps with local dates
WHERE DATE(event_timestamp_utc) = '2024-02-14'  -- Misses late-day events

-- CORRECT: Convert to consistent timezone
WHERE DATE(event_timestamp_utc AT TIME ZONE 'America/New_York') = '2024-02-14'
```

Building Reusable Analysis Templates

For recurring analyses, create command templates:

File: `commands/weekly-product-metrics.md`

```
---
description: Generate weekly product metrics report (signups, activation, retention)
argument-hint: "[week_start_date]"
---

Generate the weekly product metrics report for the week starting $1
(or current week if not specified).

## Metrics to Include

Query the following from the data warehouse:

1. **Signups:** New user registrations
2. **Activation:** % of signups who completed onboarding
3. **Weekly Active Users (WAU):** Distinct users with ≥1 event
4. **Retention:** % of prior week's active users still active this week
5. **Feature adoption:** % of WAU who used key features

## Output Format

Present as:
1. KPI summary table with week-over-week changes
2. Line chart: 8-week trend for each metric
3. Bar chart: Feature adoption breakdown
4. Brief narrative highlighting key changes

## Validation

Before presenting:
- Verify row counts match known dashboard figures
- Check for partial week data (exclude current week if incomplete)
- Confirm no test users included (is_test = false filter)

## Distribution

Save the report as `product_metrics_YYYY_WW.html` (year and week number).
```

Usage:

```
/weekly-product-metrics 2024-02-12
```

Claude runs the full analysis, generates visualizations, validates results, and produces a shareable HTML file — all from a single command.

Integrating with dbt or Data Pipelines

If your warehouse has dbt models, document them in a skill so Claude knows which tables are source-of-truth:

File: `.claude/skills/acme-data-models/SKILL.md`

```
---
name: acme-data-models
description: >
  Acme Corp dbt data models, mart layer structure, and source-of-truth
  tables. Use when determining which table to query for metrics or
  understanding table lineage.
---

# Acme Data Models

## Mart Layer (Source of Truth)

Always query mart layer for analysis. Never query raw or staging tables directly.

| Mart Table | Description | Grain | Refresh |
|---|---|---|---|
| `ANALYTICS.MART_USERS` | Customer master table | One row per user | Daily, 2am UTC |
| `ANALYTICS.MART_SUBSCRIPTIONS` | Subscription history | One row per subscription | D
| `ANALYTICS.MART_EVENTS` | Product usage events | One row per event | Hourly |
| `ANALYTICS.MART_REVENUE` | Revenue recognition | One row per transaction | Daily, 6a

## Key Dimensions and Facts

**Mart Users:**
- user_id (PK)
- account_id (FK to mart_accounts)
- Dimensions: plan_type, signup_source, country, industry
- Metrics: lifetime_value, total_spend, days_since_last_activity
- Dates: created_at, last_login_at

**Mart Subscriptions:**
- subscription_id (PK)
- account_id, user_id (FKs)
- Dimensions: plan_name, billing_frequency, status
- Metrics: mrr, annual_value
- Dates: created_at, start_date, end_date, cancelled_at
```

```
**Mart Events:**
- event_id (PK)
- user_id, session_id (FKs)
- Partitioned by: event_date (ALWAYS filter on this)
- Dimensions: event_type, platform, browser
- Metrics: event_count (use COUNT(*) not this column)

## Table Lineage
```

RAW (Fivetran sync) └── STAGING (cleaned, typed) └── INTERMEDIATE (business logic) └── MART (analytics-ready)

```
## Query Pattern

```sql
-- CORRECT: Query mart layer
SELECT
  plan_type,
  COUNT(*) as user_count,
  AVG(lifetime_value) as avg_ltv
FROM ANALYTICS.MART_USERS
WHERE status = 'active'
GROUP BY plan_type

-- WRONG: Never query raw or staging directly
SELECT * FROM RAW.SALESFORCE_CONTACTS  -- DON'T DO THIS
```

dbt Docs

Full documentation: https://acme.getdbt.com/#!/overview Lineage diagrams: https://acme.getdbt.com/#!/model/mart_users

```
**Result:** Claude always queries the right tables and understands data lineage.

---

### Version Control for Analyses

For important recurring analyses, version control the SQL and documentation:

**Structure:**
```
```

your-repo/ ├── sql/ │ └── monthly_revenue_report/ │ ├── mrr_calculation.sql │ ├── churn_analysis.sql │ └── README.md ├── dashboards/ │ └── revenue_dashboard.html └── docs/ └── revenue_methodology.md

```
**Include in README.md:**
- What the analysis calculates
- Data sources and freshness requirements
- Assumptions and limitations
- How to run (manual steps or scheduling)
- Change log (what changed in each version)

**Git workflow:**

```bash
# Create feature branch for changes
git checkout -b update-mrr-calculation

# Make changes to SQL
# Test with Claude: /validate sql/monthly_revenue_report/mrr_calculation.sql

# Commit with descriptive message
git commit -m "Fix MRR calculation to exclude trials"

# Merge to main after review
git merge update-mrr-calculation
```

Benefits:

- Track who changed what and why
- Roll back to previous versions if new logic is wrong
- Code review for important analyses
- Documentation co-located with code

Appendix A: Command Quick Reference

COMMAND	WHAT IT DOES	TYPICAL USE CASE
`/analyze <question>`	Answer data questions from quick lookups to formal reports	Ad-hoc requests, stakeholder questions, exploratory analysis
```

COMMAND	WHAT IT DOES	TYPICAL USE CASE
`/explore-data <table>`	Profile dataset structure, quality, patterns	First time using a table, data quality assessment
`/write-query <description>`	Generate optimized SQL for your dialect	Translate requirements to SQL, learn SQL patterns
`/create-viz <data> [type]`	Create publication-quality charts with Python	Presentation visuals, exploring patterns
`/build-dashboard <description>`	Build interactive HTML dashboard	Executive reports, team dashboards, shareable insights
`/validate <analysis>`	QA analysis for errors before sharing	Pre-presentation review, sanity check

Appendix B: Skill Trigger Reference

SKILL	TRIGGERS	WHAT IT PROVIDES
`sql-queries`	Writing queries, optimizing SQL, translating dialects	Dialect syntax, patterns (CTEs, window functions, cohorts), performance tips
`data-exploration`	New datasets, data quality, profiling, nulls/outliers	Profiling methodology, quality frameworks, pattern discovery
`data-visualization`	Building charts, chart selection, design principles	Chart type guide, Python code, design principles, accessibility
`statistical-analysis`	Distributions, significance testing, trends, correlations	Descriptive stats, hypothesis testing, trend analysis, caution flags
`data-validation`	Reviewing analyses, checking calculations, bias detection	QA checklist, common pitfalls, sanity checks, documentation standards
`interactive-dashboard-builder`	Creating dashboards, HTML reports, interactive charts	Dashboard templates, Chart.js patterns, filters, styling
`data-context-extractor`	"Create a data context skill", company-specific setup	Extracts tribal knowledge, generates custom skills

Appendix C: Troubleshooting

Problem: `/analyze` asks for data instead of querying the warehouse

Solution: MCP server not connected. Check `.mcp.json` and authenticate if needed. Verify with: "List available data sources"

Problem: Query returns "column not found" error

Solution:

1. Schema may have changed. Run `/explore-data <table>` to see current columns
2. Column names may be case-sensitive (PostgreSQL). Check exact capitalization
3. Table name may need schema prefix: `schema.table` not just `table`

Problem: Dashboard filters don't update charts

Solution: JavaScript error in generated code. Check browser console (F12). Report issue with `/build-dashboard` for debugging.

Problem: Results seem wrong (off by 10x, unexpected zeros, etc.)

Solution: Run `/validate` on the analysis. Common causes:

- Join explosion (many-to-many)
- Wrong aggregation level (averaging averages)
- Timezone mismatch
- Test data included
- Null handling (NULLs treated as zeros)

Problem: Visualizations have wrong chart type

Solution: Specify chart type explicitly: `/create-viz <data> as a horizontal bar chart` or `/create-viz <data> as a line chart with markers`

Problem: SQL query too slow

Solution:

1. Add partition filter (date column) at the top of the query
2. Select only needed columns (not SELECT *)
3. Check query plan: `EXPLAIN <query>`
4. Ask Claude: "Optimize this query for [your warehouse]"

Appendix D: Additional Resources

Official documentation:

- SQL dialect references in `sql-queries` skill
- Chart.js documentation: https://www.chartjs.org/docs/
- MCP server registry: https://github.com/modelcontextprotocol/servers

Learning resources:

- Mode Analytics SQL tutorial: https://mode.com/sql-tutorial/
- Storytelling with Data (book by Cole Nussbaumer Knaflic) — visualization best practices
- Thinking with Data (book by Max Shron) — analytical thinking

Complementary plugins:

- `cowork-plugin-management` — for creating and customizing plugins
- `knowledge-work` — for working with documents and presentations

Conclusion

The Data plugin transforms Claude into a collaborative data analyst capable of end-to-end analytical workflows. Whether you're writing SQL, exploring datasets, generating visualizations, building dashboards, or validating analyses, the plugin provides the knowledge and structure to work faster and more accurately.

Start simple: Begin with `/analyze` for ad-hoc questions and `/explore-data` for new tables. As you get comfortable, leverage `/build-dashboard` for recurring reports and `/validate` for quality assurance.

Customize thoughtfully: Add company-specific metric definitions and terminology through complementary skills. For team-wide changes, fork the plugin. For project-specific needs, use CLAUDE.md rules.

Iterate and improve: Every analysis teaches the plugin more about your data. Use the `data-context-extractor` skill to capture tribal knowledge and generate custom skills that make future analyses faster and more accurate.

The goal is not to replace data analysts but to augment them — handling the repetitive parts (writing boilerplate SQL, formatting charts, running QA checks) so analysts can focus on the interesting parts: asking better questions, finding deeper insights, and telling compelling stories with data.

RationalEyes.ai

contact@rationaleyes.ai

Intelligent Automation for Knowledge Work

Data Plugin Handbook — Version 1.0.0