Tools in Data Science Manual

Every Tool · Every Task · Every Lifecycle Stage

Table of Contents

SECTION 0 — The Data Stack in 2026

0.1 Roles at a Glance

```
DATA ANALYST
  Core tools:  SQL, Excel/Sheets, Tableau/Power BI/Looker, Python (pandas)
  Main tasks:  Reporting, dashboards, ad-hoc queries, business metrics
  Output:      Insights, charts, decks, KPI reports

DATA SCIENTIST
  Core tools:  Python (sklearn, pandas, xgboost), SQL, Jupyter, MLflow
  Main tasks:  ML models, experimentation, statistical analysis, forecastin
  Output:      Models, predictions, recommendations, experiments

DATA ENGINEER
  Core tools:  Python, SQL, Spark, Airflow/Prefect, dbt, Kafka, cloud (AWS/
  Main tasks:  Build pipelines, maintain warehouse, data quality, infrastru
  Output:      Reliable data pipelines, clean tables, scalable systems

ANALYTICS ENGINEER  (emerging hybrid role)
  Core tools:  dbt, SQL, Python, Looker/Lightdash
  Main tasks:  Transform raw data into clean models, own data layer
  Output:      dbt models, tested data, documentation
```

0.2 Modern Data Stack Overview

```
INGESTION          STORAGE          TRANSFORM        SERVE
──────────         ──────────       ──────────       ──────────
Fivetran           Snowflake        dbt              Looker
Airbyte            BigQuery         Spark            Tableau
Stitch             Redshift         Pandas           Power BI
Custom Python      Delta Lake       PySpark          Grafana
Kafka              S3/GCS/ADLS      Flink            Streamlit
Debezium           PostgreSQL       Airflow          FastAPI
Singer             DuckDB           Prefect          Superset

QUALITY            GOVERNANCE       MLOPS            AI/LLM
──────────         ──────────       ──────────       ──────────
```

Great Expectations	Datahub	MLflow	OpenAI API
dbt tests	Alation	Weights & Biases	LangChain
Soda	Monte Carlo	BentoML	Hugging Face
Anomalo	Collibra	Ray Serve	Ollama

0.3 Project Structure (Data Engineer Style)

```
data_project/
├── ingestion/           # Raw data loading scripts
├── dbt/                 # Transformation models
│   ├── models/
│   │   ├── staging/     # Clean raw data
│   │   ├── intermediate/ # Business logic
│   │   └── marts/       # Final tables for BI
│   └── tests/
├── pipelines/           # Airflow DAGs or Prefect flows
├── notebooks/           # Analysis notebooks
├── src/
│   ├── extract.py
│   ├── transform.py
│   └── load.py
├── tests/               # Unit tests
├── docker/
├── .env                 # Secrets (never commit)
├── requirements.txt
└── README.md
```

SECTION 1 — SQL Mastery (Every Role Needs This)

1.1 Core Query Patterns

```sql
-- Basic anatomy
SELECT
    column1,
```

```
    column2,
    COUNT(*) AS row_count,
    SUM(revenue) AS total_revenue,
    AVG(revenue) AS avg_revenue
FROM schema.table_name
WHERE created_at >= '2024-01-01'
    AND status = 'active'
    AND revenue > 0
GROUP BY column1, column2
HAVING COUNT(*) > 10
ORDER BY total_revenue DESC
LIMIT 100;

-- COUNT variants
COUNT(*)            -- count all rows including NULLs
COUNT(column)       -- count non-NULL values
COUNT(DISTINCT id)  -- count unique non-NULL values

-- Conditionals
CASE
    WHEN revenue > 10000 THEN 'high'
    WHEN revenue > 1000  THEN 'medium'
    ELSE 'low'
END AS revenue_tier

-- COALESCE (return first non-null)
COALESCE(phone, email, 'no_contact') AS contact

-- NULLIF (return null if values match)
NULLIF(denominator, 0)   -- avoid divide by zero
```

1.2 Window Functions (Most Important Advanced SQL)

```
-- Syntax: function() OVER (PARTITION BY ... ORDER BY ... ROWS/RANGE ...)

-- ROW_NUMBER — unique sequential number per partition
SELECT
    user_id,
    order_id,
    order_date,
    ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY order_date) AS order_n
FROM orders;
```

```sql
-- RANK vs DENSE_RANK
-- RANK: 1,2,2,4 (skips after tie)
-- DENSE_RANK: 1,2,2,3 (no skip)
RANK()       OVER (PARTITION BY dept ORDER BY salary DESC) AS rank_in_dept,
DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dense_rank

-- LAG / LEAD — access previous/next row
LAG(revenue, 1) OVER (PARTITION BY user_id ORDER BY month)  AS prev_month_r
LEAD(revenue, 1) OVER (PARTITION BY user_id ORDER BY month) AS next_month_r

-- Running totals / moving averages
SUM(revenue) OVER (ORDER BY date ROWS UNBOUNDED PRECEDING)      AS cumulati
AVG(revenue) OVER (ORDER BY date ROWS 6 PRECEDING)              AS rolling_
SUM(revenue) OVER (PARTITION BY user_id ORDER BY date)          AS user_cum

-- Percent of total
revenue / SUM(revenue) OVER () * 100 AS pct_of_total,
revenue / SUM(revenue) OVER (PARTITION BY region) * 100 AS pct_of_region,

-- NTILE — bucket into N equal groups
NTILE(4) OVER (ORDER BY revenue DESC) AS revenue_quartile,

-- FIRST_VALUE / LAST_VALUE
FIRST_VALUE(product) OVER (PARTITION BY user_id ORDER BY order_date) AS fir
LAST_VALUE(product) OVER (PARTITION BY user_id ORDER BY order_date
                          ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FO
```

1.3 CTEs and Subqueries

```sql
-- CTE (Common Table Expression) — preferred over subqueries
WITH
active_users AS (
    SELECT user_id, created_at
    FROM users
    WHERE status = 'active'
),
user_orders AS (
    SELECT
        u.user_id,
        COUNT(o.order_id)   AS total_orders,
        SUM(o.revenue)      AS lifetime_value,
        MAX(o.order_date)   AS last_order_date
    FROM active_users u
```

```
    LEFT JOIN orders o ON u.user_id = o.user_id
    GROUP BY u.user_id
),
ltv_tiers AS (
    SELECT
        user_id,
        total_orders,
        lifetime_value,
        NTILE(4) OVER (ORDER BY lifetime_value) AS ltv_quartile
    FROM user_orders
)
SELECT
    ltv_quartile,
    COUNT(*) AS user_count,
    AVG(lifetime_value) AS avg_ltv,
    AVG(total_orders) AS avg_orders
FROM ltv_tiers
GROUP BY ltv_quartile
ORDER BY ltv_quartile;

-- Recursive CTE (for hierarchies / org charts)
WITH RECURSIVE org_hierarchy AS (
    -- Base: top-level employees (no manager)
    SELECT employee_id, name, manager_id, 0 AS level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive: employees reporting to someone in previous level
    SELECT e.employee_id, e.name, e.manager_id, h.level + 1
    FROM employees e
    JOIN org_hierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM org_hierarchy ORDER BY level, name;
```

1.4 JOINs — Full Reference

```
-- INNER: only matching rows
SELECT * FROM a INNER JOIN b ON a.id = b.id;

-- LEFT: all from left, NULLs for unmatched right
SELECT * FROM a LEFT JOIN b ON a.id = b.id;
```

```sql
-- RIGHT: all from right
SELECT * FROM a RIGHT JOIN b ON a.id = b.id;

-- FULL OUTER: all from both
SELECT * FROM a FULL OUTER JOIN b ON a.id = b.id;

-- CROSS: cartesian product (every combo)
SELECT * FROM a CROSS JOIN b;

-- SELF JOIN (e.g., employee → manager)
SELECT e.name, m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;

-- Find rows in A not in B (anti-join)
SELECT a.* FROM a LEFT JOIN b ON a.id = b.id WHERE b.id IS NULL;
-- Or with NOT EXISTS:
SELECT * FROM a WHERE NOT EXISTS (SELECT 1 FROM b WHERE b.id = a.id);

-- Multiple join conditions
SELECT * FROM orders o
JOIN promotions p ON o.user_id = p.user_id AND o.order_date BETWEEN p.start
```

1.5 Advanced Patterns

```sql
-- Pivot / Unpivot
-- Pivot (rows → columns)
SELECT
    month,
    SUM(CASE WHEN product = 'A' THEN revenue ELSE 0 END) AS product_a,
    SUM(CASE WHEN product = 'B' THEN revenue ELSE 0 END) AS product_b,
    SUM(CASE WHEN product = 'C' THEN revenue ELSE 0 END) AS product_c
FROM sales
GROUP BY month;

-- Unpivot (columns → rows) — BigQuery syntax
SELECT month, product, revenue
FROM sales
UNPIVOT (revenue FOR product IN (product_a, product_b, product_c));

-- Deduplication — keep latest record per user
WITH ranked AS (
```

```
    SELECT *, ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY updated_at D
    FROM users_raw
)
SELECT * FROM ranked WHERE rn = 1;

-- Gaps and islands (consecutive sequences)
WITH numbered AS (
    SELECT date, ROW_NUMBER() OVER (ORDER BY date) AS rn
    FROM active_days
),
grouped AS (
    SELECT date, DATE_SUB(date, INTERVAL rn DAY) AS grp
    FROM numbered
)
SELECT MIN(date) AS start, MAX(date) AS end, COUNT(*) AS days
FROM grouped
GROUP BY grp
ORDER BY start;

-- Running median (BigQuery)
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY revenue) OVER ()

-- Date arithmetic
DATE_DIFF(end_date, start_date, DAY)    -- BigQuery
DATEDIFF(end_date, start_date)          -- MySQL/Redshift
end_date - start_date                   -- PostgreSQL

-- String manipulation
CONCAT(first_name, ' ', last_name)
SUBSTRING(email, 1, CHARINDEX('@', email) - 1)  -- SQL Server
SPLIT_PART(email, '@', 1)                        -- PostgreSQL
REGEXP_EXTRACT(url, r'https?://([^/]+)')         -- BigQuery

-- JSON parsing (BigQuery)
JSON_EXTRACT_SCALAR(json_col, '$.user.name')
JSON_EXTRACT_ARRAY(json_col, '$.items')

-- Array handling (BigQuery)
ARRAY_LENGTH(arr_col)
arr_col[OFFSET(0)]
UNNEST(arr_col) AS item
```

1.6 Performance Optimization

```sql
-- 1. Use WHERE before JOIN to reduce rows early
-- BAD:
SELECT * FROM large_table l JOIN small_table s ON l.id = s.id WHERE l.date
-- BETTER: filter in CTE
WITH filtered AS (SELECT * FROM large_table WHERE date > '2024-01-01')
SELECT * FROM filtered f JOIN small_table s ON f.id = s.id;

-- 2. Avoid SELECT * — name columns explicitly
-- 3. Use approximate functions on huge datasets
SELECT APPROX_COUNT_DISTINCT(user_id) FROM events;   -- BigQuery
SELECT HLL_COUNT.MERGE(HLL_COUNT.INIT(user_id))...   -- BigQuery exact HLL

-- 4. Partition pruning — always filter on partition column
WHERE event_date BETWEEN '2024-01-01' AND '2024-03-31'   -- uses partition

-- 5. Explain query plan
EXPLAIN SELECT * FROM orders WHERE user_id = 123;
EXPLAIN ANALYZE SELECT * FROM orders WHERE user_id = 123;  -- PostgreSQL (a

-- 6. Indexes (PostgreSQL)
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_orders_date ON orders(order_date DESC);
CREATE INDEX idx_orders_user_date ON orders(user_id, order_date);  -- compo

-- 7. Materialized views (cache expensive queries)
CREATE MATERIALIZED VIEW daily_revenue AS
SELECT DATE(created_at) AS day, SUM(revenue) AS revenue
FROM orders GROUP BY 1;

REFRESH MATERIALIZED VIEW daily_revenue;
```

SECTION 2 — Python for Data Analysis

2.1 Pandas Power Patterns

```python
import pandas as pd
import numpy as np
```

```python
# Read large files efficiently
df = pd.read_csv(
    "large_file.csv",
    dtype={"user_id": "int32", "amount": "float32"},  # reduce memory
    usecols=["user_id", "amount", "date"],             # only needed cols
    parse_dates=["date"],
    chunksize=None  # set to 10000 to read in chunks
)

# Memory usage
print(df.memory_usage(deep=True).sum() / 1024**2, "MB")

# Downcast to save memory
df["amount"] = pd.to_numeric(df["amount"], downcast="float")
df["user_id"] = pd.to_numeric(df["user_id"], downcast="integer")
df["category"] = df["category"].astype("category")  # huge savings for low-

# Read in chunks (for files bigger than RAM)
chunks = []
for chunk in pd.read_csv("huge_file.csv", chunksize=100_000):
    filtered = chunk[chunk["amount"] > 0]
    chunks.append(filtered)
df = pd.concat(chunks, ignore_index=True)
```

2.2 Advanced Pandas Operations

```python
# Multi-level groupby + transform
df["user_spend_pct"] = (
    df["amount"] / df.groupby("user_id")["amount"].transform("sum") * 100
)

# Rolling calculations
df = df.sort_values("date")
df["7d_rolling_avg"] = df.groupby("user_id")["amount"].transform(
    lambda x: x.rolling(7, min_periods=1).mean()
)

# Explode — expand lists into rows
df["tags"] = df["tags"].str.split(",")
df_exploded = df.explode("tags")

# Stack / Unstack
pivoted = df.pivot_table(values="revenue", index="region", columns="month",
```

```
# Flatten multi-index columns
pivoted.columns = [f"{col[0]}_{col[1]}" for col in pivoted.columns]

# Apply with multiple return values
def parse_address(addr):
    parts = addr.split(",")
    return pd.Series({"city": parts[0].strip(), "state": parts[1].strip() i

df[["city", "state"]] = df["address"].apply(parse_address)

# Vectorized string operations (much faster than .apply)
df["domain"] = df["email"].str.extract(r"@(.+)$")
df["initials"] = df["first_name"].str[0] + df["last_name"].str[0]

# merge_asof — join on nearest key (great for time series)
pd.merge_asof(
    df_orders.sort_values("order_date"),
    df_prices.sort_values("price_date"),
    left_on="order_date",
    right_on="price_date",
    by="product_id",
    direction="backward"  # use last known price
)
```

2.3 Profiling a Dataset

```
# Quick custom profiler
def profile_dataframe(df):
    report = pd.DataFrame({
        "dtype": df.dtypes,
        "null_count": df.isnull().sum(),
        "null_pct": (df.isnull().mean() * 100).round(2),
        "unique_count": df.nunique(),
        "cardinality_pct": (df.nunique() / len(df) * 100).round(2)
    })
    report["sample_values"] = [df[c].dropna().head(3).tolist() for c in df.
    return report

print(profile_dataframe(df).to_string())

# ydata-profiling (formerly pandas-profiling)
# pip install ydata-profiling
from ydata_profiling import ProfileReport
```

```
profile = ProfileReport(df, title="Dataset Profile", explorative=True)
profile.to_file("outputs/profile.html")
```

2.4 Working with APIs

```
import requests
import pandas as pd
import time

# Basic GET
response = requests.get(
    "https://api.example.com/data",
    headers={"Authorization": "Bearer YOUR_TOKEN"},
    params={"start_date": "2024-01-01", "end_date": "2024-12-31"}
)
response.raise_for_status()  # raises HTTPError if 4xx/5xx
data = response.json()

# Paginated API (cursor-based)
def fetch_all_pages(base_url: str, headers: dict, params: dict) -> list:
    results = []
    url = base_url
    while url:
        r = requests.get(url, headers=headers, params=params)
        r.raise_for_status()
        payload = r.json()
        results.extend(payload.get("data", []))
        url = payload.get("next_cursor")  # or "next_url", "next_page_token
        params = {}  # cursor includes all params
        time.sleep(0.2)  # rate limiting
    return results

# Paginated API (offset-based)
def fetch_paginated(base_url: str, headers: dict) -> list:
    results = []
    offset = 0
    limit = 100
    while True:
        r = requests.get(base_url, headers=headers, params={"limit": limit,
        r.raise_for_status()
        page = r.json()
        if not page["data"]:
            break
```

```
        results.extend(page["data"])
        offset += limit
    return results

df = pd.DataFrame(fetch_paginated("https://api.example.com/records", {}))
```

SECTION 3 — Data Engineering Fundamentals

3.1 Database Connections

```
# PostgreSQL
import psycopg2
import pandas as pd
from sqlalchemy import create_engine, text

# SQLAlchemy engine (recommended — works with pandas)
engine = create_engine(
    "postgresql+psycopg2://user:password@host:5432/dbname",
    pool_size=5,
    max_overflow=10,
    pool_pre_ping=True  # handle dropped connections
)

# Read
df = pd.read_sql("SELECT * FROM orders WHERE date > '2024-01-01'", engine)

# Write
df.to_sql("processed_orders", engine, schema="staging", if_exists="replace"
          index=False, chunksize=10000, method="multi")

# Execute raw SQL
with engine.connect() as conn:
    conn.execute(text("TRUNCATE TABLE staging.temp_table"))
    conn.commit()

# BigQuery
```

```
from google.cloud import bigquery
client = bigquery.Client(project="my-project")
df = client.query("SELECT * FROM `my-project.dataset.table`").to_dataframe(

# Write to BigQuery
pandas_gbq.to_gbq(df, "dataset.table", project_id="my-project", if_exists="

# Snowflake
from snowflake.connector import connect
import snowflake.connector.pandas_tools as sf_tools

conn = connect(
    user="user", password="pass", account="account.region",
    warehouse="COMPUTE_WH", database="DB", schema="SCHEMA"
)
df = pd.read_sql("SELECT * FROM table", conn)
```

3.2 Reading Different File Formats

```
import pandas as pd
import pyarrow.parquet as pq

# Parquet — fast, columnar, compressed
df = pd.read_parquet("data/file.parquet")
df = pd.read_parquet("data/file.parquet", columns=["id", "date", "amount"])

# Read partitioned parquet (Hive-style: /date=2024-01-01/part-0.parquet)
import pyarrow.dataset as ds
dataset = ds.dataset("s3://bucket/data/", partitioning="hive")
df = dataset.to_table(filter=ds.field("date") > "2024-01-01").to_pandas()

# JSON Lines (.jsonl) — one JSON object per line
df = pd.read_json("data/events.jsonl", lines=True)

# Nested JSON to flat DataFrame
import json
with open("data/nested.json") as f:
    data = json.load(f)
df = pd.json_normalize(data["records"], max_level=2)

# Avro (requires fastavro)
import fastavro
with open("data/file.avro", "rb") as f:
```

```
    reader = fastavro.reader(f)
    records = list(reader)
df = pd.DataFrame(records)

# Delta Lake (requires deltalake)
from deltalake import DeltaTable
dt = DeltaTable("s3://bucket/delta_table/")
df = dt.to_pandas()
df_filtered = dt.to_pandas(filters=[("date", ">=", "2024-01-01")])
```

3.3 S3 / Cloud Storage

```
import boto3
import pandas as pd
from io import BytesIO

# AWS S3
s3 = boto3.client(
    "s3",
    aws_access_key_id="KEY",
    aws_secret_access_key="SECRET",
    region_name="us-east-1"
)

# Read from S3
obj = s3.get_object(Bucket="my-bucket", Key="data/file.csv")
df = pd.read_csv(BytesIO(obj["Body"].read()))

# Write to S3
buffer = BytesIO()
df.to_parquet(buffer, index=False)
s3.put_object(Bucket="my-bucket", Key="processed/file.parquet", Body=buffer

# List files
response = s3.list_objects_v2(Bucket="my-bucket", Prefix="data/2024/")
files = [obj["Key"] for obj in response.get("Contents", [])]

# Using s3fs (pandas-native S3 access)
import s3fs
fs = s3fs.S3FileSystem(anon=False)
df = pd.read_parquet("s3://my-bucket/data/file.parquet", filesystem=fs)

# GCS
```

```
from google.cloud import storage
client = storage.Client()
bucket = client.bucket("my-bucket")
blob = bucket.blob("data/file.csv")
df = pd.read_csv(BytesIO(blob.download_as_bytes()))
```

SECTION 4 — ETL & ELT Pipelines

4.1 Extract Pattern

```
import pandas as pd
import requests
import logging
from datetime import datetime, date
from typing import Optional

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)s %
logger = logging.getLogger(__name__)

class Extractor:
    """Generic extractor with retry logic and validation."""

    def __init__(self, source_name: str):
        self.source_name = source_name

    def extract_csv(self, path: str, **kwargs) -> pd.DataFrame:
        logger.info(f"Extracting CSV from {path}")
        df = pd.read_csv(path, **kwargs)
        logger.info(f"Extracted {len(df)} rows, {df.shape[1]} columns")
        return df

    def extract_api(self, url: str, headers: dict = None, params: dict = No
        logger.info(f"Extracting API: {url}")
        response = requests.get(url, headers=headers or {}, params=params o
        response.raise_for_status()
        data = response.json()
        df = pd.DataFrame(data if isinstance(data, list) else data.get("dat
        logger.info(f"Extracted {len(df)} rows from API")
```

```
        return df

    def extract_sql(self, engine, query: str, params: dict = None) -> pd.Da
        logger.info(f"Extracting SQL query from {self.source_name}")
        df = pd.read_sql(query, engine, params=params)
        logger.info(f"Extracted {len(df)} rows")
        return df
```

4.2 Transform Pattern

```
class Transformer:
    """Chainable transformation steps with validation."""

    def __init__(self, df: pd.DataFrame):
        self.df = df.copy()
        self.logs = []

    def log(self, msg: str):
        self.logs.append(f"{datetime.now().isoformat()} — {msg}")
        logger.info(msg)
        return self

    def rename_columns(self, mapping: dict):
        self.df = self.df.rename(columns=mapping)
        return self.log(f"Renamed columns: {mapping}")

    def cast_types(self, type_map: dict):
        for col, dtype in type_map.items():
            self.df[col] = self.df[col].astype(dtype)
        return self.log(f"Cast types: {type_map}")

    def drop_duplicates(self, subset: list = None):
        before = len(self.df)
        self.df = self.df.drop_duplicates(subset=subset)
        dropped = before - len(self.df)
        return self.log(f"Dropped {dropped} duplicates")

    def fill_nulls(self, fill_map: dict):
        self.df = self.df.fillna(fill_map)
        return self.log(f"Filled nulls: {list(fill_map.keys())}")

    def add_metadata(self):
        self.df["_ingested_at"] = datetime.utcnow()
```

```
        self.df["_source"] = "etl_pipeline"
        return self.log("Added metadata columns")

    def validate(self, required_cols: list, not_null_cols: list = None):
        missing = [c for c in required_cols if c not in self.df.columns]
        if missing:
            raise ValueError(f"Missing required columns: {missing}")
        if not_null_cols:
            for col in not_null_cols:
                nulls = self.df[col].isnull().sum()
                if nulls > 0:
                    logger.warning(f"Column {col} has {nulls} null values")
        return self

    def result(self) -> pd.DataFrame:
        return self.df

# Usage
raw = pd.read_csv("data/raw/orders.csv")
clean = (
    Transformer(raw)
    .rename_columns({"order_id_external": "order_id", "amt": "amount"})
    .cast_types({"amount": "float64", "user_id": "int64"})
    .drop_duplicates(subset=["order_id"])
    .fill_nulls({"discount": 0.0, "notes": ""})
    .add_metadata()
    .validate(required_cols=["order_id", "user_id", "amount"])
    .result()
)
```

4.3 Load Pattern

```
from sqlalchemy import create_engine, text
import pandas as pd

class Loader:
    """Load data to targets with upsert support."""

    def __init__(self, engine):
        self.engine = engine

    def load_replace(self, df: pd.DataFrame, table: str, schema: str = "pub
        df.to_sql(table, self.engine, schema=schema, if_exists="replace",
```

```
                  index=False, chunksize=5000, method="multi")
        logger.info(f"Replaced {len(df)} rows into {schema}.{table}")

    def load_append(self, df: pd.DataFrame, table: str, schema: str = "publ
        df.to_sql(table, self.engine, schema=schema, if_exists="append",
                  index=False, chunksize=5000, method="multi")
        logger.info(f"Appended {len(df)} rows to {schema}.{table}")

    def load_upsert_postgres(self, df: pd.DataFrame, table: str,
                             schema: str, unique_cols: list):
        """Upsert: insert new, update existing based on unique_cols."""
        staging_table = f"staging_{table}"

        # Step 1: Load to staging
        df.to_sql(staging_table, self.engine, schema=schema,
                  if_exists="replace", index=False)

        # Step 2: Upsert from staging to target
        cols = df.columns.tolist()
        update_cols = [c for c in cols if c not in unique_cols]

        conflict_clause = ", ".join(unique_cols)
        update_clause = ", ".join([f"{c} = EXCLUDED.{c}" for c in update_co
        insert_clause = ", ".join(cols)
        values_clause = ", ".join([f"s.{c}" for c in cols])

        sql = f"""
        INSERT INTO {schema}.{table} ({insert_clause})
        SELECT {values_clause} FROM {schema}.{staging_table} s
        ON CONFLICT ({conflict_clause}) DO UPDATE SET {update_clause}
        """

        with self.engine.connect() as conn:
            conn.execute(text(sql))
            conn.execute(text(f"DROP TABLE IF EXISTS {schema}.{staging_tabl
            conn.commit()

        logger.info(f"Upserted {len(df)} rows into {schema}.{table}")
```

4.4 Full ETL Pipeline

```
import schedule
import time
```

```
from datetime import datetime

def run_orders_pipeline():
    """Full ETL: source DB → transform → target warehouse."""
    logger.info("=" * 50)
    logger.info(f"Starting orders pipeline at {datetime.utcnow()}")

    try:
        src_engine = create_engine("postgresql://user:pass@source-db:5432/p
        tgt_engine = create_engine("postgresql://user:pass@warehouse:5432/d

        # E — Extract
        extractor = Extractor("orders_db")
        df_raw = extractor.extract_sql(
            src_engine,
            "SELECT * FROM orders WHERE updated_at > NOW() - INTERVAL '1 da
        )

        # T — Transform
        df_clean = (
            Transformer(df_raw)
            .rename_columns({"id": "order_id", "ts": "created_at"})
            .cast_types({"amount": "float64"})
            .drop_duplicates(subset=["order_id"])
            .add_metadata()
            .validate(["order_id", "user_id", "amount"])
            .result()
        )

        # L — Load
        loader = Loader(tgt_engine)
        loader.load_upsert_postgres(df_clean, "orders", "staging", ["order_

        logger.info(f"Pipeline completed. {len(df_clean)} rows processed.")

    except Exception as e:
        logger.error(f"Pipeline failed: {e}", exc_info=True)
        raise

# Run once
run_orders_pipeline()

# Or schedule
schedule.every().hour.do(run_orders_pipeline)
while True:
```

```
    schedule.run_pending()
    time.sleep(60)
```

SECTION 5 — Cloud Data Platforms

5.1 BigQuery (Google Cloud)

```
from google.cloud import bigquery
import pandas as pd

client = bigquery.Client(project="my-project")

# Query → DataFrame
query = """
SELECT
    DATE(created_at) AS date,
    COUNT(DISTINCT user_id) AS dau,
    SUM(revenue) AS revenue
FROM `my-project.prod.events`
WHERE DATE(created_at) >= DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY)
GROUP BY 1
ORDER BY 1
"""
df = client.query(query).to_dataframe()

# Parameterized query
query_params = [
    bigquery.ScalarQueryParameter("start_date", "STRING", "2024-01-01"),
    bigquery.ScalarQueryParameter("min_revenue", "FLOAT64", 100.0)
]
job_config = bigquery.QueryJobConfig(query_parameters=query_params)
df = client.query("SELECT * FROM table WHERE date >= @start_date AND revenu
                  job_config=job_config).to_dataframe()

# Write DataFrame to BigQuery
job_config = bigquery.LoadJobConfig(
    write_disposition="WRITE_TRUNCATE",    # WRITE_APPEND or WRITE_TRUNCATE
    create_disposition="CREATE_IF_NEEDED",
```

```
    schema=[
        bigquery.SchemaField("user_id", "INTEGER"),
        bigquery.SchemaField("revenue", "FLOAT"),
        bigquery.SchemaField("date", "DATE"),
    ]
)
job = client.load_table_from_dataframe(df, "my-project.dataset.table", job_
job.result()  # wait for job to complete
print(f"Loaded {job.output_rows} rows")

# Create table from query
job_config = bigquery.QueryJobConfig(
    destination="my-project.dataset.new_table",
    write_disposition="WRITE_TRUNCATE",
    create_disposition="CREATE_IF_NEEDED"
)
client.query(query, job_config=job_config).result()
```

5.2 Snowflake

```
-- Snowflake specific SQL

-- Time travel (query data as of 7 days ago)
SELECT * FROM orders AT (OFFSET => -60*60*24*7);
SELECT * FROM orders BEFORE (STATEMENT => 'query-id-here');

-- Clone table (zero-copy, instant)
CREATE TABLE orders_backup CLONE orders;
CREATE SCHEMA prod_backup CLONE prod;

-- Merge (upsert)
MERGE INTO target_table t
USING source_table s ON t.id = s.id
WHEN MATCHED THEN UPDATE SET t.value = s.value, t.updated_at = s.updated_at
WHEN NOT MATCHED THEN INSERT (id, value, updated_at) VALUES (s.id, s.value,

-- Copy from S3
COPY INTO my_table
FROM 's3://my-bucket/data/'
CREDENTIALS = (AWS_KEY_ID='...' AWS_SECRET_KEY='...')
FILE_FORMAT = (TYPE = 'PARQUET');

-- Streams (CDC - change data capture)
```

```
CREATE STREAM orders_stream ON TABLE orders;
SELECT * FROM orders_stream WHERE METADATA$ACTION = 'INSERT';

-- Tasks (scheduled queries)
CREATE TASK refresh_daily
    WAREHOUSE = COMPUTE_WH
    SCHEDULE = 'USING CRON 0 6 * * * UTC'
AS
INSERT INTO daily_summary SELECT ...;

ALTER TASK refresh_daily RESUME;
```

5.3 Redshift

```
-- Redshift specific

-- Distribution styles (critical for performance)
CREATE TABLE orders (
    order_id BIGINT DISTKEY,       -- distribute by this column (join key)
    user_id  BIGINT,
    amount   DECIMAL(10,2)
) SORTKEY (created_at);            -- sort on disk by date

-- DISTKEY on join column: both tables distributed same way → local join
-- DISTSTYLE ALL: replicate small tables to all nodes

-- Analyze query performance
EXPLAIN SELECT * FROM orders WHERE user_id = 123;

-- Vacuum + Analyze (maintenance)
VACUUM orders;      -- reclaim space from deleted rows
ANALYZE orders;     -- update statistics for query planner

-- Unload to S3
UNLOAD ('SELECT * FROM orders WHERE date > ''2024-01-01''')
TO 's3://my-bucket/exports/orders_'
IAM_ROLE 'arn:aws:iam::123:role/RedshiftRole'
FORMAT AS PARQUET;

-- Copy from S3
COPY orders FROM 's3://my-bucket/data/'
IAM_ROLE 'arn:aws:iam::123:role/RedshiftRole'
FORMAT AS PARQUET;
```

SECTION 6 — dbt (Data Build Tool)

6.1 Setup

```
pip install dbt-postgres   # or dbt-bigquery, dbt-snowflake, dbt-redshift

dbt init my_project        # creates project structure
cd my_project
dbt debug                  # test connection
dbt run                    # run all models
dbt test                   # run all tests
dbt docs generate          # generate documentation
dbt docs serve             # serve docs locally
```

6.2 profiles.yml (Connection Config)

```
# ~/.dbt/profiles.yml
my_project:
  target: dev
  outputs:
    dev:
      type: postgres
      host: localhost
      port: 5432
      user: "{{ env_var('DBT_USER') }}"
      password: "{{ env_var('DBT_PASSWORD') }}"
      dbname: analytics
      schema: dev_yourname
      threads: 4
    prod:
      type: postgres
      host: prod-warehouse.company.com
      port: 5432
      user: dbt_prod
      password: "{{ env_var('DBT_PROD_PASSWORD') }}"
      dbname: analytics
```

```
      schema: dbt_prod
      threads: 8
```

6.3 dbt Model Structure

```
-- models/staging/stg_orders.sql
-- Staging: 1-to-1 with source, just rename + light cleaning

{{ config(materialized='view') }}

WITH source AS (
    SELECT * FROM {{ source('postgres', 'orders') }}
),
renamed AS (
    SELECT
        id                          AS order_id,
        user_id,
        CAST(amount AS NUMERIC)     AS amount,
        status,
        created_at::TIMESTAMP       AS created_at,
        updated_at::TIMESTAMP       AS updated_at
    FROM source
    WHERE id IS NOT NULL
)
SELECT * FROM renamed

---

-- models/intermediate/int_orders_with_users.sql
-- Intermediate: business logic, joins

{{ config(materialized='view') }}

WITH orders AS (
    SELECT * FROM {{ ref('stg_orders') }}
),
users AS (
    SELECT * FROM {{ ref('stg_users') }}
)
SELECT
    o.order_id,
    o.user_id,
    o.amount,
```

```
    o.status,
    o.created_at,
    u.email,
    u.country,
    u.signup_date,
    DATEDIFF('day', u.signup_date, o.created_at) AS days_since_signup
FROM orders o
LEFT JOIN users u ON o.user_id = u.user_id

---

-- models/marts/fct_orders.sql
-- Fact table: final, production-ready

{{ config(
    materialized='incremental',
    unique_key='order_id',
    on_schema_change='fail'
) }}

WITH orders AS (
    SELECT * FROM {{ ref('int_orders_with_users') }}
    {% if is_incremental() %}
    WHERE created_at > (SELECT MAX(created_at) FROM {{ this }})
    {% endif %}
)
SELECT
    order_id,
    user_id,
    amount,
    status,
    country,
    created_at,
    DATE_TRUNC('month', created_at) AS order_month,
    CURRENT_TIMESTAMP AS _dbt_inserted_at
FROM orders
```

6.4 dbt Tests

```
# models/staging/schema.yml
version: 2

sources:
```

```yaml
  - name: postgres
    database: prod
    schema: public
    tables:
      - name: orders
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 24, period: hour}
        loaded_at_field: updated_at

models:
  - name: stg_orders
    description: "Cleaned orders from source system"
    columns:
      - name: order_id
        description: "Primary key"
        tests:
          - unique
          - not_null
      - name: amount
        tests:
          - not_null
          - dbt_utils.accepted_range:
              min_value: 0
              max_value: 1000000
      - name: status
        tests:
          - accepted_values:
              values: ['pending', 'completed', 'cancelled', 'refunded']
      - name: user_id
        tests:
          - not_null
          - relationships:
              to: ref('stg_users')
              field: user_id

  - name: fct_orders
    tests:
      - dbt_utils.recency:
          datepart: day
          field: created_at
          interval: 1
```

6.5 dbt Macros and Jinja

```jinja
-- macros/generate_schema_name.sql
{% macro generate_schema_name(custom_schema_name, node) -%}
    {%- set default_schema = target.schema -%}
    {%- if custom_schema_name is none -%}
        {{ default_schema }}
    {%- else -%}
        {{ custom_schema_name | trim }}
    {%- endif -%}
{%- endmacro %}

-- macros/cents_to_dollars.sql
{% macro cents_to_dollars(column_name) %}
    ({{ column_name }} / 100.0)::NUMERIC(10,2)
{% endmacro %}

-- Usage in model:
SELECT {{ cents_to_dollars('amount_cents') }} AS amount_dollars

-- Dynamic date spine (generate a row per date)
{{ dbt_utils.date_spine(
    datepart="day",
    start_date="cast('2020-01-01' as date)",
    end_date="cast('2024-12-31' as date)"
) }}
```

6.6 dbt Commands Reference

```bash
# Run specific model and all its dependencies (+)
dbt run --select stg_orders+
dbt run --select +fct_orders    # and all upstream
dbt run --select +fct_orders+   # upstream AND downstream

# Run specific tag
dbt run --select tag:daily

# Test specific model
dbt test --select stg_orders

# Compile only (don't execute)
dbt compile --select fct_orders

# Freshness check
dbt source freshness
```

```
# Show model DAG
dbt ls --select +fct_orders --output tree

# Run in production
dbt run --target prod
dbt run --target prod --full-refresh   # rebuild incremental from scratch
```

SECTION 7 — Apache Spark (PySpark)

7.1 PySpark Setup

```
# pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql import types as T

spark = SparkSession.builder \
    .appName("DataPipeline") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.driver.memory", "4g") \
    .config("spark.executor.memory", "8g") \
    .getOrCreate()

# Read data
df = spark.read.parquet("s3://bucket/data/")
df = spark.read.csv("data/file.csv", header=True, inferSchema=True)
df = spark.read.json("data/file.jsonl")

# Basic operations (lazy — nothing runs until action)
df.printSchema()        # column names + types
df.show(10)             # action: show first 10 rows
df.count()              # action: count rows
df.describe().show()    # statistics
```

7.2 PySpark Transformations

```python
from pyspark.sql import functions as F

# Select + rename
df = df.select(
    F.col("order_id"),
    F.col("user_id"),
    F.col("amount").alias("revenue"),
    F.to_date(F.col("created_at")).alias("order_date")
)

# Filter
df_active = df.filter(F.col("status") == "active")
df_high = df.filter(F.col("amount") > 1000)

# Add columns
df = df.withColumn("revenue_usd", F.col("amount") / 100)
df = df.withColumn("order_year", F.year(F.col("order_date")))
df = df.withColumn(
    "tier",
    F.when(F.col("amount") > 10000, "premium")
     .when(F.col("amount") > 1000, "standard")
     .otherwise("basic")
)

# GroupBy
summary = df.groupBy("region", "tier").agg(
    F.count("order_id").alias("order_count"),
    F.sum("amount").alias("total_revenue"),
    F.avg("amount").alias("avg_revenue"),
    F.countDistinct("user_id").alias("unique_users")
)

# Window functions
from pyspark.sql.window import Window

window = Window.partitionBy("user_id").orderBy("order_date")
df = df.withColumn("order_num", F.row_number().over(window))
df = df.withColumn("prev_order_amount", F.lag("amount", 1).over(window))
df = df.withColumn("cumulative_spend", F.sum("amount").over(
    window.rowsBetween(Window.unboundedPreceding, Window.currentRow)
))

# Join
df_users = spark.read.parquet("data/users/")
df_joined = df.join(df_users, on="user_id", how="left")
```

```
# Broadcast join (for small lookup tables)
from pyspark.sql.functions import broadcast
df_joined = df.join(broadcast(df_small_lookup), on="category_id")

# Repartition (for better parallelism before write)
df = df.repartition(100, "date")   # 100 partitions, sorted by date
df = df.coalesce(10)               # reduce partitions (no shuffle)
```

7.3 PySpark SQL

```
# Register as temp view and use SQL
df.createOrReplaceTempView("orders")

result = spark.sql("""
    SELECT
        region,
        DATE_FORMAT(order_date, 'yyyy-MM') AS month,
        COUNT(DISTINCT user_id) AS dau,
        SUM(amount) AS revenue,
        SUM(amount) / COUNT(*) AS aov
    FROM orders
    WHERE status = 'completed'
    GROUP BY 1, 2
    ORDER BY 2, 1
""")
result.show()
```

7.4 Writing Data

```
# Write to parquet (partitioned)
df.write \
    .mode("overwrite") \
    .partitionBy("year", "month") \
    .parquet("s3://bucket/output/orders/")

# Write to Delta Lake
df.write.format("delta").mode("overwrite").save("s3://bucket/delta/orders/"

# Write to single file (careful with large data)
```

```
df.coalesce(1).write.csv("output/result.csv", header=True, mode="overwrite"

# Write to database
df.write \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "schema.table") \
    .option("user", "user") \
    .option("password", "pass") \
    .mode("append") \
    .save()
```

7.5 Performance Tips

```
# 1. Cache frequently reused DataFrames
df.cache()      # lazy: cached on first action
df.persist()    # more control over storage level
df.unpersist()  # free memory

# 2. Avoid UDFs (use built-in F. functions)
# BAD:
def my_func(x):
    return x * 2
udf_func = F.udf(my_func, T.DoubleType())
df = df.withColumn("doubled", udf_func("amount"))

# GOOD (built-in, runs in JVM):
df = df.withColumn("doubled", F.col("amount") * 2)

# 3. Enable AQE (Adaptive Query Execution) - Spark 3+
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

# 4. Check query plan
df.explain(True)   # physical + logical plan

# 5. Skew handling
# If one partition key has many more rows (data skew), add salt
import random
df = df.withColumn("salt", (F.rand() * 100).cast("int"))
df = df.withColumn("salted_key", F.concat(F.col("skewed_key"), F.col("salt"
```

SECTION 8 — Orchestration with Airflow

8.1 Airflow Setup

```
pip install apache-airflow apache-airflow-providers-postgres \
            apache-airflow-providers-google apache-airflow-providers-amazon

# Init DB and create admin user
airflow db migrate
airflow users create --username admin --firstname Admin --lastname User \
    --role Admin --email admin@example.com --password admin

# Start webserver and scheduler
airflow webserver --port 8080 &
airflow scheduler &
```

8.2 Basic DAG

```
# dags/orders_etl.py
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator
from airflow.utils.dates import days_ago
from datetime import datetime, timedelta
import pandas as pd

default_args = {
    "owner": "data-team",
    "depends_on_past": False,
    "email": ["alerts@company.com"],
    "email_on_failure": True,
    "email_on_retry": False,
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
}

with DAG(
```

```
    dag_id="orders_daily_etl",
    default_args=default_args,
    description="Daily orders ETL pipeline",
    schedule_interval="0 6 * * *",   # cron: 6am daily
    start_date=datetime(2024, 1, 1),
    catchup=False,                    # don't backfill
    tags=["etl", "orders", "daily"],
    max_active_runs=1,                # one run at a time
) as dag:

    def extract(**context):
        """Extract orders from source DB."""
        execution_date = context["ds"]   # "2024-01-15"
        # Extract logic here
        print(f"Extracting for date: {execution_date}")
        # Push result to XCom
        context["ti"].xcom_push(key="row_count", value=1000)

    def transform(**context):
        row_count = context["ti"].xcom_pull(key="row_count", task_ids="extr
        print(f"Transforming {row_count} rows")

    def load(**context):
        print("Loading data")

    def validate(**context):
        print("Validating loaded data")

    t_extract = PythonOperator(task_id="extract", python_callable=extract)

    t_transform = PythonOperator(task_id="transform", python_callable=trans

    t_load = PythonOperator(task_id="load", python_callable=load)

    t_validate = PythonOperator(task_id="validate", python_callable=validat

    t_dbt_run = BashOperator(
        task_id="dbt_run",
        bash_command="cd /opt/dbt && dbt run --select tag:orders --target p
    )

    # Define task dependencies
    t_extract >> t_transform >> t_load >> t_validate >> t_dbt_run
```

8.3 Sensors and Branching

```
from airflow.sensors.filesystem import FileSensor
from airflow.sensors.sql import SqlSensor
from airflow.operators.python import BranchPythonOperator

# File sensor — wait for file to appear
wait_for_file = FileSensor(
    task_id="wait_for_file",
    filepath="/data/raw/orders_{{ ds }}.csv",
    poke_interval=300,    # check every 5 min
    timeout=3600,         # fail after 1 hour
    mode="reschedule"     # release worker slot while waiting
)

# SQL sensor — wait for data to appear
wait_for_data = SqlSensor(
    task_id="wait_for_data",
    conn_id="postgres_prod",
    sql="SELECT COUNT(*) FROM orders WHERE date = '{{ ds }}'",
    mode="reschedule",
    poke_interval=600
)

# Branching — conditional paths
def decide_path(**context):
    row_count = context["ti"].xcom_pull(key="row_count", task_ids="extract"
    if row_count > 0:
        return "transform"      # task_id to run
    else:
        return "send_empty_alert"

branch = BranchPythonOperator(
    task_id="check_data",
    python_callable=decide_path
)
```

8.4 Prefect (Modern Alternative to Airflow)

```
# pip install prefect
from prefect import flow, task, get_run_logger
```

```python
from prefect.tasks import task_input_hash
from datetime import timedelta

@task(retries=2, cache_key_fn=task_input_hash, cache_expiration=timedelta(h
def extract_data(start_date: str) -> pd.DataFrame:
    logger = get_run_logger()
    logger.info(f"Extracting from {start_date}")
    df = pd.read_csv(f"data/raw/orders_{start_date}.csv")
    return df

@task
def transform_data(df: pd.DataFrame) -> pd.DataFrame:
    logger = get_run_logger()
    logger.info(f"Transforming {len(df)} rows")
    df = df.dropna(subset=["order_id"])
    df["amount"] = df["amount"].astype(float)
    return df

@task
def load_data(df: pd.DataFrame, target: str):
    logger = get_run_logger()
    df.to_parquet(f"data/processed/{target}.parquet", index=False)
    logger.info(f"Loaded {len(df)} rows to {target}")

@flow(name="orders-etl", log_prints=True)
def orders_pipeline(start_date: str = "2024-01-01"):
    raw = extract_data(start_date)
    clean = transform_data(raw)
    load_data(clean, target=f"orders_{start_date}")

# Run
orders_pipeline(start_date="2024-06-01")

# Deploy and schedule via Prefect Cloud or self-hosted
```

SECTION 9 — Data Quality & Testing

9.1 Great Expectations

```
# pip install great-expectations
import great_expectations as gx
import pandas as pd

context = gx.get_context()

# Create data source
data_source = context.sources.add_pandas("my_source")
data_asset = data_source.add_dataframe_asset("orders")

batch_request = data_asset.build_batch_request(dataframe=df)

# Create expectation suite
suite = context.add_or_update_expectation_suite("orders_suite")

validator = context.get_validator(batch_request=batch_request, expectation_

# Add expectations
validator.expect_column_to_exist("order_id")
validator.expect_column_values_to_not_be_null("order_id")
validator.expect_column_values_to_be_unique("order_id")
validator.expect_column_values_to_not_be_null("user_id")
validator.expect_column_values_to_be_in_set("status", ["active", "completed
validator.expect_column_values_to_be_between("amount", min_value=0, max_val
validator.expect_column_values_to_match_regex("email", r"^[^@]+@[^@]+\.[^@]
validator.expect_table_row_count_to_be_between(min_value=1000, max_value=10
validator.expect_column_mean_to_be_between("amount", min_value=10, max_valu

# Save and validate
validator.save_expectation_suite()
results = validator.validate()
print(f"Success: {results.success}")
print(f"Failed expectations: {results.statistics['unsuccessful_expectations
```

9.2 Custom Data Quality Checks

```
import pandas as pd
import numpy as np
from dataclasses import dataclass
from typing import List, Callable, Any

@dataclass
```

```python
class QualityCheck:
    name: str
    passed: bool
    message: str
    severity: str = "error"  # "error" or "warning"

def run_quality_checks(df: pd.DataFrame) -> List[QualityCheck]:
    checks = []

    # Completeness
    for col in ["order_id", "user_id", "amount"]:
        null_pct = df[col].isnull().mean()
        checks.append(QualityCheck(
            name=f"{col}_not_null",
            passed=null_pct == 0,
            message=f"{col}: {null_pct:.1%} null values"
        ))

    # Uniqueness
    dup_count = df["order_id"].duplicated().sum()
    checks.append(QualityCheck(
        name="order_id_unique",
        passed=dup_count == 0,
        message=f"order_id: {dup_count} duplicates found"
    ))

    # Range
    neg_amount = (df["amount"] < 0).sum()
    checks.append(QualityCheck(
        name="amount_positive",
        passed=neg_amount == 0,
        message=f"amount: {neg_amount} negative values"
    ))

    # Freshness
    latest = df["created_at"].max()
    hours_old = (pd.Timestamp.now() - pd.to_datetime(latest)).total_seconds
    checks.append(QualityCheck(
        name="data_freshness",
        passed=hours_old < 25,
        message=f"Latest record is {hours_old:.1f} hours old",
        severity="warning" if hours_old < 48 else "error"
    ))

    # Volume
    checks.append(QualityCheck(
```

```
        name="minimum_rows",
        passed=len(df) >= 100,
        message=f"Table has {len(df)} rows (minimum: 100)"
    ))

    return checks

def print_quality_report(checks: List[QualityCheck]):
    errors = [c for c in checks if not c.passed and c.severity == "error"]
    warnings = [c for c in checks if not c.passed and c.severity == "warnin
    passed = [c for c in checks if c.passed]

    print(f"\n{'='*50}")
    print(f"DATA QUALITY REPORT")
    print(f"{'='*50}")
    print(f"✅ Passed:   {len(passed)}")
    print(f"⚠️  Warnings: {len(warnings)}")
    print(f"❌ Errors:   {len(errors)}")

    for c in checks:
        icon = "✅" if c.passed else ("⚠️" if c.severity == "warning" else
        print(f"  {icon} {c.name}: {c.message}")

    if errors:
        raise ValueError(f"Data quality failed: {len(errors)} errors")

checks = run_quality_checks(df)
print_quality_report(checks)
```

9.3 Testing ETL with pytest

```
# tests/test_transform.py
import pytest
import pandas as pd
import numpy as np
from src.transform import clean_orders, compute_metrics

@pytest.fixture
def sample_orders():
    return pd.DataFrame({
        "order_id": [1, 2, 3, None, 5],
        "user_id": [10, 20, 30, 40, 50],
        "amount": [100.0, -5.0, 200.0, 150.0, 300.0],
```

```
        "status": ["completed", "completed", "cancelled", "completed", "com
        "created_at": pd.date_range("2024-01-01", periods=5)
    })

def test_removes_null_order_ids(sample_orders):
    result = clean_orders(sample_orders)
    assert result["order_id"].isnull().sum() == 0

def test_removes_negative_amounts(sample_orders):
    result = clean_orders(sample_orders)
    assert (result["amount"] < 0).sum() == 0

def test_output_columns(sample_orders):
    result = clean_orders(sample_orders)
    expected_cols = ["order_id", "user_id", "amount", "status", "created_at
    assert all(col in result.columns for col in expected_cols)

def test_metrics_calculation(sample_orders):
    clean = clean_orders(sample_orders)
    metrics = compute_metrics(clean)
    assert metrics["total_revenue"] == clean["amount"].sum()
    assert metrics["order_count"] == len(clean)

def test_empty_dataframe_handling():
    empty_df = pd.DataFrame(columns=["order_id", "user_id", "amount"])
    result = clean_orders(empty_df)
    assert len(result) == 0

# Run with: pytest tests/ -v --tb=short
```

SECTION 10 — Business Intelligence & Dashboards

10.1 Streamlit (Python Dashboards)

```
# pip install streamlit plotly
# Run with: streamlit run dashboard.py
```

```python
import streamlit as st
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from datetime import datetime, timedelta

st.set_page_config(page_title="Sales Dashboard", layout="wide", page_icon="

# Sidebar filters
st.sidebar.header("Filters")
date_range = st.sidebar.date_input(
    "Date Range",
    value=[datetime.now() - timedelta(days=30), datetime.now()]
)
regions = st.sidebar.multiselect("Regions", ["North", "South", "East", "Wes

@st.cache_data(ttl=3600)  # cache for 1 hour
def load_data():
    # Replace with actual DB query
    dates = pd.date_range("2024-01-01", periods=180, freq="D")
    df = pd.DataFrame({
        "date": dates,
        "revenue": (10000 + pd.Series(range(180)) * 50 + pd.Series([abs(i)
        "orders": pd.Series(range(180)) + 100,
        "region": ["North", "South", "East", "West"] * 45
    })
    return df

df = load_data()
df_filtered = df[df["region"].isin(regions)]

# KPI metrics
col1, col2, col3, col4 = st.columns(4)
col1.metric("Total Revenue", f"${df_filtered['revenue'].sum():,.0f}", delta
col2.metric("Total Orders", f"{df_filtered['orders'].sum():,}", delta="+8%"
col3.metric("Avg Order Value", f"${df_filtered['revenue'].sum()/df_filtered
col4.metric("Active Regions", len(regions))

st.divider()

# Charts
col_left, col_right = st.columns(2)

with col_left:
    st.subheader("Revenue Over Time")
```

```
    fig = px.line(df_filtered, x="date", y="revenue", color="region",
                  template="plotly_white")
    st.plotly_chart(fig, use_container_width=True)

with col_right:
    st.subheader("Revenue by Region")
    region_total = df_filtered.groupby("region")["revenue"].sum().reset_ind
    fig2 = px.bar(region_total, x="region", y="revenue",
                  color="region", template="plotly_white")
    st.plotly_chart(fig2, use_container_width=True)

# Data table
with st.expander("📋 Raw Data"):
    st.dataframe(df_filtered, use_container_width=True)
    csv = df_filtered.to_csv(index=False).encode("utf-8")
    st.download_button("Download CSV", csv, "data.csv", "text/csv")
```

10.2 Plotly Charts Cheatsheet

```
import plotly.express as px
import plotly.graph_objects as go

# Line chart
fig = px.line(df, x="date", y="revenue", color="region",
              title="Revenue by Region", template="plotly_white")
fig.update_xaxes(rangeslider_visible=True)
fig.show()

# Bar chart (grouped and stacked)
px.bar(df, x="month", y="revenue", color="product",
       barmode="group",   # or "stack"
       template="plotly_white")

# Scatter with size and color
px.scatter(df, x="spend", y="revenue", color="region",
           size="orders", hover_data=["customer_id"],
           trendline="ols")  # OLS regression line

# Heatmap
px.imshow(corr_matrix, color_continuous_scale="RdBu_r", zmin=-1, zmax=1)

# Histogram
px.histogram(df, x="amount", nbins=50, color="status",
```

```
            marginal="box")  # box plot on top

# Pie
px.pie(df, values="revenue", names="region", hole=0.3)  # hole=0.3 for donu

# Treemap
px.treemap(df, path=["region", "product"], values="revenue", color="margin"

# Funnel
px.funnel(df, x="count", y="stage")

# Multiple y-axes
fig = go.Figure()
fig.add_trace(go.Bar(x=df["month"], y=df["orders"], name="Orders", yaxis="y
fig.add_trace(go.Scatter(x=df["month"], y=df["revenue"], name="Revenue", ya
fig.update_layout(yaxis2=dict(overlaying="y", side="right"))
```

10.3 Key Metrics Every Analyst Builds

```
import pandas as pd

# DAU / MAU / WAU
def compute_engagement_metrics(events_df):
    """
    events_df must have: user_id, event_date
    """
    events_df["event_date"] = pd.to_datetime(events_df["event_date"])
    events_df["week"] = events_df["event_date"].dt.to_period("W")
    events_df["month"] = events_df["event_date"].dt.to_period("M")

    dau = events_df.groupby("event_date")["user_id"].nunique().rename("DAU"
    wau = events_df.groupby("week")["user_id"].nunique().rename("WAU")
    mau = events_df.groupby("month")["user_id"].nunique().rename("MAU")

    return dau, wau, mau

# Cohort Retention
def compute_cohort_retention(df):
    df["signup_month"] = pd.to_datetime(df["signup_date"]).dt.to_period("M"
    df["activity_month"] = pd.to_datetime(df["activity_date"]).dt.to_period
    df["months_since_signup"] = (df["activity_month"] - df["signup_month"])

    cohort_size = df.groupby("signup_month")["user_id"].nunique().rename("c
```

```
    retention = df.groupby(["signup_month", "months_since_signup"])["user_i
    retention = retention.reset_index()
    retention = retention.merge(cohort_size.reset_index(), on="signup_month
    retention["retention_rate"] = retention["user_id"] / retention["cohort_

    pivot = retention.pivot(index="signup_month", columns="months_since_sig
    return pivot

# Customer LTV
def compute_ltv(orders_df, periods=12):
    orders_df["date"] = pd.to_datetime(orders_df["date"])
    monthly = orders_df.groupby([
        "user_id",
        orders_df["date"].dt.to_period("M")
    ])["amount"].sum().reset_index()

    ltv = monthly.groupby("user_id")["amount"].agg(
        total_ltv="sum",
        avg_monthly="mean",
        months_active="count"
    ).reset_index()

    return ltv

# Churn Rate
def compute_churn(df, lookback_days=30):
    cutoff = pd.Timestamp.now() - pd.Timedelta(days=lookback_days)
    active_last_period = df[df["last_active"] >= cutoff - pd.Timedelta(days
    active_this_period = df[df["last_active"] >= cutoff]["user_id"]
    churned = set(active_last_period) - set(active_this_period)
    return len(churned) / len(active_last_period) if len(active_last_period
```

SECTION 11 — Statistics for Data Analysts

11.1 Descriptive Statistics

```
import pandas as pd
import numpy as np
```

```python
from scipy import stats

df = pd.read_csv("data/sales.csv")

# Central tendency
print("Mean:", df["revenue"].mean())
print("Median:", df["revenue"].median())
print("Mode:", df["revenue"].mode()[0])

# Spread
print("Std Dev:", df["revenue"].std())
print("Variance:", df["revenue"].var())
print("IQR:", df["revenue"].quantile(0.75) - df["revenue"].quantile(0.25))
print("Range:", df["revenue"].max() - df["revenue"].min())

# Distribution shape
print("Skewness:", df["revenue"].skew())   # >0 = right-skewed, <0 = left-s
print("Kurtosis:", df["revenue"].kurtosis())  # >0 = heavy tails, <0 = ligh

# Percentiles
print(df["revenue"].describe(percentiles=[0.05, 0.25, 0.5, 0.75, 0.95]))
```

11.2 Hypothesis Testing

```python
from scipy import stats
import numpy as np

# t-test: compare means of two groups
group_a = df[df["group"] == "A"]["revenue"]
group_b = df[df["group"] == "B"]["revenue"]

# Independent t-test (two different groups)
t_stat, p_value = stats.ttest_ind(group_a, group_b, equal_var=False)  # Wel
print(f"t-statistic: {t_stat:.4f}, p-value: {p_value:.4f}")
print("Significant (p < 0.05):", p_value < 0.05)

# Paired t-test (same users, before/after)
t_stat, p_value = stats.ttest_rel(before_values, after_values)

# Mann-Whitney U (non-parametric alternative to t-test)
stat, p = stats.mannwhitneyu(group_a, group_b, alternative="two-sided")

# Chi-square test (categorical vs categorical)
```

```
contingency = pd.crosstab(df["gender"], df["converted"])
chi2, p, dof, expected = stats.chi2_contingency(contingency)
print(f"Chi2: {chi2:.4f}, p-value: {p:.4f}, dof: {dof}")

# Normality test
stat, p = stats.shapiro(group_a[:50])  # Shapiro-Wilk (best for n < 50)
stat, p = stats.normaltest(group_a)   # D'Agostino-Pearson (n > 50)
print(f"Normal distribution (p > 0.05): {p > 0.05}")

# ANOVA: compare means across 3+ groups
groups = [df[df["segment"] == s]["revenue"] for s in df["segment"].unique()
f_stat, p_value = stats.f_oneway(*groups)
print(f"ANOVA F: {f_stat:.4f}, p: {p_value:.4f}")
```

11.3 Confidence Intervals

```
import numpy as np
from scipy import stats

def confidence_interval(data, confidence=0.95):
    n = len(data)
    mean = np.mean(data)
    se = stats.sem(data)   # standard error
    ci = stats.t.interval(confidence, df=n-1, loc=mean, scale=se)
    return mean, ci[0], ci[1]

mean, lower, upper = confidence_interval(df["revenue"])
print(f"Mean: {mean:.2f}, 95% CI: [{lower:.2f}, {upper:.2f}]")

# Bootstrap CI (distribution-free)
def bootstrap_ci(data, statistic=np.mean, n_boot=10000, ci=0.95):
    boot_stats = [statistic(np.random.choice(data, size=len(data), replace=
                  for _ in range(n_boot)]
    alpha = (1 - ci) / 2
    return np.percentile(boot_stats, [alpha*100, (1-alpha)*100])

lower, upper = bootstrap_ci(df["revenue"].values)
print(f"Bootstrap 95% CI: [{lower:.2f}, {upper:.2f}]")
```

11.4 Correlation Analysis

```python
# Pearson (linear, numeric-numeric)
r, p = stats.pearsonr(df["ad_spend"], df["revenue"])
print(f"Pearson r={r:.4f}, p={p:.4f}")

# Spearman (monotonic, handles outliers better)
r, p = stats.spearmanr(df["ad_spend"], df["revenue"])
print(f"Spearman r={r:.4f}, p={p:.4f}")

# Point-biserial (binary vs continuous)
r, p = stats.pointbiserialr(df["converted"], df["revenue"])

# Cramér's V (categorical vs categorical)
def cramers_v(x, y):
    ct = pd.crosstab(x, y)
    chi2 = stats.chi2_contingency(ct)[0]
    n = ct.sum().sum()
    return np.sqrt(chi2 / (n * (min(ct.shape) - 1)))

v = cramers_v(df["gender"], df["segment"])
print(f"Cramér's V: {v:.4f}")
```

SECTION 12 — A/B Testing & Experimentation

12.1 Sample Size Calculation

```python
from scipy import stats
import numpy as np

def calculate_sample_size(
    baseline_rate: float,
    min_detectable_effect: float,
    alpha: float = 0.05,
    power: float = 0.80
) -> int:
    """
    Calculate required sample size per group for a proportion A/B test.
```

```
    baseline_rate: current conversion rate (e.g., 0.05 for 5%)
    min_detectable_effect: smallest change worth detecting (e.g., 0.01 = 1p
    alpha: significance level (Type I error rate)
    power: 1 - beta (Type II error rate)
    """
    p1 = baseline_rate
    p2 = baseline_rate + min_detectable_effect

    z_alpha = stats.norm.ppf(1 - alpha / 2)   # two-tailed
    z_beta = stats.norm.ppf(power)

    p_bar = (p1 + p2) / 2

    n = (z_alpha * np.sqrt(2 * p_bar * (1 - p_bar)) +
         z_beta * np.sqrt(p1 * (1-p1) + p2 * (1-p2)))**2 / (p2 - p1)**2

    return int(np.ceil(n))

n = calculate_sample_size(baseline_rate=0.05, min_detectable_effect=0.01)
print(f"Required sample size per group: {n:,}")
print(f"Total users needed: {n*2:,}")
```

12.2 Running the Test

```
import pandas as pd
import numpy as np
from scipy import stats

# Load experiment data
# Expected columns: user_id, group (control/treatment), converted (0/1), re
exp_df = pd.read_csv("data/experiment_results.csv")

# Sanity checks
print("Group distribution:")
print(exp_df["group"].value_counts())
print("\nConversion rates:")
print(exp_df.groupby("group")["converted"].mean())

# Check for SRM (Sample Ratio Mismatch)
control_n = (exp_df["group"] == "control").sum()
treatment_n = (exp_df["group"] == "treatment").sum()
total_n = len(exp_df)
```

```
expected_n = total_n / 2
chi2_srm = ((control_n - expected_n)**2 / expected_n +
            (treatment_n - expected_n)**2 / expected_n)
p_srm = 1 - stats.chi2.cdf(chi2_srm, df=1)
if p_srm < 0.01:
    print(f"⚠️ SRM DETECTED! p={p_srm:.4f} — randomization may be broken")

# Two-proportion z-test (conversion rate)
control = exp_df[exp_df["group"] == "control"]["converted"]
treatment = exp_df[exp_df["group"] == "treatment"]["converted"]

n_c, n_t = len(control), len(treatment)
conv_c, conv_t = control.mean(), treatment.mean()

# Pooled z-test
p_pool = (control.sum() + treatment.sum()) / (n_c + n_t)
se = np.sqrt(p_pool * (1-p_pool) * (1/n_c + 1/n_t))
z_stat = (conv_t - conv_c) / se
p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

print(f"\n{'='*40}")
print(f"A/B TEST RESULTS")
print(f"{'='*40}")
print(f"Control conversion:   {conv_c:.4f} (n={n_c:,})")
print(f"Treatment conversion: {conv_t:.4f} (n={n_t:,})")
print(f"Relative lift: {(conv_t - conv_c) / conv_c:.2%}")
print(f"Absolute lift: {conv_t - conv_c:.4f} ({(conv_t - conv_c)*100:.2f} p
print(f"Z-statistic: {z_stat:.4f}")
print(f"P-value: {p_value:.6f}")
print(f"Significant: {'YES ✅' if p_value < 0.05 else 'NO ❌'}")
```

12.3 Confidence Intervals for A/B Test

```
# CI for the lift
def ab_test_ci(conv_c, n_c, conv_t, n_t, alpha=0.05):
    diff = conv_t - conv_c
    se = np.sqrt(conv_c*(1-conv_c)/n_c + conv_t*(1-conv_t)/n_t)
    z = stats.norm.ppf(1 - alpha/2)
    ci_low = diff - z * se
    ci_high = diff + z * se
    return diff, ci_low, ci_high

diff, ci_low, ci_high = ab_test_ci(conv_c, n_c, conv_t, n_t)
```

```
print(f"Absolute lift: {diff:.4f} [{ci_low:.4f}, {ci_high:.4f}]")

# Revenue t-test (continuous metric)
rev_c = exp_df[exp_df["group"] == "control"]["revenue"]
rev_t = exp_df[exp_df["group"] == "treatment"]["revenue"]
t_stat, p_val = stats.ttest_ind(rev_t, rev_c, equal_var=False)
print(f"\nRevenue per user — Control: ${rev_c.mean():.2f}, Treatment: ${rev
print(f"t={t_stat:.4f}, p={p_val:.4f}")
```

12.4 Multiple Testing Correction

```
from statsmodels.stats.multitest import multipletests

# If testing multiple metrics or segments
p_values = [0.03, 0.04, 0.01, 0.06, 0.02, 0.08]  # p-values from multiple t

# Bonferroni (very conservative)
reject_bonf, pvals_corrected, _, _ = multipletests(p_values, method="bonfer

# Benjamini-Hochberg (FDR control — recommended)
reject_bh, pvals_corrected_bh, _, _ = multipletests(p_values, method="fdr_b

for i, (p, r_b, r_bh) in enumerate(zip(p_values, reject_bonf, reject_bh)):
    print(f"Test {i+1}: p={p:.3f} | Bonferroni: {'✅' if r_b else '❌'} | I
```

SECTION 13 — Time Series Analysis

13.1 Time Series Basics

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
```

```
# Load and set time index
df = pd.read_csv("data/daily_revenue.csv", parse_dates=["date"], index_col=
ts = df["revenue"]

# Resample to different frequencies
weekly = ts.resample("W").sum()
monthly = ts.resample("ME").mean()

# Decompose: Trend + Seasonality + Residual
decomposition = seasonal_decompose(ts, model="additive", period=7)  # weekl
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 8))
decomposition.observed.plot(ax=ax1, title="Observed")
decomposition.trend.plot(ax=ax2, title="Trend")
decomposition.seasonal.plot(ax=ax3, title="Seasonal")
decomposition.resid.plot(ax=ax4, title="Residual")
plt.tight_layout()
plt.show()

# Stationarity test (ADF)
result = adfuller(ts.dropna())
print(f"ADF Statistic: {result[0]:.4f}")
print(f"P-value: {result[1]:.4f}")
print(f"Stationary: {result[1] < 0.05}")

# Make stationary via differencing
ts_diff = ts.diff().dropna()
ts_log_diff = np.log(ts).diff().dropna()
```

13.2 Rolling Statistics

```
# Rolling stats (moving window)
ts_df = ts.to_frame("revenue")
ts_df["rolling_mean_7d"] = ts.rolling(window=7).mean()
ts_df["rolling_std_7d"] = ts.rolling(window=7).std()
ts_df["rolling_mean_30d"] = ts.rolling(window=30).mean()
ts_df["ewm_7d"] = ts.ewm(span=7).mean()   # exponential weighted

# Lag features for ML
for lag in [1, 7, 14, 30]:
    ts_df[f"lag_{lag}"] = ts.shift(lag)

# Rolling on grouped data
df["rolling_user_avg"] = df.groupby("user_id")["amount"].transform(
```

```
    lambda x: x.rolling(7, min_periods=1).mean()
)
```

13.3 Prophet Forecasting

```
# pip install prophet
from prophet import Prophet
import pandas as pd

# Prophet expects df with columns 'ds' (datetime) and 'y' (value)
prophet_df = df.reset_index().rename(columns={"date": "ds", "revenue": "y"}

model = Prophet(
    yearly_seasonality=True,
    weekly_seasonality=True,
    daily_seasonality=False,
    seasonality_mode="multiplicative",  # or "additive"
    changepoint_prior_scale=0.05,       # flexibility of trend
    interval_width=0.95                 # confidence interval
)

# Add holidays
from prophet.make_holidays import make_holidays_df
holidays = make_holidays_df(year_list=[2023, 2024], country="US")
model = Prophet(holidays=holidays)

# Add custom seasonality
model.add_seasonality(name="monthly", period=30.5, fourier_order=5)

# Add regressor (external variable)
model.add_regressor("promo_flag")
prophet_df["promo_flag"] = 0  # set your actual promo flags

model.fit(prophet_df)

# Forecast 90 days
future = model.make_future_dataframe(periods=90, freq="D")
future["promo_flag"] = 0   # set regressors for future
forecast = model.predict(future)

# Components
fig1 = model.plot(forecast)
fig2 = model.plot_components(forecast)
```

```
plt.show()

# Evaluate
from prophet.diagnostics import cross_validation, performance_metrics
df_cv = cross_validation(model, initial="365 days", period="30 days", horiz
df_perf = performance_metrics(df_cv)
print(df_perf[["horizon", "mae", "rmse", "mape"]].tail(10))
```

13.4 ARIMA

```
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import warnings
warnings.filterwarnings("ignore")

# Check ACF and PACF to determine p, d, q
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_acf(ts.dropna(), lags=30, ax=axes[0])
plot_pacf(ts.dropna(), lags=30, ax=axes[1])
plt.show()

# Fit ARIMA(p=1, d=1, q=1)
model = ARIMA(ts, order=(1, 1, 1))
result = model.fit()
print(result.summary())

# Forecast
forecast = result.forecast(steps=30)
conf_int = result.get_forecast(steps=30).conf_int()

plt.figure(figsize=(12, 5))
plt.plot(ts, label="Observed")
plt.plot(forecast.index, forecast, label="Forecast", color="red")
plt.fill_between(conf_int.index, conf_int.iloc[:, 0], conf_int.iloc[:, 1],
                 alpha=0.3, color="red")
plt.legend()
plt.show()
```

SECTION 14 — Data Modeling & Warehouse Design

14.1 Kimball Dimensional Modeling

```
-- FACT TABLE: Numeric measures + foreign keys to dimensions
CREATE TABLE fct_orders (
    order_sk        BIGINT PRIMARY KEY,        -- surrogate key
    order_id        VARCHAR(50) NOT NULL,       -- natural/business key
    user_sk         BIGINT REFERENCES dim_users(user_sk),
    product_sk      BIGINT REFERENCES dim_products(product_sk),
    date_sk         INT REFERENCES dim_date(date_sk),

    -- Metrics (additive measures)
    amount          DECIMAL(12, 2),
    quantity        INT,
    discount_amount DECIMAL(12, 2),
    net_revenue     DECIMAL(12, 2),

    -- Audit
    _loaded_at      TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    _source         VARCHAR(50)
);

-- DIMENSION TABLE: Descriptive attributes
CREATE TABLE dim_users (
    user_sk         BIGINT PRIMARY KEY,         -- surrogate key
    user_id         VARCHAR(50) NOT NULL,        -- natural key
    email           VARCHAR(255),
    first_name      VARCHAR(100),
    country         VARCHAR(50),
    signup_date     DATE,

    -- SCD Type 2 fields
    valid_from      DATE NOT NULL,
    valid_to        DATE,
    is_current      BOOLEAN DEFAULT TRUE
);

-- DATE DIMENSION (pre-populate)
CREATE TABLE dim_date (
```

```
    date_sk         INT PRIMARY KEY,      -- YYYYMMDD
    date            DATE,
    year            INT,
    quarter         INT,
    month           INT,
    month_name      VARCHAR(20),
    week            INT,
    day_of_week     INT,
    day_name        VARCHAR(20),
    is_weekend      BOOLEAN,
    is_holiday      BOOLEAN
);

-- Populate date dimension
INSERT INTO dim_date
SELECT
    TO_CHAR(d, 'YYYYMMDD')::INT AS date_sk,
    d AS date,
    EXTRACT(YEAR FROM d) AS year,
    EXTRACT(QUARTER FROM d) AS quarter,
    EXTRACT(MONTH FROM d) AS month,
    TO_CHAR(d, 'Month') AS month_name,
    EXTRACT(WEEK FROM d) AS week,
    EXTRACT(DOW FROM d) AS day_of_week,
    TO_CHAR(d, 'Day') AS day_name,
    EXTRACT(DOW FROM d) IN (0, 6) AS is_weekend,
    FALSE AS is_holiday
FROM generate_series('2020-01-01'::DATE, '2030-12-31'::DATE, '1 day') AS d;
```

14.2 SCD Type 2 (Slowly Changing Dimensions)

```
-- SCD Type 2: Track historical changes with valid_from/valid_to

-- When user changes email, old record is closed and new one is opened
-- Step 1: Close old record
UPDATE dim_users
SET valid_to = CURRENT_DATE - 1, is_current = FALSE
WHERE user_id = '123' AND is_current = TRUE;

-- Step 2: Insert new record
INSERT INTO dim_users (user_sk, user_id, email, country, valid_from, valid_
VALUES (nextval('dim_users_seq'), '123', 'newemail@example.com', 'US', CURR
```

```sql
-- Query: get current state
SELECT * FROM dim_users WHERE is_current = TRUE;

-- Query: get historical state at specific date
SELECT * FROM dim_users
WHERE user_id = '123'
AND valid_from <= '2023-06-01'
AND (valid_to IS NULL OR valid_to >= '2023-06-01');
```

14.3 Data Vault Modeling

```sql
-- Data Vault 2.0: Hub, Link, Satellite

-- HUB: business key + metadata
CREATE TABLE hub_customer (
    customer_hk     CHAR(32) PRIMARY KEY,    -- MD5 of business key
    customer_id     VARCHAR(50) NOT NULL,
    load_date       TIMESTAMP NOT NULL,
    record_source   VARCHAR(100) NOT NULL
);

-- LINK: relationship between hubs
CREATE TABLE lnk_customer_order (
    link_hk         CHAR(32) PRIMARY KEY,
    customer_hk     CHAR(32) REFERENCES hub_customer,
    order_hk        CHAR(32) REFERENCES hub_order,
    load_date       TIMESTAMP NOT NULL,
    record_source   VARCHAR(100) NOT NULL
);

-- SATELLITE: descriptive attributes (versioned)
CREATE TABLE sat_customer_profile (
    customer_hk     CHAR(32) REFERENCES hub_customer,
    load_date       TIMESTAMP NOT NULL,
    load_end_date   TIMESTAMP,
    record_source   VARCHAR(100),
    hash_diff       CHAR(32),    -- MD5 of all attributes to detect changes

    email           VARCHAR(255),
    country         VARCHAR(50),
    plan            VARCHAR(50),
```

```
    PRIMARY KEY (customer_hk, load_date)
);
```

SECTION 15 — APIs, Web Scraping & Data Collection

15.1 REST API Integration

```
import requests
import pandas as pd
import time
from functools import wraps
from typing import Optional

def retry_on_error(max_retries=3, delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except requests.exceptions.RequestException as e:
                    if attempt == max_retries - 1:
                        raise
                    wait = delay * (2 ** attempt)  # exponential backoff
                    print(f"Attempt {attempt+1} failed: {e}. Retrying in {w
                    time.sleep(wait)
        return wrapper
    return decorator

class APIClient:
    def __init__(self, base_url: str, api_key: str, rate_limit_rpm: int = 6
        self.base_url = base_url
        self.headers = {
            "Authorization": f"Bearer {api_key}",
            "Content-Type": "application/json"
        }
```

```
        self.min_interval = 60 / rate_limit_rpm
        self._last_request = 0

    def _rate_limit(self):
        elapsed = time.time() - self._last_request
        if elapsed < self.min_interval:
            time.sleep(self.min_interval - elapsed)
        self._last_request = time.time()

    @retry_on_error(max_retries=3, delay=2)
    def get(self, endpoint: str, params: dict = None) -> dict:
        self._rate_limit()
        r = requests.get(f"{self.base_url}/{endpoint}",
                         headers=self.headers, params=params, timeout=30)
        r.raise_for_status()
        return r.json()

    def get_paginated(self, endpoint: str, page_key: str = "page") -> list:
        results = []
        page = 1
        while True:
            data = self.get(endpoint, params={page_key: page, "limit": 100}
            items = data.get("data", data if isinstance(data, list) else []
            if not items:
                break
            results.extend(items)
            page += 1
        return results

# Example: Fetch from Stripe API
client = APIClient("https://api.stripe.com/v1", api_key="sk_live_...")
charges = client.get_paginated("charges")
df = pd.DataFrame(charges)
```

15.2 Web Scraping with BeautifulSoup

```
# pip install requests beautifulsoup4 lxml
import requests
from bs4 import BeautifulSoup
import pandas as pd
import time

def scrape_table(url: str, table_index: int = 0) -> pd.DataFrame:
```

```
    headers = {"User-Agent": "Mozilla/5.0"}
    r = requests.get(url, headers=headers, timeout=15)
    r.raise_for_status()
    soup = BeautifulSoup(r.content, "lxml")
    tables = soup.find_all("table")
    if not tables:
        raise ValueError("No tables found")
    return pd.read_html(str(tables[table_index]))[0]

def scrape_page(url: str) -> dict:
    headers = {"User-Agent": "Mozilla/5.0 (compatible; DataScraper/1.0)"}
    r = requests.get(url, headers=headers, timeout=15)
    r.raise_for_status()
    soup = BeautifulSoup(r.content, "lxml")

    return {
        "title": soup.find("h1").text.strip() if soup.find("h1") else None,
        "price": soup.find(class_="price").text.strip() if soup.find(class_
        "description": soup.find("p", class_="description").text if soup.fi
        "links": [a["href"] for a in soup.find_all("a", href=True)]
    }

# Scrape multiple pages with politeness
urls = ["https://example.com/products/1", "https://example.com/products/2"]
results = []
for url in urls:
    try:
        data = scrape_page(url)
        data["url"] = url
        results.append(data)
        time.sleep(1)  # be polite — don't hammer the server
    except Exception as e:
        print(f"Failed {url}: {e}")

df = pd.DataFrame(results)
```

15.3 Selenium for Dynamic Pages

```
# pip install selenium webdriver-manager
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
```

```python
from selenium.webdriver.chrome.options import Options

options = Options()
options.add_argument("--headless")   # run without browser window
options.add_argument("--no-sandbox")
options.add_argument("--disable-dev-shm-usage")

driver = webdriver.Chrome(options=options)

try:
    driver.get("https://example.com/login")

    # Fill form
    WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "email"))
    )
    driver.find_element(By.ID, "email").send_keys("user@example.com")
    driver.find_element(By.ID, "password").send_keys("password")
    driver.find_element(By.XPATH, "//button[@type='submit']").click()

    # Wait for page load
    WebDriverWait(driver, 10).until(EC.url_contains("dashboard"))

    # Extract data
    elements = driver.find_elements(By.CLASS_NAME, "data-row")
    data = [{"text": el.text, "attr": el.get_attribute("data-id")} for el i

    df = pd.DataFrame(data)
finally:
    driver.quit()
```

15.4 Playwright (Better than Selenium in 2026)

```python
# pip install playwright
# playwright install chromium   ← run once to download browser

from playwright.sync_api import sync_playwright
import pandas as pd

def scrape_with_playwright(url: str) -> list[dict]:
    results = []
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True)
```

```
        context = browser.new_context(
            user_agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebK
            viewport={"width": 1280, "height": 800}
        )
        page = context.new_page()

        # Block images/fonts to speed up scraping
        page.route("**/*.{png,jpg,jpeg,gif,svg,woff,woff2}", lambda r: r.ab

        page.goto(url, wait_until="networkidle", timeout=30000)

        # Wait for specific element
        page.wait_for_selector(".product-card", timeout=10000)

        # Extract data
        cards = page.query_selector_all(".product-card")
        for card in cards:
            results.append({
                "title": card.query_selector("h2").inner_text() if card.que
                "price": card.query_selector(".price").inner_text() if card
                "rating": card.get_attribute("data-rating"),
                "url": card.query_selector("a").get_attribute("href") if ca
            })

        browser.close()
    return results

# Async Playwright (faster for multiple pages)
import asyncio
from playwright.async_api import async_playwright

async def scrape_many(urls: list[str]) -> list[dict]:
    results = []
    async with async_playwright() as p:
        browser = await p.chromium.launch(headless=True)
        # Scrape pages concurrently
        async def scrape_one(url):
            page = await browser.new_page()
            await page.goto(url, wait_until="domcontentloaded")
            title = await page.title()
            await page.close()
            return {"url": url, "title": title}

        tasks = [scrape_one(url) for url in urls]
        results = await asyncio.gather(*tasks)
        await browser.close()
```

```
    return results

# Run async
urls = ["https://example.com/page/1", "https://example.com/page/2"]
results = asyncio.run(scrape_many(urls))
df = pd.DataFrame(results)
```

15.5 Scrapy (Production-Grade Crawler)

```
# pip install scrapy
# scrapy startproject myproject
# scrapy genspider products example.com

# myproject/spiders/products_spider.py
import scrapy
import pandas as pd

class ProductSpider(scrapy.Spider):
    name = "products"
    start_urls = ["https://example.com/products"]
    custom_settings = {
        "DOWNLOAD_DELAY": 1,          # 1 second between requests
        "CONCURRENT_REQUESTS": 4,
        "ROBOTSTXT_OBEY": True,        # respect robots.txt
        "FEEDS": {"data/products.jsonl": {"format": "jsonlines"}},  # auto-
        "USER_AGENT": "MyBot/1.0 (research purposes)"
    }

    def parse(self, response):
        # Extract all product links
        for href in response.css("a.product-link::attr(href)").getall():
            yield response.follow(href, callback=self.parse_product)

        # Follow pagination
        next_page = response.css("a.next-page::attr(href)").get()
        if next_page:
            yield response.follow(next_page, callback=self.parse)

    def parse_product(self, response):
        yield {
            "url": response.url,
            "title": response.css("h1::text").get("").strip(),
            "price": response.css(".price::text").get("").strip(),
```

```
            "description": " ".join(response.css(".description p::text").ge
            "images": response.css("img.product-image::attr(src)").getall()
            "breadcrumb": response.css(".breadcrumb a::text").getall()
        }

# Run from CLI:
# scrapy crawl products
# scrapy crawl products -o output.csv
```

15.6 httpx — Async Requests (Faster than requests)

```
# pip install httpx
import httpx
import asyncio
import pandas as pd

# Sync (drop-in requests replacement)
with httpx.Client(timeout=30, follow_redirects=True) as client:
    r = client.get("https://api.example.com/data", headers={"Authorization"
    r.raise_for_status()
    data = r.json()

# Async — fetch many URLs concurrently
async def fetch_all(urls: list[str], headers: dict = None) -> list[dict]:
    async with httpx.AsyncClient(timeout=30, headers=headers or {}) as clie
        tasks = [client.get(url) for url in urls]
        responses = await asyncio.gather(*tasks, return_exceptions=True)

    results = []
    for url, resp in zip(urls, responses):
        if isinstance(resp, Exception):
            results.append({"url": url, "error": str(resp), "data": None})
        else:
            results.append({"url": url, "error": None, "data": resp.json()}
    return results

urls = [f"https://api.example.com/users/{i}" for i in range(1, 101)]
results = asyncio.run(fetch_all(urls, headers={"Authorization": "Bearer TOK
df = pd.DataFrame([{**r["data"], "url": r["url"]} for r in results if not r
```

15.7 Handling Anti-Scraping Measures

```python
import requests
import time
import random
from itertools import cycle

# Rotate User-Agents
USER_AGENTS = [
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 Chrome/12
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 Chr
    "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 Chrome/118.0.0.0",
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Fir
]

def get_headers():
    return {
        "User-Agent": random.choice(USER_AGENTS),
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/
        "Accept-Language": "en-US,en;q=0.5",
        "Accept-Encoding": "gzip, deflate, br",
        "DNT": "1",
        "Connection": "keep-alive"
    }

# Use a session (reuses TCP connection, handles cookies)
session = requests.Session()
session.headers.update(get_headers())

# Rotate proxies
PROXIES = [
    "http://user:pass@proxy1.example.com:8080",
    "http://user:pass@proxy2.example.com:8080",
]
proxy_pool = cycle(PROXIES)

def scrape_with_proxy(url: str) -> requests.Response:
    proxy = next(proxy_pool)
    return session.get(url, proxies={"http": proxy, "https": proxy}, timeou

# Random delay between requests (human-like)
def polite_delay(min_sec=1.0, max_sec=3.0):
    time.sleep(random.uniform(min_sec, max_sec))

# Handle cookies and sessions (e.g., login-protected pages)
session = requests.Session()
login_payload = {"username": "user@example.com", "password": "pass"}
```

```
session.post("https://example.com/login", data=login_payload)
# Session now has cookies — all subsequent requests are authenticated
r = session.get("https://example.com/dashboard")

# Check robots.txt before scraping
from urllib.robotparser import RobotFileParser
from urllib.parse import urljoin

def is_allowed(url: str, user_agent: str = "*") -> bool:
    from urllib.parse import urlparse
    parsed = urlparse(url)
    robots_url = f"{parsed.scheme}://{parsed.netloc}/robots.txt"
    rp = RobotFileParser()
    rp.set_url(robots_url)
    rp.read()
    return rp.can_fetch(user_agent, url)

print(is_allowed("https://example.com/products"))  # True/False
```

15.8 Parsing Common Data Formats from Scraping

```
import pandas as pd
from bs4 import BeautifulSoup
import json
import re

# Extract ALL tables from a page at once
def extract_all_tables(url: str) -> list[pd.DataFrame]:
    r = requests.get(url, headers=get_headers(), timeout=15)
    return pd.read_html(r.content)   # returns list of DataFrames

# Extract JSON embedded in page (common in SPAs)
def extract_embedded_json(html: str, var_name: str = "window.__DATA__") ->
    """Find JSON assigned to a JS variable inside <script> tags."""
    soup = BeautifulSoup(html, "lxml")
    for script in soup.find_all("script"):
        text = script.string or ""
        if var_name in text:
            match = re.search(rf"{re.escape(var_name)}\s*=\s*(\{{.*?\}})",
            if match:
                return json.loads(match.group(1))
    return {}
```

```python
# Extract JSON-LD structured data (many e-commerce sites use this)
def extract_json_ld(html: str) -> list[dict]:
    soup = BeautifulSoup(html, "lxml")
    results = []
    for tag in soup.find_all("script", type="application/ld+json"):
        try:
            results.append(json.loads(tag.string))
        except Exception:
            pass
    return results

# Example: scrape product with JSON-LD
r = requests.get("https://example.com/product/123", headers=get_headers())
structured = extract_json_ld(r.text)
# structured may contain: {"@type": "Product", "name": "...", "price": "...

# Extract CSV/Excel from download links
def download_file(url: str, dest: str):
    r = requests.get(url, headers=get_headers(), stream=True, timeout=60)
    r.raise_for_status()
    with open(dest, "wb") as f:
        for chunk in r.iter_content(chunk_size=8192):
            f.write(chunk)
    print(f"Downloaded: {dest}")

download_file("https://example.com/export?format=csv", "data/export.csv")
df = pd.read_csv("data/export.csv")
```

15.9 Scraping Cheatsheet — Selector Reference

```python
# BeautifulSoup selectors
soup.find("div")                          # first <div>
soup.find("div", class_="card")           # first <div class="card">
soup.find("div", id="main")               # by id
soup.find_all("a", href=True)             # all <a> with href
soup.select("div.card > h2")             # CSS selector
soup.select_one(".price span")            # first match
el.get_attribute_list("class")            # all classes as list
el.text.strip()                           # text content cleaned
el["href"]                                # attribute value
el.get("src", "")                         # safe attribute get

# Scrapy CSS selectors
```

```
response.css("h1::text").get()           # text content of h1
response.css("a::attr(href)").getall()   # all href values
response.css(".price::text").get("0")    # with default
response.xpath("//h1/text()").get()      # XPath equivalent
response.xpath("//a/@href").getall()

# Playwright selectors
page.query_selector("h1").inner_text()
page.query_selector_all(".card")
page.locator("text=Add to Cart").click()       # by text
page.locator("[data-testid=price]").inner_text()  # by data attr
page.wait_for_selector(".results", state="visible")
page.evaluate("document.title")          # run JS
page.screenshot(path="debug.png")        # debug screenshot
```

15.10 Quick Tools Reference — When to Use What

TOOL	INSTALL	USE WHEN
requests	pip install requests	Simple HTTP, static pages, API
httpx	pip install httpx	Async fetching, many URLs at o
BeautifulSoup4	pip install bs4 lxml	Parse static HTML, quick jobs
Scrapy	pip install scrapy	Large crawls, 1000s of pages,
Selenium	pip install selenium	Old-school JS sites, Chrome au
Playwright	pip install playwright	Modern JS sites, faster than S
mechanize	pip install mechanize	Form submission on old sites
cloudscraper	pip install cloudscraper	Bypass Cloudflare (basic)
curl_cffi	pip install curl_cffi	Impersonate browsers at HTTP l
selectolax	pip install selectolax	Fastest HTML parsing (C-based)
pandas.read_html	(built-in pandas)	Scrape HTML tables instantly
camelot	pip install camelot-py	Extract tables from PDFs
pdfplumber	pip install pdfplumber	Read text + tables from PDFs

SECTION 16 — Docker & Containerization for Data

16.1 Dockerfile for Data Pipeline

```
# Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

# Install Python packages
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy code
COPY src/ ./src/
COPY config/ ./config/

# Non-root user (security best practice)
RUN useradd --create-home appuser
USER appuser

# Run pipeline
CMD ["python", "src/pipeline.py"]
```

16.2 Docker Compose for Full Stack

```
# docker-compose.yml
version: "3.9"

services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: datauser
      POSTGRES_PASSWORD: datapass
      POSTGRES_DB: analytics
    ports:
```

```yaml
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U datauser"]
      interval: 10s
      retries: 5

  pipeline:
    build: .
    depends_on:
      postgres:
        condition: service_healthy
    environment:
      DATABASE_URL: postgresql://datauser:datapass@postgres:5432/analytics
      API_KEY: ${API_KEY}
    volumes:
      - ./data:/app/data
    restart: "no"

  jupyter:
    image: jupyter/datascience-notebook:python-3.11
    ports:
      - "8888:8888"
    volumes:
      - ./notebooks:/home/jovyan/work
      - ./data:/home/jovyan/data
    environment:
      JUPYTER_TOKEN: "mytoken"

  airflow-webserver:
    image: apache/airflow:2.8.0
    command: webserver
    ports:
      - "8080:8080"
    environment:
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://datauser:d
      AIRFLOW__CORE__EXECUTOR: LocalExecutor
    depends_on:
      - postgres
    volumes:
      - ./dags:/opt/airflow/dags

volumes:
  postgres_data:
```

16.3 Common Docker Commands

```
# Build image
docker build -t my-pipeline:latest .

# Run container
docker run --env-file .env my-pipeline:latest

# Run interactively
docker run -it --entrypoint bash my-pipeline:latest

# Docker Compose
docker-compose up -d                # start all services detached
docker-compose logs -f pipeline     # follow logs for service
docker-compose exec postgres psql -U datauser -d analytics  # shell into se
docker-compose down -v              # stop and remove volumes

# Debug a running container
docker exec -it container_name bash
docker logs container_name --tail 100
docker inspect container_name

# Clean up
docker system prune -a              # remove all unused resources
```

SECTION 17 — Git & Version Control for Data Teams

17.1 Git Essentials

```
# Setup
git config --global user.name "Your Name"
git config --global user.email "you@company.com"

# Daily workflow
```

```
git status                          # see changes
git add file.py                     # stage specific file
git add -p                          # stage interactively (review each chan
git commit -m "feat: add orders ETL pipeline"
git push origin feature/orders-etl

# Branching strategy
git checkout -b feature/add-cohort-analysis  # new branch
git checkout main && git pull                 # update main
git merge feature/add-cohort-analysis         # merge
git branch -d feature/add-cohort-analysis     # delete branch

# Undo things
git restore file.py                   # discard unstaged changes
git restore --staged file.py          # unstage
git revert abc1234                    # revert a commit (safe, creates new
git reset --soft HEAD~1               # undo last commit, keep changes stag
git reset --hard HEAD~1               # undo last commit, discard changes (

# Stash (temporarily save work)
git stash                             # save current work
git stash pop                         # restore saved work
git stash list                        # see all stashes
```

17.2 .gitignore for Data Projects

```
# .gitignore for data projects

# Secrets — NEVER commit these
.env
.env.*
secrets/
credentials.json
*.pem
*.key

# Data files — usually too large for git
data/raw/
data/processed/
*.csv
*.parquet
*.xlsx
*.pkl
```

```
*.joblib

# Jupyter checkpoints
.ipynb_checkpoints/
*/.ipynb_checkpoints/*

# Python
__pycache__/
*.py[cod]
*.egg-info/
dist/
build/
venv/
.venv/
.python-version

# dbt
dbt_packages/
target/
logs/
profiles.yml  # contains passwords — use environment variables

# OS
.DS_Store
Thumbs.db

# IDEs
.idea/
.vscode/
*.swp
```

17.3 Git for Data (dvc)

```
# DVC — version control for data and models
pip install dvc dvc-s3

dvc init                        # initialize in git repo
dvc add data/raw/orders.csv     # track data file
git add data/raw/orders.csv.dvc .gitignore
git commit -m "Track orders dataset"

# Remote storage
dvc remote add -d s3remote s3://my-bucket/dvc
```

```
dvc push                          # upload data to S3
dvc pull                          # download data from S3

# Reproduce pipeline
dvc run -n process_data \
    -d data/raw/orders.csv \
    -d src/process.py \
    -o data/processed/clean_orders.parquet \
    python src/process.py

dvc repro                         # reproduce all pipeline stages
dvc dag                           # show pipeline DAG
```

SECTION 18 — Real-Time & Streaming Data (Kafka)

18.1 Kafka Concepts

```
PRODUCER  →  TOPIC  →  CONSUMER GROUP
                ↓
            PARTITIONS (parallel lanes within a topic)
            Each message has: offset, timestamp, key, value

Key concepts:
- Topic: named stream of records (like a table)
- Partition: ordered, immutable log (enables parallel processing)
- Consumer Group: multiple consumers sharing work, each partition read by o
- Offset: position of a message in a partition
- Retention: how long messages are stored (e.g., 7 days)
```

18.2 Kafka Producer (Python)

```
# pip install confluent-kafka
from confluent_kafka import Producer
```

```
import json
from datetime import datetime

conf = {
    "bootstrap.servers": "localhost:9092",
    "client.id": "my-producer",
    "acks": "all",              # wait for all replicas
    "retries": 3,
    "compression.type": "gzip"
}
producer = Producer(conf)

def delivery_report(err, msg):
    if err:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Delivered to {msg.topic()} [{msg.partition()}] @ offset {ms

# Produce messages
def produce_order(order: dict):
    producer.produce(
        topic="orders",
        key=str(order["user_id"]).encode("utf-8"),  # key for partitioning
        value=json.dumps(order).encode("utf-8"),
        callback=delivery_report
    )
    producer.poll(0)   # trigger delivery reports

orders = [
    {"order_id": 1, "user_id": 101, "amount": 99.99, "ts": datetime.utcnow(
    {"order_id": 2, "user_id": 102, "amount": 149.0, "ts": datetime.utcnow(
]

for order in orders:
    produce_order(order)

producer.flush()  # wait for all messages to be delivered
```

18.3 Kafka Consumer (Python)

```
from confluent_kafka import Consumer
import json
```

```
conf = {
    "bootstrap.servers": "localhost:9092",
    "group.id": "orders-consumer-group",
    "auto.offset.reset": "earliest",    # start from beginning if no offset
    "enable.auto.commit": False          # manual commit for exactly-once
}
consumer = Consumer(conf)
consumer.subscribe(["orders"])

def process_order(order: dict):
    """Your processing logic here."""
    print(f"Processing order {order['order_id']} for user {order['user_id']

try:
    while True:
        msg = consumer.poll(timeout=1.0)
        if msg is None:
            continue
        if msg.error():
            print(f"Consumer error: {msg.error()}")
            continue

        order = json.loads(msg.value().decode("utf-8"))
        process_order(order)

        # Manual commit (after successful processing)
        consumer.commit(asynchronous=False)

except KeyboardInterrupt:
    pass
finally:
    consumer.close()
```

18.4 Stream Processing with Faust

```
# pip install faust-streaming
import faust
import json

app = faust.App("orders-processor", broker="kafka://localhost:9092")

class Order(faust.Record):
    order_id: int
```

```
    user_id: int
    amount: float
    status: str

orders_topic = app.topic("orders", value_type=Order)
enriched_topic = app.topic("enriched_orders")

@app.agent(orders_topic)
async def process_orders(orders):
    async for order in orders:
        # Enrich the order
        enriched = {
            "order_id": order.order_id,
            "user_id": order.user_id,
            "amount": order.amount,
            "tier": "premium" if order.amount > 100 else "standard",
            "processed": True
        }
        await enriched_topic.send(value=enriched)
        print(f"Processed order {order.order_id}")

if __name__ == "__main__":
    app.main()
```

SECTION 19 — MLOps & Model Deployment

19.1 MLflow Experiment Tracking

```
# pip install mlflow
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, roc_auc_score
import pandas as pd

mlflow.set_tracking_uri("http://localhost:5000")  # or file:./mlruns
```

```
mlflow.set_experiment("titanic-survival")

with mlflow.start_run(run_name="rf_baseline") as run:
    # Log parameters
    params = {"n_estimators": 100, "max_depth": 5, "random_state": 42}
    mlflow.log_params(params)

    # Train
    clf = RandomForestClassifier(**params)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_proba = clf.predict_proba(X_test)[:, 1]

    # Log metrics
    metrics = {
        "f1": f1_score(y_test, y_pred),
        "roc_auc": roc_auc_score(y_test, y_proba),
        "train_size": len(X_train),
        "test_size": len(X_test)
    }
    mlflow.log_metrics(metrics)

    # Log model
    mlflow.sklearn.log_model(clf, "model",
                             registered_model_name="titanic-rf",
                             input_example=X_train.head(1))

    # Log artifacts
    mlflow.log_artifact("outputs/confusion_matrix.png")

    print(f"Run ID: {run.info.run_id}")
    print(f"F1: {metrics['f1']:.4f}, AUC: {metrics['roc_auc']:.4f}")

# Load model from registry
model = mlflow.sklearn.load_model("models:/titanic-rf/Production")
predictions = model.predict(X_new)

# Compare runs
client = mlflow.MlflowClient()
runs = client.search_runs(experiment_ids=["1"], order_by=["metrics.roc_auc
for run in runs[:5]:
    print(f"{run.data.tags.get('mlflow.runName')}: AUC={run.data.metrics.ge
```

19.2 FastAPI Model Serving

```python
# pip install fastapi uvicorn pydantic
# Run: uvicorn src.api:app --reload

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, validator
import joblib
import pandas as pd
import numpy as np
from typing import List, Optional

app = FastAPI(title="ML Model API", version="1.0.0")
model = joblib.load("models/titanic_rf_pipeline.joblib")

class PassengerInput(BaseModel):
    pclass: int
    sex: str
    age: float
    sibsp: int
    parch: int
    fare: float
    embarked: Optional[str] = "S"

    @validator("pclass")
    def pclass_valid(cls, v):
        if v not in [1, 2, 3]:
            raise ValueError("pclass must be 1, 2, or 3")
        return v

class PredictionResponse(BaseModel):
    prediction: int
    probability: float
    label: str

@app.get("/health")
def health():
    return {"status": "ok", "model": "titanic-rf"}

@app.post("/predict", response_model=PredictionResponse)
def predict(passenger: PassengerInput):
    try:
        df = pd.DataFrame([passenger.dict()])
        pred = int(model.predict(df)[0])
        prob = float(model.predict_proba(df)[0][1])
        return PredictionResponse(
            prediction=pred,
```

```
            probability=round(prob, 4),
            label="Survived" if pred == 1 else "Died"
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/predict/batch")
def predict_batch(passengers: List[PassengerInput]):
    df = pd.DataFrame([p.dict() for p in passengers])
    preds = model.predict(df).tolist()
    probs = model.predict_proba(df)[:, 1].tolist()
    return [{"prediction": p, "probability": round(pr, 4)} for p, pr in zip
```

19.3 Model Monitoring

```
import pandas as pd
import numpy as np
from scipy.stats import ks_2samp, chi2_contingency
from typing import Dict

class ModelMonitor:
    """Monitor model inputs and outputs for drift and degradation."""

    def __init__(self, reference_df: pd.DataFrame):
        self.reference = reference_df
        self.alerts = []

    def check_data_drift(self, current_df: pd.DataFrame, threshold: float =
        results = {}

        for col in current_df.select_dtypes(include=[np.number]).columns:
            if col not in self.reference.columns:
                continue
            stat, p_value = ks_2samp(
                self.reference[col].dropna(),
                current_df[col].dropna()
            )
            drifted = p_value < threshold
            results[col] = {"ks_stat": round(stat, 4), "p_value": round(p_v
            if drifted:
                self.alerts.append(f"DRIFT: {col} (p={p_value:.4f})")

        return results
```

```
    def check_prediction_drift(self, ref_preds: np.ndarray, current_preds:
        stat, p_value = ks_2samp(ref_preds, current_preds)
        if p_value < 0.05:
            self.alerts.append(f"PREDICTION DRIFT: p={p_value:.4f}")
        return {"ks_stat": stat, "p_value": p_value, "drifted": p_value < 0

    def check_missing_rates(self, current_df: pd.DataFrame, threshold: floa
        current_missing = current_df.isnull().mean()
        ref_missing = self.reference.isnull().mean()

        for col in current_df.columns:
            if col in ref_missing.index:
                diff = abs(current_missing[col] - ref_missing[col])
                if diff > threshold:
                    self.alerts.append(f"MISSING RATE CHANGE: {col} ({diff:

    def report(self) -> str:
        if not self.alerts:
            return "✅ No issues detected"
        return "\n".join([f"⚠️ {a}" for a in self.alerts])

# Usage
monitor = ModelMonitor(reference_df=X_train)
drift = monitor.check_data_drift(X_new)
monitor.check_missing_rates(X_new)
print(monitor.report())
```

SECTION 20 — LLMs & AI in Data Workflows (2026)

20.1 OpenAI / Anthropic API for Data Tasks

```
# pip install openai anthropic
from openai import OpenAI
import anthropic
import pandas as pd
```

```python
import json

# OpenAI
openai_client = OpenAI(api_key="sk-...")

def ask_gpt(prompt: str, model: str = "gpt-4o", temperature: float = 0) ->
    response = openai_client.chat.completions.create(
        model=model,
        temperature=temperature,
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content

# Anthropic Claude
claude = anthropic.Anthropic(api_key="sk-ant-...")

def ask_claude(prompt: str) -> str:
    message = claude.messages.create(
        model="claude-opus-4-6",
        max_tokens=2048,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

# Use LLM to classify free-text data
def classify_support_tickets(tickets: list[str]) -> list[str]:
    categories = ["billing", "technical", "shipping", "returns", "general"]
    results = []
    for ticket in tickets:
        prompt = f"""Classify this support ticket into exactly one of: {cat
Ticket: {ticket}
Return only the category name, nothing else."""
        category = ask_gpt(prompt).strip().lower()
        results.append(category)
    return results

# Use LLM to generate SQL
def nl_to_sql(question: str, schema: str) -> str:
    prompt = f"""You are a SQL expert. Given this schema:
{schema}

Write a SQL query to answer: {question}
Return only the SQL query, no explanation."""
    return ask_claude(prompt)

schema = """
```

```
Table: orders (order_id, user_id, amount, status, created_at)
Table: users (user_id, email, country, signup_date)
"""
sql = nl_to_sql("What is the average order value by country for the last 30
print(sql)
```

20.2 LLM for Data Quality & Enrichment

```
# Use LLM to extract structured data from unstructured text
def extract_structured(text: str) -> dict:
    prompt = f"""Extract information from this text and return JSON with ke
company_name, industry, revenue (in millions USD), employee_count, founding
Use null for missing values.

Text: {text}

Return only valid JSON."""

    response = ask_gpt(prompt)
    # Strip markdown code fences if present
    clean = response.strip().removeprefix("```json").removesuffix("```").st
    return json.loads(clean)

# Batch processing with rate limiting
import time

def batch_extract(texts: list[str], delay: float = 0.5) -> list[dict]:
    results = []
    for i, text in enumerate(texts):
        try:
            result = extract_structured(text)
            results.append(result)
        except Exception as e:
            results.append({"error": str(e)})
        if i < len(texts) - 1:
            time.sleep(delay)
    return results

# Use embeddings for semantic similarity
def get_embeddings(texts: list[str]) -> list[list[float]]:
    response = openai_client.embeddings.create(
        input=texts,
        model="text-embedding-3-small"
```

```
    )
    return [item.embedding for item in response.data]

# Semantic deduplication
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def find_similar_records(df: pd.DataFrame, text_col: str, threshold: float
    embeddings = get_embeddings(df[text_col].tolist())
    sim_matrix = cosine_similarity(embeddings)
    duplicates = []
    for i in range(len(sim_matrix)):
        for j in range(i+1, len(sim_matrix)):
            if sim_matrix[i][j] >= threshold:
                duplicates.append((i, j, sim_matrix[i][j]))
    return duplicates
```

20.3 LLM-Powered Data Pipeline (LangChain)

```
# pip install langchain langchain-openai
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from typing import List, Optional

# Structured output with Pydantic
class ProductReview(BaseModel):
    sentiment: str = Field(description="positive, negative, or neutral")
    rating: int = Field(description="1-5 estimated rating")
    topics: List[str] = Field(description="main topics mentioned")
    summary: str = Field(description="one sentence summary")

parser = PydanticOutputParser(pydantic_object=ProductReview)
llm = ChatOpenAI(model="gpt-4o", temperature=0)

def analyze_review(review_text: str) -> ProductReview:
    messages = [
        SystemMessage(content="You analyze product reviews and return struc
        HumanMessage(content=f"""Analyze this review:
{review_text}

{parser.get_format_instructions()}""")
```

```
    ]
    response = llm.invoke(messages)
    return parser.parse(response.content)

# Batch reviews
reviews = [
    "Amazing product! Fast shipping, great quality. Would buy again. 10/10"
    "Terrible experience. Broke after 2 days. Customer service was useless.
    "It's ok. Does what it says but nothing special. Delivery was slow."
]

for review in reviews:
    result = analyze_review(review)
    print(f"Sentiment: {result.sentiment} | Rating: {result.rating} | Topic
```

SECTION 21 — Data Governance, Privacy & Compliance

21.1 PII Detection and Masking

```
import re
import hashlib
import pandas as pd

# PII patterns
PII_PATTERNS = {
    "email": r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b",
    "phone_us": r"\b(?:\+1)?[\s.-]?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}\b",
    "ssn": r"\b\d{3}-\d{2}-\d{4}\b",
    "credit_card": r"\b(?:\d{4}[-\s]?){3}\d{4}\b",
    "ip_address": r"\b(?:\d{1,3}\.){3}\d{1,3}\b"
}

def detect_pii(text: str) -> dict:
    findings = {}
    for pii_type, pattern in PII_PATTERNS.items():
        matches = re.findall(pattern, str(text), re.IGNORECASE)
```

```
        if matches:
            findings[pii_type] = matches
    return findings

def mask_pii(text: str, replacement: str = "[REDACTED]") -> str:
    masked = str(text)
    for pii_type, pattern in PII_PATTERNS.items():
        masked = re.sub(pattern, replacement, masked, flags=re.IGNORECASE)
    return masked

def pseudonymize(value: str, salt: str = "your-secret-salt") -> str:
    """One-way hash for pseudonymization (reversible with salt, but not obv
    return hashlib.sha256(f"{salt}{value}".encode()).hexdigest()[:16]

def tokenize_pii_dataframe(df: pd.DataFrame, pii_columns: list) -> pd.DataF
    df = df.copy()
    for col in pii_columns:
        df[f"{col}_original"] = df[col]
        df[col] = df[col].apply(lambda x: pseudonymize(str(x)) if pd.notna(
    return df

# Scan entire DataFrame for PII
def scan_dataframe_for_pii(df: pd.DataFrame) -> pd.DataFrame:
    results = []
    for col in df.columns:
        sample = df[col].dropna().head(100).astype(str)
        pii_found = {}
        for val in sample:
            pii = detect_pii(val)
            for k, v in pii.items():
                pii_found.setdefault(k, []).extend(v)
        if pii_found:
            results.append({"column": col, "pii_types": list(pii_found.keys
                            "sample_count": len(pii_found)})
    return pd.DataFrame(results)
```

21.2 GDPR / CCPA Compliance Patterns

```
# Right to deletion (RTBF — Right to be Forgotten)
def delete_user_data(user_id: str, engine) -> dict:
    """Delete or anonymize all user data across tables."""
    from sqlalchemy import text
    deleted_records = {}
```

```
    with engine.connect() as conn:
        # Hard delete from non-essential tables
        for table in ["events", "sessions", "user_logs"]:
            result = conn.execute(text(f"DELETE FROM {table} WHERE user_id
            deleted_records[table] = result.rowcount

        # Anonymize instead of delete where you need to keep aggregate inte
        conn.execute(text("""
            UPDATE orders
            SET user_id = 'DELETED', email = 'deleted@deleted.com',
                name = 'Deleted User', ip_address = NULL
            WHERE user_id = :uid
        """), {"uid": user_id})

        conn.commit()

    return deleted_records

# Data retention policy
def apply_retention_policy(df: pd.DataFrame, date_col: str, retention_days:
    cutoff = pd.Timestamp.now() - pd.Timedelta(days=retention_days)
    before = len(df)
    df = df[df[date_col] >= cutoff]
    print(f"Retention policy: removed {before - len(df)} rows older than {r
    return df

# Consent tracking
def check_user_consent(user_id: str, purpose: str, engine) -> bool:
    """Check if user has given consent for a specific data processing purpo
    from sqlalchemy import text
    with engine.connect() as conn:
        result = conn.execute(text("""
            SELECT consented FROM user_consents
            WHERE user_id = :uid AND purpose = :purpose
            AND consent_date >= NOW() - INTERVAL '1 year'
        """), {"uid": user_id, "purpose": purpose})
        row = result.fetchone()
        return bool(row and row[0])
```

SECTION 22 — Interview Prep & Career Cheatsheet

22.1 SQL Interview Patterns

```sql
-- PATTERN 1: Find Nth highest value
SELECT DISTINCT salary FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET N-1;  -- N=2 gives 2nd highest

-- PATTERN 2: Running total that resets
SELECT date, amount,
    SUM(amount) OVER (PARTITION BY user_id ORDER BY date) AS running_total
FROM orders;

-- PATTERN 3: Consecutive days active
WITH daily AS (SELECT DISTINCT user_id, DATE(event_date) AS day FROM events
gaps AS (
    SELECT user_id, day,
        day - ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY day) * INTER
    FROM daily
)
SELECT user_id, MIN(day), MAX(day), COUNT(*) AS streak_length
FROM gaps GROUP BY user_id, grp HAVING COUNT(*) >= 3;

-- PATTERN 4: Self-join for relationship
SELECT a.user_id AS user, b.user_id AS referred_friend
FROM users a
JOIN users b ON a.referral_code = b.referred_by;

-- PATTERN 5: Find users who did A then B (funnel)
SELECT DISTINCT s.user_id
FROM events s
JOIN events p ON s.user_id = p.user_id
    AND s.event = 'signup'
    AND p.event = 'purchase'
    AND p.created_at > s.created_at;

-- PATTERN 6: Month-over-month growth
WITH monthly AS (
    SELECT DATE_TRUNC('month', date) AS month, SUM(revenue) AS revenue
```

```
    FROM orders GROUP BY 1
)
SELECT month, revenue,
    LAG(revenue) OVER (ORDER BY month) AS prev_month,
    (revenue - LAG(revenue) OVER (ORDER BY month)) / LAG(revenue) OVER (ORD
FROM monthly;

-- PATTERN 7: Median (no native function in many DBs)
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) AS median_salary

-- PATTERN 8: Duplicate records
SELECT order_id, COUNT(*) FROM orders GROUP BY order_id HAVING COUNT(*) > 1
```

22.2 Python Interview Patterns

```
import pandas as pd
import numpy as np

# PATTERN 1: Find top N per group
df.groupby("region").apply(lambda x: x.nlargest(3, "revenue")).reset_index(

# PATTERN 2: Fill missing with group median
df["amount"] = df.groupby("category")["amount"].transform(lambda x: x.filln

# PATTERN 3: Pivot-style aggregate
df.pivot_table(values="revenue", index="region", columns="quarter", aggfunc

# PATTERN 4: Merge and check
df_merged = df_left.merge(df_right, on="id", how="left", indicator=True)
unmatched = df_merged[df_merged["_merge"] == "left_only"]

# PATTERN 5: Rolling retention
def cohort_retention(df):
    df["cohort"] = df.groupby("user_id")["date"].transform("min").dt.to_per
    df["period"] = df["date"].dt.to_period("M")
    df["age"] = (df["period"] - df["cohort"]).apply(lambda x: x.n)
    cohort_size = df.groupby("cohort")["user_id"].nunique()
    retention = df.groupby(["cohort", "age"])["user_id"].nunique().div(coho
    return retention.unstack()
```

22.3 Key Metrics Definitions

```
metrics = {
    # Growth
    "DAU": "Daily Active Users — unique users with activity on a given day"
    "MAU": "Monthly Active Users",
    "DAU/MAU": "Stickiness ratio — what % of monthly users are daily active
    "MoM Growth": "(current_month - prev_month) / prev_month * 100",

    # Revenue
    "MRR": "Monthly Recurring Revenue — sum of all subscription revenue in
    "ARR": "Annual Recurring Revenue = MRR × 12",
    "ARPU": "Average Revenue Per User = Revenue / Users",
    "ACV": "Annual Contract Value — annual value of a contract",
    "LTV": "Lifetime Value = ARPU × (1/churn_rate) or ARPU × average_months
    "LTV:CAC": "LTV / Customer Acquisition Cost — should be > 3",

    # Engagement / Retention
    "Churn Rate": "(Users lost in period) / (Users at start of period)",
    "Retention D30": "% of day-0 users still active on day 30",
    "NPS": "Net Promoter Score = %Promoters - %Detractors",

    # Conversion
    "CVR": "Conversion Rate = Conversions / Visitors",
    "CTR": "Click-Through Rate = Clicks / Impressions",
    "CAC": "Customer Acquisition Cost = Marketing Spend / New Customers",

    # Product
    "Activation Rate": "% new users reaching key activation milestone",
    "Feature Adoption": "% of users using a specific feature",
    "Time to Value": "Time from signup to first value moment"
}

for k, v in metrics.items():
    print(f"{k:20s}: {v}")
```

22.4 Data Engineer Interview Checklist

```
SYSTEM DESIGN QUESTIONS — Know How to Answer:
═══════════════════════════════════════════════
□ Design a pipeline to ingest 10TB of data daily
□ How would you handle late-arriving data in a streaming pipeline?
□ Design a data warehouse for an e-commerce company
□ How do you ensure exactly-once processing in Kafka?
```

□ What is data lake vs data warehouse vs data lakehouse?
□ How do you handle schema evolution in Avro/Parquet?
□ Explain partitioning strategies in Spark and when to use each
□ How do you detect and handle data quality issues at scale?
□ What is CDC (Change Data Capture) and how does it work?
□ Design a real-time recommendation system data pipeline

CONCEPTS TO KNOW:
═══════════════════

□ CAP theorem (Consistency, Availability, Partition tolerance)
□ Lambda vs Kappa architecture
□ Star schema vs Snowflake schema vs Data Vault
□ ACID vs BASE
□ Normalization vs Denormalization
□ Partitioning vs Bucketing (Hive/Spark)
□ Shuffle in Spark and how to minimize it
□ Data skew and handling strategies
□ Idempotency in pipelines
□ SCD Types 1, 2, 3
□ Batch vs micro-batch vs streaming

22.5 Tools Quick Reference Card

TOOL	CATEGORY	USE WHEN
pandas	Data Analysis	< 10M rows, single machine
polars	Data Analysis	Fast alternative to pandas (Rust-based)
PySpark	Big Data	> 10M rows, distributed processing
DuckDB	Analytics SQL	SQL on local files, fast analytics
Snowflake	Cloud DW	Production DW, SaaS analytics
BigQuery	Cloud DW	Google ecosystem, massive scale
dbt	Transformation	SQL transformations, analytics engineering
Airflow	Orchestration	Complex DAG-based workflows
Prefect	Orchestration	Modern alternative, Python-native
Kafka	Streaming	Real-time events, high throughput
Flink	Streaming	Stateful stream processing
MLflow	MLOps	Experiment tracking, model registry
FastAPI	Serving	REST API for models, lightweight
Streamlit	BI/Dashboards	Quick Python dashboards, prototypes
Tableau	BI	Business dashboards, non-technical users
Power BI	BI	Microsoft ecosystem, self-service
Looker	BI	LookML, semantic layer, enterprise
Great Exp.	Data Quality	Expectations-based testing

```
Docker          Infrastructure   Containerize any data workload
Terraform       Infrastructure   Infrastructure as code for cloud
GitHub Actions CI/CD             Automate tests, deployments, dbt runs
```

22.6 Environment Variables Best Practices

```
# .env file (never commit to git)
# DATABASE_URL=postgresql://user:pass@host:5432/db
# API_KEY=sk-...
# S3_BUCKET=my-bucket
# ENVIRONMENT=production

import os
from dotenv import load_dotenv   # pip install python-dotenv

load_dotenv()  # loads .env file into environment

DATABASE_URL = os.environ["DATABASE_URL"]          # raises if missing (exp
API_KEY = os.environ.get("API_KEY", "default")     # returns default if mis
DEBUG = os.environ.get("DEBUG", "false").lower() == "true"

# Validate config at startup
required_vars = ["DATABASE_URL", "API_KEY", "S3_BUCKET"]
missing = [v for v in required_vars if not os.environ.get(v)]
if missing:
    raise EnvironmentError(f"Missing required environment variables: {missi

# Config class
from dataclasses import dataclass

@dataclass
class Config:
    database_url: str = os.environ.get("DATABASE_URL", "")
    api_key: str = os.environ.get("API_KEY", "")
    s3_bucket: str = os.environ.get("S3_BUCKET", "")
    environment: str = os.environ.get("ENVIRONMENT", "development")
    debug: bool = os.environ.get("DEBUG", "false").lower() == "true"
    batch_size: int = int(os.environ.get("BATCH_SIZE", "1000"))

config = Config()
print(f"Running in {config.environment} mode")
```

Final Reference: The Data Lifecycle

```
1. COLLECT          Web scraping, APIs, DB replication, Kafka events, IoT s
        ↓
2. STORE            S3/GCS/ADLS (raw), PostgreSQL, Snowflake, BigQuery, Del
        ↓
3. PROCESS          Pandas (small), PySpark (large), dbt (SQL transforms),
        ↓
4. QUALITY          Great Expectations, dbt tests, custom checks, monitorin
        ↓
5. MODEL/ANALYZE    scikit-learn, XGBoost, Prophet, SQL analytics, A/B test
        ↓
6. SERVE            FastAPI, Streamlit, Tableau, Looker, scheduled reports
        ↓
7. MONITOR          MLflow, Grafana, Monte Carlo, Anomalo, custom dashboard
        ↓
8. GOVERN           Datahub, Collibra, PII masking, GDPR compliance, access
```

By Atharv Khare (1mystic) 2026

For Data roles : Analyst · Scientist · Engineer · Analytics Engineer