**COVID-19 Data Ingestion**

Technical Documentation

## Table of Contents

1. Project Overview
2. System Architecture
3. Data Ingestion Process (End-to-End Detail)
4. Data Source and Schema
5. Infrastructure Setup
6. Data Pipeline Components
7. Data Quality and Validation
8. Testing Framework
9. Deployment and Operations
10. Conclusion

# 1. Project Overview

## 1.1 Purpose

This project implements a robust data ingestion pipeline that extracts COVID-19 case surveillance data from the CDC's public API, processes and cleans the data, and loads it into a PostgreSQL database for analysis. The system is designed to handle large datasets (100k+ records) with built-in error handling, rate limiting, and data quality validation.

## 1.2 Key Features

* Large-scale data ingestion: Handles 100k+ records efficiently
* Dual-table architecture: Raw data landing table + cleaned analytical table
* API rate limiting handling: Built-in retry logic for 429 errors
* Data quality assurance: Comprehensive cleaning and validation
* Comprehensive testing: Unit tests covering all major components
* Batch processing: Configurable page sizes for optimal performance

## 1.3 Technology Stack

* Language: Python 3.x
* Database: PostgreSQL (Neon Cloud)
* Data Processing: pandas, SQLAlchemy
* API Client: requests
* Testing: pytest
* Environment: Google Colab compatible

# 2. System Architecture

## 2.1 High-Level Architecture

CDC COVID-19 API → API Client → Data Processing → PostgreSQL Database  
 ↓  
 Unit Tests ← Data Validation

## 2.2 Data Flow

1. Extraction: Paginated API calls to CDC COVID-19 dataset
2. Raw Storage: Immediate storage in "covid\_case\_surveillance\_raw" table
3. Processing: Data cleaning, deduplication, and standardization
4. Final Storage: Clean data stored in "covid\_case\_surveillance" table
5. Validation: Continuous data quality checks throughout the process

## 2.3 Database Schema

* Raw Table: "covid\_case\_surveillance\_raw" (with "\_ingested\_at" timestamp)
* Clean Table: "covid\_case\_surveillance" (processed data)

# 3. Data Ingestion Process (End-to-End Detail)

## 3.1 Source: CDC COVID-19 Case Surveillance Public Use Data

API Endpoint: "https://data.cdc.gov/resource/n8mc-b4w4.json"

Dataset Information:

* Name: COVID-19 Case Surveillance Public Use Data
* Provider: Centers for Disease Control and Prevention (CDC)
* Format: JSON via REST API
* Update Frequency: Regular updates by CDC
* Access: Public, no authentication required

## 3.2 Pagination Logic

The system implements sophisticated pagination to handle large datasets efficiently:

# Configuration  
page\_size = 50000 # Records per API call  
max\_records = 150000 # Total records to process  
offset = 0 # Starting position

Pagination Flow:

1. Calculate required records for current page: "min(page\_size, max\_records - total\_fetched)"
2. Make API call with "$limit" and "$offset" parameters
3. Process received data
4. Update offset: "offset += fetched\_count"
5. Continue until all records processed or API returns fewer records than requested

## 3.3 Rate Limiting and Error Handling

Rate Limiting Strategy:

* Detection: Monitor for HTTP 429 (Too Many Requests) responses
* Recovery: 2-second wait period before retry
* Prevention: 0.7-second delay between successful requests

Error Handling Hierarchy:

1. HTTP 429: Automatic retry with backoff
2. Other HTTP Errors: Immediate failure with error propagation
3. Network Errors: Logged and returned as None
4. Timeout: 60-second timeout per request

## 3.4 Data Processing Workflow

### Step 1: Raw Data Ingestion

# Fetch from API  
rows = fetch\_page(limit=need, offset=offset, where=where)  
  
# Convert to DataFrame  
df = transform\_rows\_to\_df(rows)  
  
# Add ingestion timestamp  
df["\_ingested\_at"] = pd.Timestamp.utcnow()  
  
# Store in raw table  
load\_df\_to\_postgres\_raw(engine, df)

### Step 2: Data Retrieval and Cleaning

# Read back the exact batch just inserted  
raw\_sql = "SELECT \* FROM covid\_case\_surveillance\_raw WHERE \_ingested\_at = %(ts)s"  
df\_raw\_from\_pg = pd.read\_sql\_query(raw\_sql, engine, params={"ts": batch\_ts})  
  
# Clean the data  
df\_clean = clean\_df(df\_raw\_from\_pg)  
  
# Store in analytical table  
load\_df\_to\_postgres(engine, df\_clean)

# 4. Data Source and Schema

## 4.1 Source Schema (API Response)

The CDC API returns JSON objects with the following key fields:

**Field | Type | Description**

`case\_month` | String | Year-month of case reporting

`cdc\_case\_earliest\_dt` | String | Earliest case date (YYYY-MM-DD)

`res\_state` | String | Residence state

`age\_group` | String | Age group classification

`sex` | String | Patient sex

`race` | String | Patient race

`ethnicity` | String | Patient ethnicity

`death\_yn` | String | Death indicator (Y/N)

`hosp\_yn` | String | Hospitalization indicator (Y/N)

`icu\_yn` | String | ICU admission indicator (Y/N)

`medcond\_yn` | String | Medical condition indicator (Y/N)

## 4.2 Target Schema (PostgreSQL Tables)

### Raw Table: `covid\_case\_surveillance\_raw`

CREATE TABLE covid\_case\_surveillance\_raw (  
 case\_month VARCHAR,  
 cdc\_case\_earliest\_dt VARCHAR,  
 res\_state VARCHAR,  
 age\_group VARCHAR,  
 sex VARCHAR,  
 race VARCHAR,  
 ethnicity VARCHAR,  
 death\_yn VARCHAR,  
 hosp\_yn VARCHAR,  
 icu\_yn VARCHAR,  
 medcond\_yn VARCHAR,  
 ingested\_at TIMESTAMP  
);

### Clean Table: `covid\_case\_surveillance`

CREATE TABLE covid\_case\_surveillance (  
 case\_month VARCHAR,  
 cdc\_case\_earliest\_dt TIMESTAMP,  
 res\_state VARCHAR,  
 age\_group VARCHAR,  
 sex VARCHAR,  
 race VARCHAR,  
 ethnicity VARCHAR,  
 death\_yn VARCHAR,  
 hosp\_yn VARCHAR,  
 icu\_yn VARCHAR,  
 medcond\_yn VARCHAR  
);

# 5. Infrastructure Setup

## 5.1 Database Configuration

PostgreSQL Instance: Neon Cloud Database

* Host: "ep-patient-dawn-advbhv1g-pooler.c-2.us-east-1.aws.neon.tech"
* Database: "neondb"
* SSL: Required
* Connection Pooling: Enabled with pre-ping validation

## 5.2 Environment Configuration

# Required environment variable  
os.environ["PG\_URL"] = (  
 "postgresql+psycopg2://neondb\_owner:npg\_2wdZpRaF0LmG@"  
 "ep-patient-dawn-advbhv1g-pooler.c-2.us-east-1.aws.neon.tech/"  
 "neondb?sslmode=require"  
)

## 5.3 Dependencies

pip install pandas sqlalchemy psycopg2-binary requests matplotlib pytest

Core Dependencies:

* "pandas": Data manipulation and analysis
* "sqlalchemy": Database ORM and connection management
* "psycopg2-binary": PostgreSQL adapter
* "requests": HTTP library for API calls
* "pytest": Testing framework

# 6. Data Pipeline Components

## 6.1 Database Engine (`get\_engine()`)

**Purpose: Create and configure PostgreSQL connection**

**Features:**

* Environment variable validation
* Connection pooling with pre-ping
* Graceful error handling for missing configuration

**Error Conditions:**

* Missing "PG\_URL" environment variable → "SystemExit"

## 6.2 API Client (`fetch\_page()`)

**Purpose: Retrieve paginated data from CDC API**

**Parameters:**

* "limit": Number of records to fetch
* "offset": Starting position in dataset
* "where": Optional SQL-like filter clause

**Features:**

* Configurable timeout (60 seconds)
* Rate limiting detection and handling
* Comprehensive error handling

**Return Value: List of dictionaries (JSON records) or None on error**

## 6.3 Data Transformation (`transform\_rows\_to\_df()`)

**Purpose: Convert API JSON response to structured DataFrame**

**Features:**

* Schema validation and enforcement
* Missing column handling
* Empty dataset handling

**Process:**

1. Check for empty input → return empty DataFrame with correct schema
2. Convert JSON records to DataFrame
3. Ensure all required columns exist
4. Select and reorder columns according to "KEEP\_COLS"

## 6.4 Data Cleaning (`clean\_df()`)

**Purpose: Standardize and clean raw data for analysis**

Cleaning Operations:

### Deduplication

df = df.drop\_duplicates()

### Date Processing

df["cdc\_case\_earliest\_dt"] = pd.to\_datetime(df["cdc\_case\_earliest\_dt"], errors="coerce")

### Text Standardization

# State codes: trim whitespace and convert to uppercase  
df["res\_state"] = df["res\_state"].astype(str).str.strip().str.upper()  
  
# Other text fields: trim whitespace  
for col in text\_columns:  
 df[col] = df[col].astype(str).str.strip()

### Null Value Handling

# Replace various null representations with actual None  
df[col] = df[col].replace({"None": None, "nan": None, "": None})  
  
# Fill with "Unknown" for categorical analysis  
df[col] = df[col].fillna("Unknown")

## 6.5 Data Loading

### Raw Data Loader (`load\_df\_to\_postgres\_raw()`)

* Target: "covid\_case\_surveillance\_raw" table
* Strategy: Append-only for audit trail
* Batch Size: 10,000 records per chunk

### Clean Data Loader (`load\_df\_to\_postgres()`)

* Target: "covid\_case\_surveillance" table
* Strategy: Append-only for analytical queries
* Optimization: Multi-row insert method

# 7. Data Quality and Validation

## 7.1 Data Quality Checks

### Input Validation

* API Response: JSON structure validation
* Required Fields: Presence of all "KEEP\_COLS"
* Data Types: Appropriate type conversion

### Cleaning Validation

* Duplicate Detection: Row-level deduplication
* Date Validation: Invalid dates converted to NaT
* Categorical Standardization: Consistent value representation

### Output Validation

* Record Count Tracking: Monitor processed vs. inserted records
* Schema Compliance: Ensure all columns match target schema

## 7.2 Data Lineage

The system maintains complete data lineage through:

1. Ingestion Timestamps: Each raw batch tagged with "\_ingested\_at"
2. Audit Trail: Raw table preserves original API responses
3. Batch Tracking: Ability to trace clean records back to raw data

## 7.3 Data Quality Metrics

Metrics Tracked:

* Records fetched per batch
* Records after deduplication
* Invalid date conversions
* Null value imputation counts

# 8. Testing Framework

## 8.1 Test Architecture

Framework: pytest with comprehensive mocking

Test Categories:

1. Unit Tests: Individual function testing
2. Integration Tests: Database interaction testing
3. Error Simulation: Failure scenario testing

## 8.2 Test Coverage

### Database Connection Testing

def test\_get\_engine\_env\_missing(monkeypatch):  
 # Test missing environment variable handling  
   
def test\_get\_engine\_valid(monkeypatch):  
 # Test valid connection creation

### API Client Testing

def test\_fetch\_page\_success(monkeypatch):  
 # Test successful API response handling  
   
def test\_fetch\_page\_rate\_limit(monkeypatch):  
 # Test rate limiting and retry logic

### Data Processing Testing

def test\_transform\_rows\_to\_df\_and\_empty():  
 # Test DataFrame creation with empty and populated data  
   
def test\_clean\_df\_basic():  
 # Test data cleaning operations

### Database Loading Testing

def test\_load\_df\_to\_postgres(monkeypatch):  
 # Test clean data loading  
   
def test\_load\_df\_to\_postgres\_raw(monkeypatch):  
 # Test raw data loading

## 8.3 Test Execution

# Run all tests  
python ingtest.py  
  
# Run with pytest directly  
pytest -q ingtest.py

# 9. Deployment and Operations

## 9.1 Google Colab Deployment

### Step 1: Install Dependencies

!pip -q install pandas sqlalchemy psycopg2-binary requests matplotlib pytest

### Step 2: Configure Environment

import os  
os.environ["PG\_URL"] = "postgresql+psycopg2://..."

### Step 3: Execute Ingestion

!python Ingestion.py

# Conclusion

This COVID-19 data ingestion system provides a robust, scalable solution for processing large healthcare datasets from public APIs. The architecture emphasizes data quality, error resilience, and operational visibility, making it suitable for both research and production environments.

The comprehensive testing framework ensures reliability, while the modular design allows for easy maintenance and enhancement. The dual-table approach (raw + clean) provides both data lineage and analytical flexibility.

Key Achievements:

* Successfully ingests 100k+ records from CDC COVID-19 API
* Implements robust error handling and rate limiting
* Provides comprehensive data cleaning and validation
* Maintains full data lineage through dual-table architecture
* Includes extensive unit test coverage for all components

For additional support or enhancements, refer to the individual module documentation within the source code or contact the development team.