Cloud-Based Data Pipeline Automation with Error Handling & Logging

Purpose: Demonstrates advanced Python skills, best practices, and technical writing

Audience: Technical reviewers, MAANG HR & Engineering teams

Keywords: Python Automation, Cloud Data Pipeline, ETL, Error Handling, Logging, Scalability, Performance Optimization, Data Engineering, API Integration, Documentation, Modular Design

```
# --------------------------
# IMPORT STANDARD AND THIRD-PARTY LIBRARIES
# --------------------------
import os                       # Operating system utilities (file paths, environment variables)
import sys                      # System-specific parameters and functions
import logging                  # Robust logging for production-grade applications
import requests                 # HTTP requests for API integration
import json                     # JSON parsing for data interchange
from datetime import datetime   # Handle timestamps and scheduling
from typing import List, Dict   # Type hints for clean, readable code

# --------------------------
# GLOBAL CONFIGURATION
# --------------------------
# Set up logging configuration with timestamp, log level, and message
```

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)s | %(message)s',
    handlers=[
        logging.FileHandler("data_pipeline.log"),  # Persist logs to a file for monitoring
        logging.StreamHandler(sys.stdout)          # Also print logs to console
    ]
)

# Global constants for API endpoints, credentials, and file paths
API_ENDPOINT = os.getenv("DATA_API_ENDPOINT", "https://api.example.com/data")
API_KEY = os.getenv("DATA_API_KEY", "REPLACE_WITH_SECURE_KEY")  # Never hardcode keys in production
OUTPUT_DIR = "processed_data"
os.makedirs(OUTPUT_DIR, exist_ok=True)  # Ensure output directory exists



# --------------------------
# FUNCTION: fetch_data_from_api
# Description: Retrieves JSON data from a RESTful API endpoint securely.
# Keywords: API Integration, RESTful API, JSON Parsing, Python Requests
# --------------------------
def fetch_data_from_api(endpoint: str, api_key: str) -> List[Dict]:
    """
    Fetches data from a cloud-based API with authentication headers.
```

```
    Args:
        endpoint (str): REST API endpoint URL
        api_key (str): API key for authentication

    Returns:
        List[Dict]: List of JSON objects representing structured data

    Raises:
        requests.exceptions.RequestException: Handles network or HTTP errors gracefully
    """
    try:
        logging.info(f"Initiating API request to: {endpoint}")
        headers = {"Authorization": f"Bearer {api_key}"}
        response = requests.get(endpoint, headers=headers, timeout=15)  # Timeout to avoid hanging
        response.raise_for_status()  # Raise exception for HTTP errors
        data = response.json()       # Parse JSON response
        logging.info(f"Successfully fetched {len(data)} records from API.")
        return data
    except requests.exceptions.RequestException as e:
        logging.error(f"Error fetching data from API: {e}")
        return []  # Return empty list on failure for robustness
```

```
# -------------------------
# FUNCTION: process_data
# Description: Cleans, validates, and transforms raw data into a structured format.
# Keywords: Data Processing, Data Transformation, ETL Pipeline, Data Validation
# -------------------------
def process_data(raw_data: List[Dict]) -> List[Dict]:
    """
    Processes raw JSON data by filtering invalid entries and normalizing fields.

    Args:
        raw_data (List[Dict]): Raw JSON data from API

    Returns:
        List[Dict]: Cleaned and structured data ready for downstream use
    """
    processed_data = []
    logging.info("Starting data transformation and validation process...")

    for record in raw_data:
        # Validate required fields
        if "id" not in record or "value" not in record:
            logging.warning(f"Skipping invalid record: {record}")
            continue

        # Normalize fields for consistent schema
```

```
        processed_record = {
            "id": str(record["id"]),
            "value": float(record["value"]),
            "timestamp": record.get("timestamp", datetime.utcnow().isoformat())
        }
        processed_data.append(processed_record)

    logging.info(f"Data processing complete. {len(processed_data)} valid records prepared.")
    return processed_data



# --------------------------
# FUNCTION: save_data_to_file
# Description: Persists processed data to JSON files for analytics or cloud storage.
# Keywords: Data Storage, File I/O, JSON Export, Cloud Readiness
# --------------------------
def save_data_to_file(data: List[Dict], output_dir: str) -> str:
    """
    Saves processed data to a timestamped JSON file in the output directory.

    Args:
        data (List[Dict]): Processed data to save
        output_dir (str): Directory path to save the file
```

```
    Returns:
        str: Full path of the saved file
    """
    timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    output_file = os.path.join(output_dir, f"processed_data_{timestamp}.json")

    with open(output_file, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=4)

    logging.info(f"Data successfully saved to file: {output_file}")
    return output_file



# --------------------------
# MAIN FUNCTION: Orchestrates the entire ETL pipeline
# Keywords: Automation, Data Pipeline, Python Scripting, Modular Design
# --------------------------
def main():
    """
    Main function to execute end-to-end ETL pipeline:
        1. Fetch data from API
        2. Process and validate data
        3. Save processed data to JSON files
        4. Robust logging for monitoring
    """
```

```
    logging.info("===== Starting ETL Pipeline Execution =====")

    # Step 1: Fetch data
    raw_data = fetch_data_from_api(API_ENDPOINT, API_KEY)

    if not raw_data:
        logging.error("No data retrieved. Exiting pipeline.")
        return

    # Step 2: Process data
    processed_data = process_data(raw_data)

    if not processed_data:
        logging.error("No valid data to save. Exiting pipeline.")
        return

    # Step 3: Save processed data
    save_data_to_file(processed_data, OUTPUT_DIR)

    logging.info("===== ETL Pipeline Execution Completed Successfully =====")


# --------------------------
# ENTRY POINT: Pythonic standard to ensure script runs only when executed directly
# --------------------------
```

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