Boon Solution Technical Document

Project: Gitlab API Extraction

Name: Nguyen Nam Tung
Email: nguyennamtung123@gmail.com
Phone Number: 0405335420

Table of Content

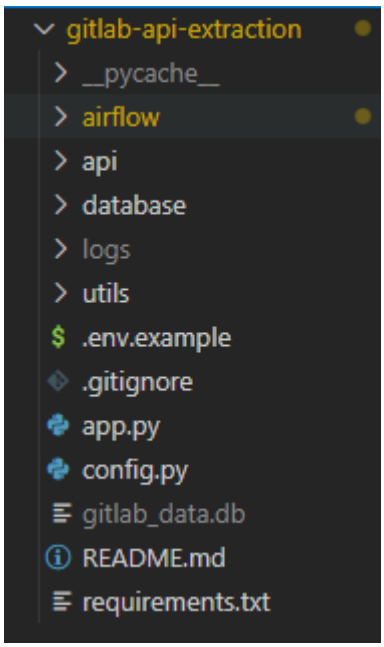
Table of Content **2**
Introduction **3**
Project Structure **3**
Project Outcome **4**
Requirements Fulfillment **6**
Use of requirements.txt 7
Readme.md 7
Use of logger library 7
Error handling (try, catch) 8
Use of database 9
Database record management (CRUD) 10
Code commentary 10
Code quality 11
Secure coding 11
Use of SQL Alchemy 12
Use of OpenAPI library 13
Additional Implementation: Airflow **14**
Future Implementation and Improvement **19**

Introduction

This document outlines the development process for the assignment submitted for the Graduate Developer role at Boon Solutions. It highlights what I have learned throughout the project, including an additional automation feature demonstrating how to set up an automated workflow using Apache Airflow.

Additionally, I will share alternative approaches and recommendations on how the system can be further enhanced for a more complete and scalable solution. Due to time constraints, I was unable to implement all the additional features; however, this document provides an overview of potential improvements and technologies that could be integrated in future iterations.

Project Structure

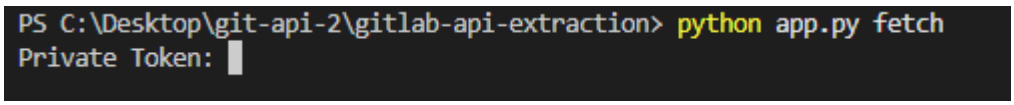


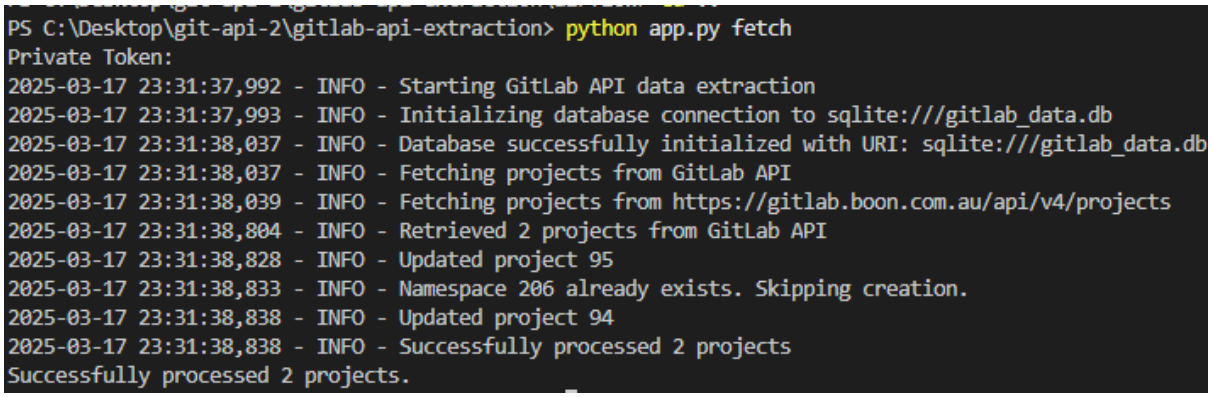
For the main part of this project, we will not be using any components from the airflow/ directory. The airflow/ directory is dedicated to the implementation of an automated data pipeline using Astronomer and Apache Airflow, which is covered separately as an additional automation feature. Therefore, this section will focus solely on the core project structure and functionality, excluding the Airflow automation setup.

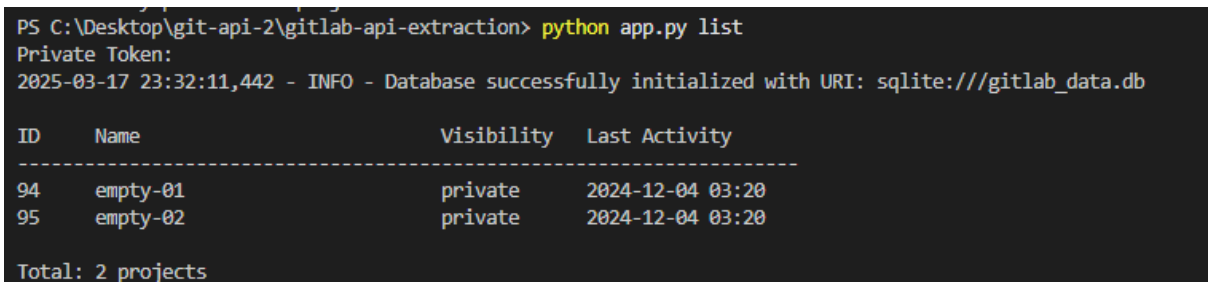
Below is a table that outlines the main folders and files in the project, along with their purpose:

File / Folder	Purpose
airflow	Airflow project
api	GitLab API client and request handlers
database	SQLAlchemy models, DB sessions, queries
logs	Logs for debugging
utils	Utility functions
.env.example	Example environment variables for configuration
README.md	Project documentation
requirements.txt	Dependencies to run the app and Airflow
app.py	Core logic to fetch and store projects
config.py	Configuration and secrets loader

Project Outcome







```
PS C:\Desktop\git-api-2\gitlab-api-extraction> python app.py -h
Private Token:
usage: app.py [-h] {fetch,list,show,update,delete} ...

GitLab API Data Extraction Tool

positional arguments:
  {fetch,list,show,update,delete}
                        Command to execute
    fetch               Fetch projects from GitLab API
    list                List all projects in the database
    show                Show details of a specific project
    update              Update a project in the database
    delete              Delete a project from the database

options:
  -h, --help            show this help message and exit
```

```
PS C:\Desktop\git-api-2\gitlab-api-extraction> python app.py show 94
Private Token:
2025-03-17 23:32:58,960 - INFO - Database successfully initialized with URI: sqlite:///gitlab_data.db

Project Details:
ID:                 94
Name:               empty-01
Description:        N/A
Full Path:          dumpsite/candidate/empty-01
Web URL:            https://gitlab.boon.com.au/dumpsite/candidate/empty-01
Visibility:         private
Created At:         2024-12-04 03:20:16.890000
Last Activity:      2024-12-04 03:20:16.848000
Archived:           False
```

```
Archived:           False
PS C:\Desktop\git-api-2\gitlab-api-extraction> python app.py delete 94
Private Token:
2025-03-17 23:33:42,605 - INFO - Database successfully initialized with URI: sqlite:///gitlab_data.db
Are you sure you want to delete project 'empty-01' (ID: 94)? [y/N]: y
2025-03-17 23:33:47,433 - INFO - Deleted project 94
Project 'empty-01' (ID: 94) has been successfully deleted.
PS C:\Desktop\git-api-2\gitlab-api-extraction>
```

```
PS C:\Desktop\git-api-2\gitlab-api-extraction> python app.py update 95
Private Token:
2025-03-17 23:34:16,420 - INFO - Database successfully initialized with URI: sqlite:///gitlab_data.db
Current project details:
ID:                 95
Name:               empty-02
Description:        N/A
Full Path:          dumpsite/candidate/empty-02
Web URL:            https://gitlab.boon.com.au/dumpsite/candidate/empty-02
Visibility:         private
Enter new name (leave empty to keep current):
```

Requirements Fulfillment

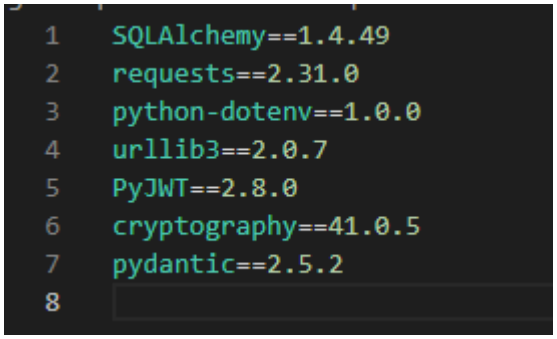
In this section, I will outline all the requirements specified for the assignment and explain the methods and techniques I used to satisfy each of them. Each requirement is addressed with a clear implementation strategy to demonstrate how the project fulfills the objectives

Criteria

Criteria	Points
Use of a requirements.txt	1
Readme.md	1
Use of logger library	1
Error handling (try, catch)	1
Use of a database	1
Database record management (CRUD)	1
Code commentary	1
Code quality	1
Secure coding	2
Use of SQL Alchemy	2
Use of OpenAPI library	2
Total Points	**14**

Use of requirements.txt

The project includes a comprehensive requirements.txt file that lists all necessary Python dependencies required to run the application. This ensures a smooth and consistent setup of the development environment, allowing for easy installation of all libraries and packages through a single command.



```
SQLAlchemy==1.4.49
requests==2.31.0
python-dotenv==1.0.0
urllib3==2.0.7
PyJWT==2.8.0
cryptography==41.0.5
pydantic==2.5.2
```

Readme.md

This README.md provides comprehensive documentation, including the following sections:

- **Project Overview and Description**: A brief introduction to the purpose, functionality, and objectives of the project.
- **Installation Instructions:** Step-by-step guidance on how to set up and install the project dependencies.
- **Usage Examples:** Detailed examples demonstrating how to use the project's features and run the application.
- **Project Structure Explanation:** A breakdown of the project's folder and file organization with explanations of their purposes.
- **Security Considerations:** An overview of the security practices implemented, such as environment variable usage and sensitive data handling.
- **Development Notes:** Additional notes and recommendations for future improvements, as well as insights gained during the development process.

Use of logger library

The project includes a robust custom logging system implemented in utils/logger.py, providing enhanced visibility and traceability throughout the application. Key features of the logging system include:

- **Configurable Log Levels:** Log levels can be easily configured via environment variables, allowing flexibility between development and production environments.

- **Dual Output Logging:** Supports both console and file-based logging to ensure logs are accessible during runtime and persisted for future analysis.
- **Structured Log Format:** Each log entry includes timestamps, log levels, and source file information for improved readability and debugging efficiency.
- **Log File Rotation:** Automatically rotates log files and appends timestamps to maintain organized log archives and prevent files from growing too large.
- **Context-Specific Logging:** Logging is implemented throughout the application in a context-aware manner, providing meaningful insights and aiding in troubleshooting.

Error handling (try, catch)

The application implements comprehensive error handling throughout the codebase to ensure stability, reliability, and ease of troubleshooting. Key aspects of the error handling strategy include:

- **Robust Database Error Handling:** All database operations are wrapped in try/except blocks to gracefully handle potential issues such as connection failures or query errors.
- **API Request Error Handling:** API calls include exception handling for network-related issues, timeouts, and invalid or unexpected responses from the GitLab API.
- **Command Execution Safeguards:** Critical command executions are protected with appropriate exception handling to prevent unexpected crashes and ensure smooth execution.
- **Detailed Exception Logging:** All caught exceptions are logged with comprehensive contextual information, including error details and the specific part of the code where the exception occurred.
- **User-Friendly Console Feedback:** Clear and informative error messages are displayed in the console, ensuring that users are aware of issues without exposing sensitive technical details.

Example from api/client.py:

```python
def get_projects(self, params=None):
    """
    Get projects from the GitLab API.

    Args:
        params (dict, optional): Query parameters for the API request

    Returns:
        list: List of project dictionaries

    Raises:
        requests.RequestException: If API request fails
    """
    try:
        url = f"{self.api_url}/projects"
        logger.info(f"Fetching projects from {url}")

        response = requests.get(url, headers=self.headers, params=params)
        response.raise_for_status()

        projects = response.json()
        logger.info(f"Retrieved {len(projects)} projects from GitLab API")

        # Log sample data (for debugging)
        if projects:
            logger.debug(f"Sample project data: {json.dumps(projects[0], indent=2)}")

        return projects

    except requests.RequestException as e:
        logger.error(f"Error fetching projects from GitLab API: {str(e)}")
        # Re-raise exception after logging
        raise

def get_project(self, project_id):
    """
    Get a specific project by ID.
```

Use of database

The project features a well-structured and efficient database solution, designed to ensure reliable and persistent data storage. The key components of the database implementation include:

- **SQLite Database for Persistent Storage:** A lightweight, file-based SQLite database is used to persistently store GitLab project data, suitable for local development and testing environments.
- **Well-Defined Schema and Relationships:** The database schema is thoughtfully designed, incorporating appropriate relationships and constraints to maintain data integrity and consistency.
- **Configurable Database URI:** The database connection URI is configurable via environment variables, enabling easy transitions between development, testing, and production environments.
- **Database Initialization and Connection Management:** The system includes proper database initialization routines and efficient connection management, ensuring stable and consistent access to the database.

Here is the database Structure Overview:

- **Project Table:** Stores detailed information about GitLab projects, including project names, descriptions, creation dates, and other relevant metadata.
- **Namespace Table:** Manages namespace information associated with GitLab projects and includes foreign key relationships to maintain referential integrity with the Project table.
- **Data Types and Constraints:** Each table field is defined with appropriate data types and constraints to enforce data validity and consistency.

Database record management (CRUD)

The project implements complete CRUD (Create, Read, Update, Delete) operations through the ProjectRepository class, providing a clean and organized interface for managing project data within the database.

CRUD Method Overview:
Create:
create_project() adds new project records to the database.
Read:
get_project_by_id() retrieves a specific project by its ID.
get_all_projects() fetches all project records from the database.
Update:
update_project() modifies existing project details based on specified parameters.
Delete:
delete_project() removes a project record from the database by ID.
Each CRUD operation is implemented with:

- **Proper error handling:** Ensures exceptions are caught and logged appropriately.
- **Transaction management:** Guarantees data consistency by committing or rolling back transactions as needed.
- **Input validation:** Verifies input data before performing operations to maintain data integrity.

Code commentary

The project maintains clear and comprehensive source code documentation to ensure readability, maintainability, and ease of understanding for future developers. The documentation strategy includes the following key practices:

- **Module-Level Docstrings:** Each module begins with a detailed docstring that explains its overall purpose, functionality, and how it fits into the project structure.
- **Class and Method Docstrings:** Every class and method includes descriptive docstrings outlining their purpose, expected parameters, return values, and any exceptions that may be raised during execution.

- **Inline Comments for Complex Logic:** Complex or non-obvious code logic is accompanied by concise inline comments to clarify implementation details and reasoning.
- **Consistent and Standardized Formatting:** All docstrings follow standard Python documentation conventions (PEP 257), ensuring consistency in formatting and structure across the entire codebase.

Code quality

The project follows high-quality coding standards to ensure readability, maintainability, and scalability. Key practices and principles implemented include:

- **Adherence to PEP 8:** The entire codebase follows Python's official PEP 8 style guidelines, ensuring consistency in formatting and readability.
- **Consistent Naming Conventions:** Variables, functions, classes, and modules are named consistently and descriptively, following Python naming standards.
- **Proper Code Organization and Module Structure:** The project is structured logically into well-defined modules, each with a clear purpose and responsibility.
- **DRY (Don't Repeat Yourself) Principle:** Code duplication is minimized by extracting reusable logic into shared functions or classes, promoting efficiency and reducing maintenance overhead.
- **Single Responsibility Principle (SRP):** Each class and function is designed to perform a single, well-defined task, improving modularity and simplifying testing.
- **Appropriate Use of Python Idioms**: Modern and efficient Python idioms and constructs are used throughout the project to write clear, concise, and pythonic code.
- **Clear and Maintainable Code Structure:** The project emphasizes clarity and maintainability, making it easy for future developers to understand and extend the codebase.

Secure coding

1. Credential Management
    - **Secure API Token Handling:** API tokens are never hardcoded in the source code. Instead, they are stored in environment variables and loaded securely at runtime.
    - **Environment Configuration:** Sensitive credentials are managed through .env files, with an .env.example template provided to guide setup without exposing actual secrets.
2. Input Validation and Sanitization
    - **API Data Validation:** All incoming data from API requests is thoroughly validated before processing to ensure it meets expected formats and structures.
    - **Database Input Filtering:** Data inputs for database operations are filtered to allow only permitted fields, reducing the risk of unexpected or malicious data insertion.
3. SQL Injection Prevention

- **Use of SQLAlchemy ORM:** All database interactions are performed through SQLAlchemy's Object Relational Mapper (ORM), which automatically uses parameterized queries to prevent SQL injection.
- **No Raw SQL Execution:** The application avoids using raw SQL statements, eliminating risks associated with manual query building.

4. Secure Error Handling
   - **Detailed Logging Without Sensitive Exposure:** Logs capture sufficient detail for debugging but avoid revealing sensitive data such as API tokens or internal system details.
   - **User-Friendly Error Messages:** Errors presented to end-users are concise and do not expose internal implementation or configuration information, preventing leakage of system details.
5. Database Security
   - **Connection Pooling with Pre-Ping Validation:** Ensures that database connections are valid before use, reducing the risk of unexpected connection failures.
   - **Transaction Management:** All database operations are wrapped in transactions with proper commit/rollback handling to ensure data consistency and integrity.
6. Secure Defaults
   - **Minimal Logging in Production:** Logging verbosity is reduced in production environments to avoid unnecessary data exposure while maintaining sufficient operational insight.
   - **Secure Configuration Defaults:** Default settings prioritize security, such as enabling pre-ping for database connections and disabling debugging features in production.

Use of SQL Alchemy

The project takes full advantage of SQLAlchemy's capabilities to manage database interactions efficiently and maintain a well-structured, scalable data model. Below is an overview of how SQLAlchemy is utilized in the project:

1. ORM Models
   - **Declarative Base Models:** The database schema is defined using SQLAlchemy's declarative base system, enabling a clean and concise representation of the database structure through Python classes.
   - **Proper Model Relationships:** A many-to-one relationship is established between the Project and Namespace models, where multiple projects are associated with a single namespace.
   - **Appropriate Column Types and Constraints:** Each table column is explicitly defined with appropriate data types (e.g., Integer, String), constraints (e.g., nullable=False), and default values to ensure data integrity and consistency.
2. Session Management
   - **Proper Session Initialization and Lifecycle Handling:** Sessions are created and managed in a centralized manner, ensuring they are properly committed, rolled back on error, and closed after each operation.
   - **Connection Pooling and Configuration:** The application benefits from SQLAlchemy's connection pooling to optimize database performance and manage

resource usage efficiently. Connection health checks (e.g., pre-ping) are also enabled to maintain stability.

3. Query Operations
    - **ORM-Based Queries:** All data retrieval and manipulation operations are performed through SQLAlchemy's ORM query interface, promoting safety, readability, and maintainability.
    - **Filtering, Relationship Traversal, and Joins:** The system performs complex queries that include filtering, joining tables, and traversing relationships between models, all using SQLAlchemy's expressive ORM API.
4. Relationships
    - **Foreign Key Constraints:** Referential integrity is enforced through foreign key constraints, ensuring that each Project is correctly associated with a valid Namespace.
    - **Cascading Deletes:** Cascade behavior is configured so that deleting a Namespace will automatically delete all associated Project records, preventing orphaned data.
    - **Bidirectional Relationships with back_populates:** Relationships between Project and Namespace models are bidirectional, allowing for easy navigation and querying from either side of the relationship.

Use of OpenAPI library

The project leverages the OpenAPI specification and tooling to streamline API integration, ensure type safety, and improve maintainability. Below are the key areas where OpenAPI is utilized:

1. Client Generation
    - **Automated Client Generation:** The project uses openapi-generator-cli to generate a Python client from GitLab's official OpenAPI specification.
    - **Simplified API Handling:** The generated client code manages request formatting, authentication, and response parsing, reducing manual effort and potential errors.
2. API Integration
    - **Custom API Client:** A custom client, implemented in api/client.py, extends and works alongside the OpenAPI-generated client to provide additional functionality and abstraction specific to the project's needs.
    - **Robust Error Handling:** Comprehensive error handling is implemented for all API interactions, ensuring stability and clear logging of any issues encountered during API calls.
3. Documentation
    - **Instructions in README:** The project's README.md includes step-by-step instructions for generating API clients from OpenAPI specifications using openapi-generator-cli.
    - **Reference to GitLab OpenAPI Spec:** A direct link to GitLab's OpenAPI specification is provided for transparency and to enable users to regenerate or customize the client as needed.
4. Type Safety and Validation
    - **Type-Safe Models:** The OpenAPI-generated models enforce type safety across API interactions, ensuring that request and response data adheres to the defined schemas.

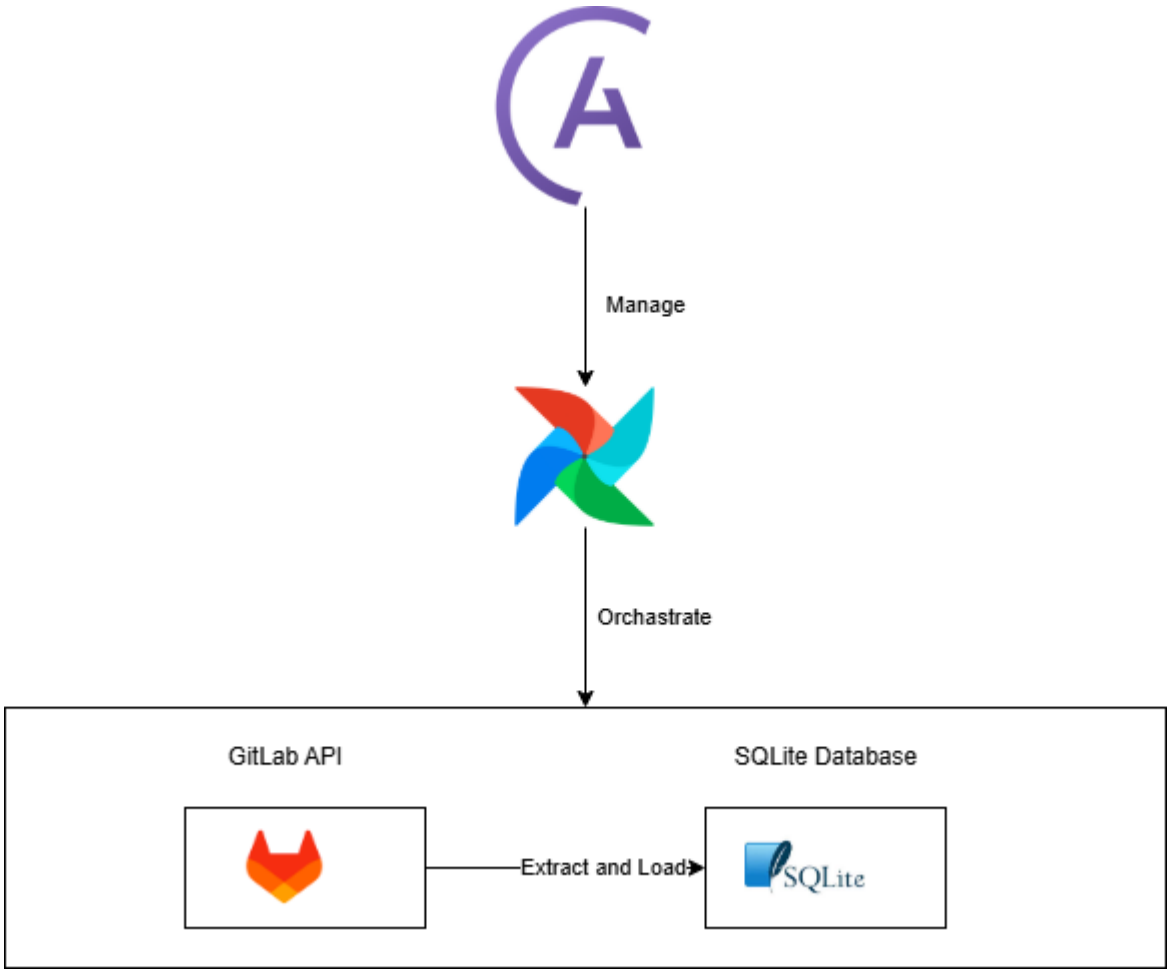
- **Request Validation:** Request payloads are validated based on OpenAPI schemas, reducing the risk of sending invalid or malformed data to the API.

Additional Implementation: Airflow

To automate the process of extracting and storing data from GitLab, I introduced an integration with Apache Airflow, a powerful data orchestration tool designed for automating ETL (Extract, Transform, Load) workflows. By leveraging Airflow, the extraction and storage tasks are executed on a defined schedule without the need for manual intervention.

For easier management, scalability, and deployment of Airflow, I utilized Astronomer, which provides a streamlined platform for running and managing Airflow environments. Astronomer simplifies environment configuration, version control, and monitoring, making the workflow orchestration more efficient and maintainable.

Below is a diagram illustrating the architecture and flow of the automated pipeline I have implemented:



Step 1: Install Docker

Make sure Docker is installed and running on your machine. You can download Docker from Docker's official website.

Step 2: Install Astro CLI
Open PowerShell as an administrator and run the following commands to download and install the Astro CLI:

```
Invoke-WebRequest -UseBasicParsing https://install.astronomer.io -OutFile astro-installer.ps1
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
.\astro-installer.ps1
```

Step 3: Verify Installation
After the installation is complete, confirm the Astro CLI is installed correctly by running:

```
astro version
```

You should see the installed version of Astro CLI displayed.

Step 4: Initialize Your Astro Project
To create a new Astro project, run the following command:

```
astro dev init
```

This will set up the necessary files and folder structure for your Astro project.

Create a DAG that imports and uses the function from app.py to fetch GitLab data. Save the DAG file in the dags folder within your project directory.

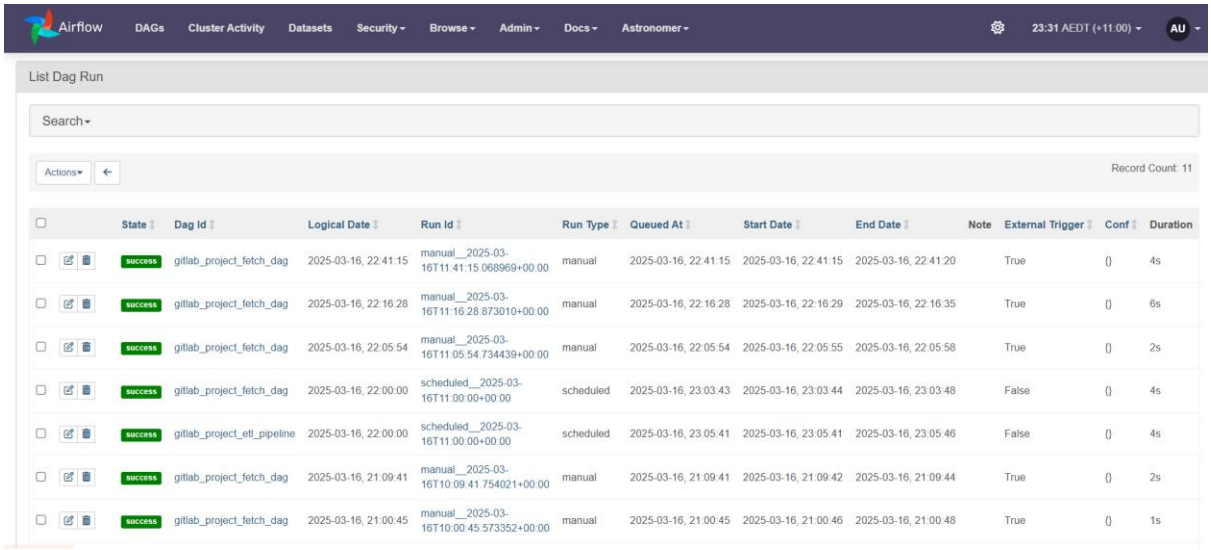
```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta
import sys
import os

sys.path.append('/usr/local/airflow')

from app import fetch_and_store_projects

default_args = {
    'owner': 'tung',
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Define the DAG
with DAG(

    dag_id='gitlab_project_fetch_dag',
    description='Fetch GitLab projects and store them in the database',
    default_args=default_args,
    start_date=datetime(2025, 3, 1),
    schedule_interval='@hourly',
    catchup=False,
    is_paused_upon_creation=False
) as dag:


    fetch_projects_task = PythonOperator(
        task_id='fetch_gitlab_projects',
        python_callable=fetch_and_store_projects
    )


    fetch_projects_task
```

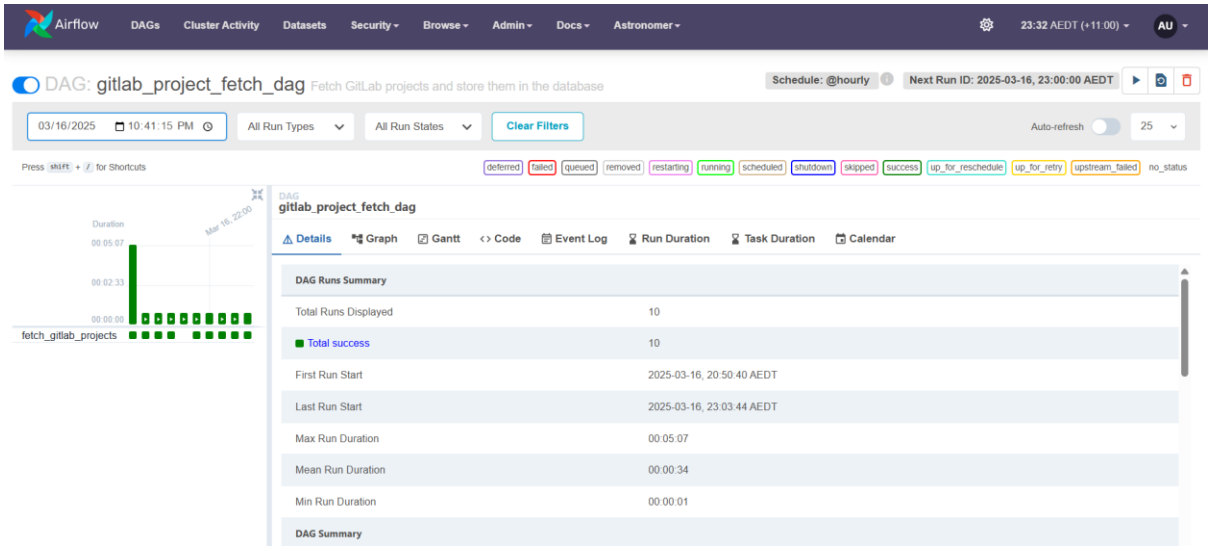
Edit the Docker file for setup and run command astro dev start to run the Airflow server on localhost:

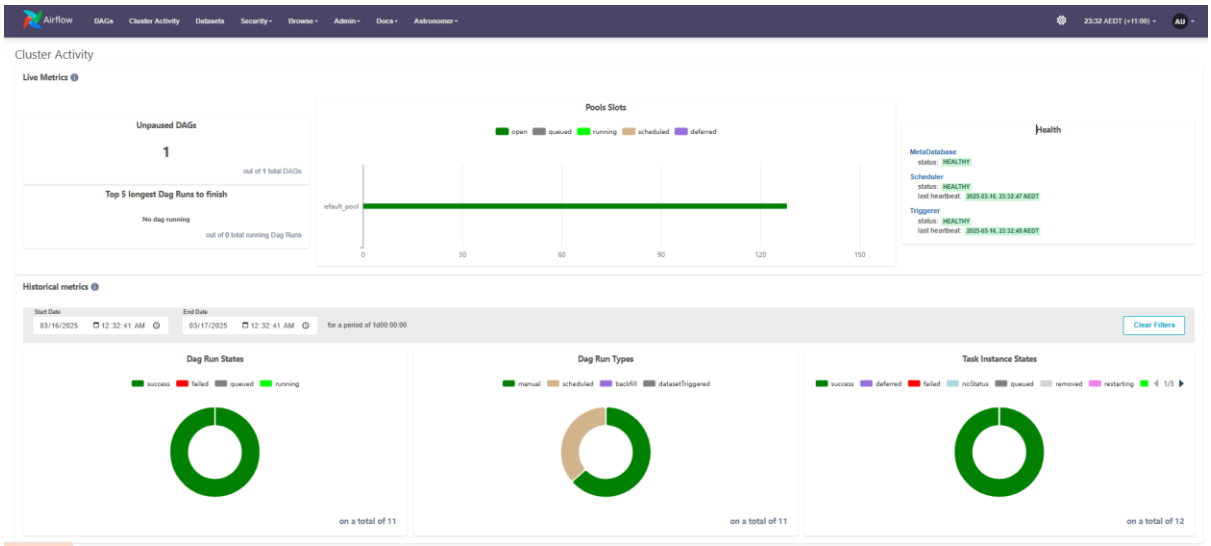
ow > Dockerfile

```
# Use the Astronomer Certified Airflow Runtime image
FROM quay.io/astronomer/astro-runtime:12.7.1

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

# Copy your external project files into the Airflow container
COPY api/ /usr/local/airflow/api/
COPY database/ /usr/local/airflow/database/
COPY utils/ /usr/local/airflow/utils/
COPY app.py /usr/local/airflow/app.py
COPY config.py /usr/local/airflow/config.py


# Expose Airflow default webserver port
EXPOSE 8080
```

Now you can run the DAG either on a schedule or trigger it manually, on the Airflow server:



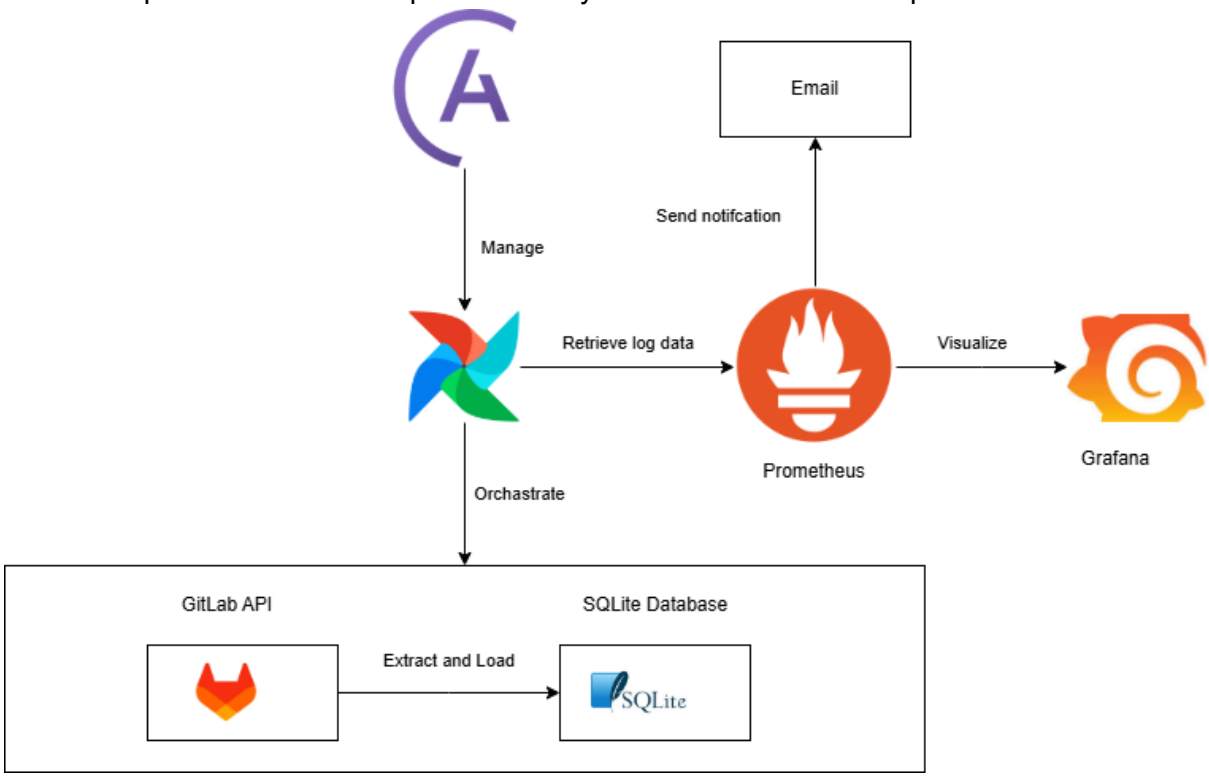

Airflow
DAGs
Cluster Activity
Datasets
Security
Browse
Admin
Docs
Astronomer
23:32 AEDT (+11:00)
AU
DAG: gitlab_project_fetch_dag
Fetch GitLab projects and store them in the database
Schedule: @hourly
Next Run ID: 2025-03-16, 23:00:00 AEDT
03/16/2025 10:41:15 PM
All Run Types
All Run States
Clear Filters
Auto-refresh
25
Press shift + / for Shortcuts
deferred
failed
queued
removed
restarting
running
scheduled
shutdown
skipped
success
up_for_reschedule
up_for_retry
upstream_failed
no_status
Duration
Mar 16, 22:00
00:05:07
00:02:33
00:00:00
fetch_gitlab_projects
DAG
gitlab_project_fetch_dag
Details
Graph
Gantt
Code
Event Log
Run Duration
Task Duration
Calendar
DAG Runs Summary
Total Runs Displayed
10
Total success
10
First Run Start
2025-03-16, 20:50:40 AEDT
Last Run Start
2025-03-16, 23:03:44 AEDT
Max Run Duration
00:05:07
Mean Run Duration
00:00:34
Min Run Duration
00:00:01
DAG Summary


Airflow
Cluster Activity
Live Metrics
Unpaused DAGs
1
out of 1 total DAGs
Top 5 longest Dag Runs to finish
No dag running
out of 0 total running Dag Runs
Pools Slots
open
queued
running
scheduled
deferred
default_pool
0
30
60
90
120
150
Health
MetaDatabase
status: HEALTHY
Scheduler
status: HEALTHY
last heartbeat: 2025-03-16, 23:32:47 AEDT
Triggerer
status: HEALTHY
last heartbeat: 2025-03-16, 23:32:49 AEDT
Historical metrics
Start Date
03/16/2025 12:32:41 AM
End Date
03/17/2025 12:32:41 AM
for a period of 1d00:00:00
Clear Filters
Dag Run States
success
failed
queued
running
on a total of 11
Dag Run Types
manual
scheduled
backfill
dataset_triggered
on a total of 11
Task Instance States
success
deferred
failed
no_status
queued
removed
restarting
1/3
on a total of 12


DAG: gitlab_project_fetch_dag
Fetch GitLab projects and store them in the database
Schedule: @hourly
Next Run ID: 2025-03-16, 23:00:00 AEDT

Future Implementation and Improvement

This is a plan for a more sophisticated system that I want to implement in the future:



In the future, I plan to implement a more efficient and scalable system by integrating Prometheus to collect metrics and send automated email alerts to users. The system will also leverage Grafana dashboards for real-time data visualization and monitoring. To ensure security and high availability, the entire solution will be hosted on AWS within a secure cloud infrastructure.