Comands

### **Updated Understanding**

From your request and the previous discussion, you are looking for a **comprehensive guide** that not only details the **Airflow and Airbyte integration** but also provides **best practices for SSH access via VS Code for development** on a GCP VM. You want this documentation to address the needs of both beginners and advanced users, with clear **guidelines, edge cases**, and **best practices** to ensure security, scalability, and maintainability.

Here’s the updated understanding of what you want:

### **What You’re Asking For**

1. **Airflow-Airbyte Integration**:
   * How to configure Airbyte workflows to trigger Airflow DAGs (using APIs or other mechanisms).
   * Step-by-step instructions for setting up both services with isolated users and permissions.
   * Secure communication and troubleshooting between the two services.
2. **SSH Access for Development via VS Code**:
   * How to configure **Remote-SSH** in VS Code to connect securely to a GCP VM.
   * Best practices for using SSH in a development workflow:
     + Securely managing SSH keys.
     + Avoiding common pitfalls, like exposing private keys or using weak passwords.
     + Guidelines for handling multiple users on the VM.
3. **GCP VM Setup**:
   * How to set up a GCP VM, manage users, configure firewall rules, and ensure security (e.g., disabling root login, setting up least-privilege access).
   * How to scale the VM for Airflow and Airbyte workloads.
4. **Documentation Style**:
   * Professional and **mentor-like tone**, including a "mindset and approach" section that explains the principles behind each decision.
   * Clear "dos and don'ts" to help developers avoid pitfalls.
   * Edge cases to cover common errors and troubleshooting tips.

### **Plan for Documentation**

Here’s how I’ll structure the guide:

#### **1. Mindset and Approach**

* The philosophy behind secure development workflows with SSH and VS Code.
* Why Airflow and Airbyte should be deployed together with isolation and secure communication.
* Key principles for working with GCP, SSH, and service integrations.

#### **2. Prerequisites**

* Tools required (VS Code, GCP access, SSH keys, Docker, Python, etc.).
* High-level overview of the architecture (how Airbyte triggers Airflow, user isolation, API interaction).

#### **3. GCP VM Setup**

* Step-by-step guide to creating and configuring a VM on GCP.
* Firewall rules and security configurations.
* Setting up Linux users for Airflow and Airbyte.

#### **4. SSH Access and VS Code Setup**

* Generating SSH keys securely and sharing them with the GCP VM.
* Configuring VS Code’s Remote-SSH extension for seamless development.
* Best practices for managing SSH keys and user roles.

#### **5. Airflow and Airbyte Deployment**

* Installing and configuring Airflow and Airbyte on the VM.
* Assigning proper file and directory permissions.
* Starting and testing the services.

#### **6. Integration: Airflow Triggers Airbyte**

* Step-by-step guide to using Airbyte’s API or webhooks to trigger Airflow DAGs.
* Secure API setup with authentication (e.g., basic auth, tokens).
* Monitoring and debugging the integration.

#### **7. Best Practices and Do’s/Don’ts**

* For SSH:
  + Always use key-based authentication.
  + Disable root login and password-based access.
  + Rotate keys regularly.
* For Airflow and Airbyte:
  + Use separate users for each service.
  + Avoid running services as root.
  + Secure APIs with authentication and firewalls.
* For GCP:
  + Use IAM roles for access management.
  + Enable audit logging for tracking changes.

#### **8. Edge Cases and Troubleshooting**

* SSH connection issues (e.g., permission denied, key not working).
* Airflow-Airbyte integration failures (e.g., API errors, authentication issues).
* Resource scaling on GCP (e.g., VM running out of memory, high CPU usage).

### **Next Steps**

If this matches your needs, I’ll start drafting the full documentation. It will include detailed commands, explanations, and examples for each section, written in a professional, mentor-like tone. If you’d like, I can also prioritize **SSH via VS Code** or **Airflow-Airbyte integration** first, depending on your preference.

Let me know if this aligns with your vision, or if there’s anything else you’d like me to refine or include!

1 . Mindset and Approach

### **1 . Mindset and Approach**

When setting up a robust data infrastructure where **Airbyte** and **Airflow** work in tandem on a **GCP VM**, it's essential to adopt a structured and secure mindset. This section outlines the guiding principles, strategic goals, and best practices that will ensure a scalable, secure, and developer-friendly setup.

#### **The Big Picture**

1. **Why Integrate Airflow and Airbyte?**
   * **Airbyte** handles data movement, syncing data between systems like databases, APIs, or cloud storage.
   * **Airflow** orchestrates workflows, ensuring tasks like triggering data syncs, running transformations, or notifying teams happen in the right order and on schedule.
   * By combining the two, you create an end-to-end pipeline where Airflow triggers Airbyte jobs, monitors progress, and ties them into larger workflows.
2. **Role of the GCP VM**
   * The GCP VM is your controlled environment for running Airflow and Airbyte securely. It acts as the backbone for data processing, offering compute power, flexibility, and scalability.
   * Proper configuration ensures this VM remains secure, performant, and developer-friendly.

#### **Guiding Principles**

1. **Security by Design**
   * Protect your environment by adhering to the principle of least privilege:
     + **Separate users**: Run Airbyte and Airflow as different Linux users, each with minimal permissions to isolate their scope.
     + **Secure communication**: Use SSH key-based access for developers, disable root login, and firewall unnecessary ports.
     + **Secure APIs**: Use authentication mechanisms (like API tokens) for Airflow-Airbyte interactions.
2. **Isolation and Modularity**
   * Deploy Airbyte and Airflow as separate services on the same VM, but isolate their processes, logs, and configurations to make troubleshooting easier and minimize impact if one service fails.
   * Keep workflows modular—Airbyte should focus on syncing data, while Airflow oversees orchestration and integration into larger pipelines.
3. **Developer-Focused Environment**
   * Ensure developers can access and work on the system easily without compromising security:
     + Use **VS Code Remote-SSH** for direct editing and debugging on the VM.
     + Use environment variables and shared configuration files for flexibility.
4. **Scalability and Maintainability**
   * Design the environment to grow with your needs:
     + Use Docker for containerization if the workload increases and requires deployment across multiple servers.
     + Monitor resource usage on the VM (CPU, memory, disk) and upgrade instances as needed.
     + Log and monitor workflows using Airflow’s and Airbyte’s built-in dashboards.

#### **Goals for This Guide**

By the end of this guide, you will have:

* A **secure and isolated deployment** of Airbyte and Airflow on a GCP VM.
* Configured **SSH access via VS Code**, making it easier for developers to work on workflows and troubleshoot issues.
* Established a **seamless integration** where Airflow orchestrates and triggers Airbyte jobs.
* Followed **best practices** to ensure scalability, security, and ease of maintenance.

#### **Dos and Don’ts**

| **Do** | **Don’t** |
| --- | --- |
| Use **SSH key-based authentication** for accessing the VM. | Never enable password-based SSH access or share private keys openly. |
| Separate users and processes for Airflow and Airbyte for isolation and security. | Avoid running both services under the same user or as root. |
| Configure firewalls to allow only required traffic (e.g., SSH, Airflow, Airbyte ports). | Don’t expose all VM ports publicly—minimize the attack surface. |
| Regularly update Airflow, Airbyte, and the VM’s packages for security patches. | Avoid using outdated software that might have unpatched vulnerabilities. |
| Monitor VM resource usage and set up logging for both services. | Don’t ignore resource bottlenecks—scale the VM as workflows grow in complexity. |
| Use environment variables to store sensitive configurations (e.g., API keys). | Avoid hardcoding sensitive data like credentials in scripts or workflows. |

This mindset and approach will guide every decision in the setup, ensuring that your Airbyte-Airflow integration is **secure, scalable, and developer-friendly**. The next section will cover **GCP VM setup**, including creating the VM, configuring users, and securing SSH access. Let me know if you're ready to proceed!

2. Prerequisites

### **2. Prerequisites**

Before you begin setting up the Airflow-Airbyte integration on a GCP VM, ensure that you have the necessary tools, accounts, and a high-level understanding of the architecture. This section will guide you through preparing everything you need.

#### **Tools and Resources Required**

1. **Google Cloud Platform (GCP) Access**:
   * A GCP account with permissions to create and manage Compute Engine resources.
   * Billing enabled on the GCP project.
2. **Local Development Environment**:
   * A machine with:
     + **VS Code** installed (latest version recommended).
     + The **Remote - SSH** extension installed.
     + **SSH client** installed (default on Linux/macOS; use PuTTY or install OpenSSH on Windows).
3. **Software Dependencies** (for the VM):
   * **Python 3.7+**: Required for Airflow.
   * **Docker**: Required for Airbyte.
   * **SSH keys**: For secure access to the VM.
4. **Networking**:
   * A stable internet connection.
   * Ensure you can connect to the GCP VM via SSH.

#### **High-Level Architecture**

Here’s an overview of how Airbyte and Airflow will work together:

1. **Airbyte**:
   * Responsible for syncing data from source to destination.
   * Runs as a service on the VM using Docker.
   * Provides an API endpoint for triggering sync jobs.
2. **Airflow**:
   * Orchestrates workflows, including triggering Airbyte jobs via API calls.
   * Runs as a Python-based service on the VM, with its own scheduler and webserver.
3. **Communication**:
   * Airflow triggers Airbyte syncs using HTTP API calls.
   * Logs and configuration files are stored separately for each service.
4. **VS Code Remote-SSH**:
   * Used for editing configuration files, writing Airflow DAGs, and troubleshooting directly on the VM.

#### **Firewall and Networking Requirements**

Configure your firewall and networking settings to allow communication:

* **SSH**: Port 22 (for accessing the VM).
* **Airflow Webserver**: Port 8080 (to access Airflow’s dashboard).
* **Airbyte Web UI**: Port 8000 (to manage Airbyte connections).
* Block all unnecessary ports to minimize the attack surface.

#### **Permissions and Roles**

For security and smooth operation:

* **On GCP**:
  + Use **IAM roles** for access control. A role like Compute Admin is required to manage the VM.
* **On the VM**:
  + Create separate Linux users for Airflow (airflow) and Airbyte (airbyte).
  + Use least-privilege principles: each user should have access only to the directories and files they need.

#### **Preparation Checklist**

Before proceeding to VM setup, ensure you have:

1. A GCP account and project with Compute Engine API enabled.
2. The **Remote - SSH** extension in VS Code.
3. An SSH key pair (public and private) generated locally. If you don’t have one, we’ll cover this in the setup.
4. A clear understanding of the ports to be used for Airflow (8080) and Airbyte (8000).

Now that the prerequisites are in place, the next section will cover **GCP VM setup**, including creating the VM, configuring users, and securing SSH access. Let me know if you're ready to proceed!

3. GCP VM Setup (with Explanations)

### **3. GCP VM Setup (with Explanations)**

This section walks through the creation and configuration of a GCP Virtual Machine (VM) to host **Airflow** and **Airbyte**. Each step includes an explanation of *why* it’s being done and a breakdown of the associated commands.

### **Step 1: Create the GCP VM**

#### **Why?**

The VM serves as the central environment for hosting both Airflow and Airbyte. Using a GCP VM ensures scalability, availability, and flexibility for development and production use.

#### **Steps to Follow**

1. **Navigate to the GCP Console**:
   * The GCP Console is where you manage cloud resources.
   * **Why**: Creating the VM here gives you access to GCP's full suite of tools and features for monitoring, scaling, and networking.
2. **Choose a Machine Type and Configuration**:
   * **Machine Type**: Select e2-medium or higher depending on the workload.
     + **Why**: e2-medium offers 2 vCPUs and 4GB RAM, which is sufficient for moderate workloads. You can scale this later as needed.
   * **Boot Disk**: Select **Debian 11** or **Ubuntu 20.04 LTS**.
     + **Why**: These are lightweight, stable operating systems that are well-suited for hosting services like Airflow and Airbyte.
   * **Firewall**:
     + Check **Allow HTTP and HTTPS traffic** to enable access to the services.

**3. Command to Verify VM Creation** (if using gcloud CLI):  
bash  
CopyEdit  
gcloud compute instances list

* + **What it does**: Lists all your VM instances to confirm that your VM is up and running.

### **Step 2: Configure Firewall Rules**

#### **Why?**

By default, GCP blocks all incoming traffic except for SSH (port 22). To access the Airflow webserver (8080) and Airbyte UI (8000), you need to create rules to allow traffic on these ports.

#### **Steps to Configure**

1. **Add Firewall Rules** (via GCP Console or CLI):
   * Use the GCP Console under **VPC Network → Firewall** to configure.

Alternatively, use the gcloud command:  
bash  
CopyEdit  
gcloud compute firewall-rules create allow-airflow-airbyte-traffic \

--direction=INGRESS \

--priority=1000 \

--network=default \

--action=ALLOW \

--rules=tcp:22,tcp:8080,tcp:8000 \

--source-ranges=0.0.0.0/0 \

--target-tags=airflow-airbyte

* + **What it does**:
    - Opens ports 22, 8080, and 8000 for incoming traffic.
    - --source-ranges=0.0.0.0/0: Allows all IPs (not secure for production—restrict to specific IPs in production).
    - --target-tags=airflow-airbyte: Associates the rule with your VM.

### **Step 3: SSH into the VM**

#### **Why?**

You need SSH access to the VM to install software, configure services, and manage users.

#### **Steps to Connect**

1. **From GCP Console**:
   * Click **SSH** next to your VM in the **Compute Engine → VM Instances** section.
   * **Why**: This provides a browser-based terminal, which is quick and doesn’t require setting up SSH keys locally.

**Using the Command Line**: If you’ve set up the Google Cloud SDK locally:  
bash  
CopyEdit  
gcloud compute ssh <vm-name> --zone=<zone>

* + **What it does**: Opens an SSH session into your VM from your local terminal.
  + Replace <vm-name> with the name of your VM and <zone> with its zone (e.g., us-central1-a).

### **Step 4: Create Linux Users for Airflow and Airbyte**

#### **Why?**

Running services under separate Linux users ensures isolation. If one service is compromised or crashes, the other remains unaffected. This also limits permissions, reducing security risks.

#### **Steps to Create Users**

**Create a User**:  
bash  
CopyEdit  
sudo adduser airflow

* + **What it does**:
    - Creates a new Linux user named airflow.
    - Prompts you to set a password (optional, since SSH keys will be used).
    - Automatically creates a home directory at /home/airflow.

**Repeat for Airbyte**:  
bash  
CopyEdit  
sudo adduser airbyte

* + **Why**: Each service gets its own user, keeping processes and files isolated.

**Set Up .ssh Directories**: Each user needs an .ssh directory to store their authorized keys for SSH access.  
bash  
CopyEdit  
sudo mkdir -p /home/airflow/.ssh

sudo mkdir -p /home/airbyte/.ssh

* + **What it does**: Creates a secure directory for storing SSH keys.

**Set Correct Permissions**:  
bash  
CopyEdit  
sudo chmod 700 /home/airflow/.ssh

sudo chmod 700 /home/airbyte/.ssh

sudo chown -R airflow:airflow /home/airflow/.ssh

sudo chown -R airbyte:airbyte /home/airbyte/.ssh

* + **What it does**:
    - chmod 700: Ensures only the user has access to their .ssh directory.
    - chown: Makes the user (airflow or airbyte) the owner of their .ssh directory.

**Authorized Keys**: Add the public SSH keys for secure access:  
bash  
CopyEdit  
echo "ssh-rsa AAAAB..." | sudo tee /home/airflow/.ssh/authorized\_keys

echo "ssh-rsa AAAAB..." | sudo tee /home/airbyte/.ssh/authorized\_keys

* + **What it does**:
    - Stores the public key in the authorized\_keys file.
    - The VM will use this key to authenticate the user during SSH login.

**Set Permissions for authorized\_keys**:  
bash  
CopyEdit  
sudo chmod 600 /home/airflow/.ssh/authorized\_keys

sudo chmod 600 /home/airbyte/.ssh/authorized\_keys

* + **What it does**:
    - chmod 600: Ensures only the user can read or write their authorized\_keys file.

#### **Summary of Commands**

| **Command** | **Purpose** |
| --- | --- |
| sudo adduser airflow | Creates a user for Airflow. |
| sudo adduser airbyte | Creates a user for Airbyte. |
| sudo mkdir -p /home/<user>/.ssh | Creates a secure directory for storing SSH keys. |
| sudo chmod 700 /home/<user>/.ssh | Sets permissions to secure the .ssh directory. |
| sudo chown -R <user>:<user> /home/<user> | Assigns ownership of the home directory to the user. |
| `echo "ssh-rsa..." | sudo tee /home/<user>/.ssh/authorized\_keys` |

The VM is now ready with isolated Linux users and SSH access configured. In the next section, we’ll set up **Airflow and Airbyte** on this VM. Let me know if you’re ready to proceed!

SSH Key Management

### **SSH Key Management Related to Airflow and Airbyte Setup**

For your Airflow and Airbyte deployment on a GCP VM, managing SSH keys properly ensures secure developer access and service isolation. Here's how you should handle SSH keys to align with the setup:

### **Why SSH Keys are Essential for Airflow and Airbyte Setup**

1. **Secure Developer Access**:
   * Each developer needs access to the GCP VM to edit Airflow DAGs, configure Airbyte connections, and troubleshoot.
   * SSH keys ensure secure, passwordless access for these tasks.
2. **Isolated Users**:
   * You have separate Linux users for airflow and airbyte.
   * Each user can have its own SSH key to maintain isolation, ensuring no unnecessary access between services.
3. **Convenience with Security**:
   * Developers can use **VS Code Remote-SSH** for seamless file editing and debugging directly on the VM.

### **Steps to Generate and Add Missing SSH Keys**

#### **Step 1: Generate SSH Keys for Developers or Services**

Developers or team members who need access to the VM should generate their SSH keys locally.

**Command (Run on Developer’s Machine)**:  
  
ls ~/.ssh →

This command will give a list all keys

bash

CopyEdit

ssh-keygen -t rsa -b 4096 -C "developer\_email@example.com"

ssh-keygen -t rsa -b 4096 -C malik\_hassan

**What This Does**:

* Generates a public-private key pair:
  + **Private key**: Stays securely on the developer's machine.
  + **Public key**: Will be added to the VM for authentication.  
      
      
    Where to save it: https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.z62r8mgb1iu4

#### **Step 2: Add SSH Keys to the GCP VM**

1. **For Airflow User**:

Add the developer’s public key to the airflow user’s authorized\_keys file:  
bash  
CopyEdit  
echo "ssh-rsa AAAAB...developer-key" | sudo tee -a /home/airflow/.ssh/authorized\_keys

sudo chmod 600 /home/airflow/.ssh/authorized\_keys

sudo chown airflow:airflow /home/airflow/.ssh/authorized\_keys

* + **What This Does**:
    - Appends the developer’s public key to the authorized\_keys file.
    - Ensures the airflow user can authenticate using the developer’s private key.

1. **For Airbyte User**:

Similarly, add the developer’s key to the airbyte user:  
bash  
CopyEdit  
echo "ssh-rsa AAAAB...developer-key" | sudo tee -a /home/airbyte/.ssh/authorized\_keys

sudo chmod 600 /home/airbyte/.ssh/authorized\_keys

sudo chown airbyte:airbyte /home/airbyte/.ssh/authorized\_keys

Error: (in case if this occurs)   
malik\_hassan@instance-20250124-201353:~$ sudo chmod 600 /home/airbyte/.ssh/authorized\_keys

chmod: cannot access '/home/airbyte/.ssh/authorized\_keys': No such file or directory

1st make sure, followed the preoviious commands right. Should be good then.   
Still error then: <https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.1xpyq539b7yq>

#### **Step 3: Configure Developers to Access the GCP VM**

On the developer’s machine:

Add the VM’s public IP to the local ~/.ssh/config file for seamless access:  
bash  
CopyEdit  
nano ~/.ssh/config

Add the following entry:  
ini  
CopyEdit  
Host airflow-vm

HostName <vm-ip>

User airflow

IdentityFile ~/.ssh/id\_rsa

Host airbyte-vm

HostName <vm-ip>

User airbyte

IdentityFile ~/.ssh/id\_rsa

Host airflow-vm

HostName 34.130.223.147

User airflow

IdentityFile "C:\Users\Malik Hassan Qayyum\.ssh\id\_airflow"

Host airbyte-vm

HostName 34.130.223.147

User airbyte

IdentityFile "C:\Users\Malik Hassan Qayyum\.ssh\id\_airbyte"

Test the connection:  
bash  
CopyEdit  
ssh airflow-vm

(IN VS Code)

1. This should log the developer into the airflow user on the VM.

#### **Step 4: Set Up VS Code Remote-SSH**

1. **Install the Remote-SSH Extension**:
   * In VS Code, go to **Extensions** and search for **"Remote - SSH"**.
   * Install the extension.
2. **Connect to the VM**:
   * Open the **Command Palette** (Ctrl+Shift+P or Cmd+Shift+P).
   * Select **Remote-SSH: Connect to Host**.
   * Enter the host configured in ~/.ssh/config (e.g., airflow-vm or airbyte-vm).
3. **Work with Files**:
   * Once connected, VS Code will open the remote VM’s file system.
   * You can now directly edit Airflow DAGs or Airbyte configuration files on the VM.

### **Best Practices for SSH Key Management**

1. **Unique Keys for Developers**:
   * Each developer should have their own SSH key added to the VM. This ensures accountability and allows you to revoke individual access if needed.
2. **Rotate Keys Regularly**:
   * Rotate SSH keys periodically (e.g., every 6 months) to maintain security.
3. **Avoid Sharing Keys**:
   * Never share private keys between developers or store them in public repositories.
4. **Secure the VM**:

Disable password authentication in /etc/ssh/sshd\_config:  
ini  
CopyEdit  
PasswordAuthentication no

Restart SSH to apply changes:  
bash  
CopyEdit  
sudo systemctl restart sshd

1. **Restrict Access to Necessary Users**:

Use the AllowUsers directive in /etc/ssh/sshd\_config to limit SSH access to specific users:  
ini  
CopyEdit  
AllowUsers airflow airbyte

### **Key Benefits**

* Secure and passwordless access for developers and services.
* Proper isolation between airflow and airbyte users.
* Convenient editing and debugging using VS Code, directly on the GCP VM.

Let me know if you'd like me to proceed with the **Airflow and Airbyte deployment section**! 😊

Where to save them locally

**ssh-keygen -t rsa -b 4096 -C "developer\_email@example.com"**

**Where to save it?**

When running the ssh-keygen command, you’ll be prompted with:

plaintext

CopyEdit

Enter file in which to save the key (/home/youruser/.ssh/id\_rsa):

### **Where Should You Save It?**

The default location is /home/youruser/.ssh/id\_rsa (or ~/.ssh/id\_rsa), and in most cases, you should **accept the default** by pressing Enter.

For saving in the SSH folder.   
Enter file in which to save the key (C:\Users\Malik Hassan Qayyum/.ssh/id\_rsa): C:\Users\Malik Hassan Qayyum/.ssh/id\_airflow

C:\Users\Malik Hassan Qayyum/.ssh/id\_airbyte

### **Why Use the Default Location?**

1. **Standardized Location**: The SSH client automatically looks for private keys in ~/.ssh, so using the default ensures seamless operation.
2. **Easy Management**: If you already have other keys saved in the same directory, the default location keeps them organized.

### **When to Specify a Custom Location**

You may want to specify a different filename or directory in these cases:

1. **Multiple Projects or VMs**:

If you need separate keys for different VMs, services, or users, you can use a custom name to differentiate them:  
plaintext  
CopyEdit  
/home/youruser/.ssh/id\_airflow

/home/youruser/.ssh/id\_airbyte

1. **Avoid Overwriting Existing Keys**:

If you already have a key at ~/.ssh/id\_rsa and don’t want to overwrite it, specify a unique name:  
plaintext  
CopyEdit  
/home/youruser/.ssh/id\_custom\_project

1. **For a Specific Service**:

When generating a key specifically for airflow or airbyte, you might save it as:  
plaintext  
CopyEdit  
/home/youruser/.ssh/id\_airflow

### **How to Handle Multiple Keys**

If you save the key to a custom location, you need to tell the SSH client which key to use when connecting to the VM. To do this:

Edit your SSH configuration file:  
bash  
CopyEdit  
nano ~/.ssh/config

Add an entry for your VM:  
plaintext  
CopyEdit  
Host airflow-vm

HostName <vm-ip>

User airflow

IdentityFile ~/.ssh/id\_airflow

### **Final Recommendations**

* Use the default location unless you need multiple keys.
* Always remember where you save your key, as you’ll need it for configuration and connection.
* Avoid overwriting existing keys unless you’re certain they’re no longer in use.

Let me know if you’d like further clarification or help with custom locations! 😊

error

The error chmod: cannot access '/home/airbyte/.ssh/authorized\_keys': No such file or directory occurs because the .ssh directory or the authorized\_keys file does not exist for the airbyte user. Here’s how you can fix this step by step:

### **Step 1: Create the .ssh Directory for the User**

**Create the .ssh directory**:  
bash  
CopyEdit  
sudo mkdir -p /home/airbyte/.ssh

1. **Explanation**:
   * -p: Ensures the directory is created, including any necessary parent directories.

**Set the correct permissions for the .ssh directory**:  
bash  
CopyEdit  
sudo chmod 700 /home/airbyte/.ssh

1. **Why?**
   * chmod 700: Grants full access to the airbyte user but prevents access by others.

**Change ownership of the directory**:  
bash  
CopyEdit  
sudo chown -R airbyte:airbyte /home/airbyte/.ssh

1. **Why?**
   * Ensures the airbyte user owns the .ssh directory and can access it.

### **Step 2: Create the authorized\_keys File**

**Add the public key to the authorized\_keys file**:  
bash  
CopyEdit  
echo "ssh-rsa AAAAB...developer-key" | sudo tee -a /home/airbyte/.ssh/authorized\_keys

1. **What This Does**:
   * Appends the developer’s public SSH key to the authorized\_keys file for the airbyte user.

**Set the correct permissions for the file**:  
bash  
CopyEdit  
sudo chmod 600 /home/airbyte/.ssh/authorized\_keys

1. **Why?**
   * chmod 600: Grants read and write access only to the airbyte user, ensuring security.

**Change ownership of the file**:  
bash  
CopyEdit  
sudo chown airbyte:airbyte /home/airbyte/.ssh/authorized\_keys

1. **Why?**
   * Ensures the file is owned by the airbyte user.

### **Step 3: Verify the Setup**

Check the .ssh directory and file permissions:  
bash  
CopyEdit  
ls -ld /home/airbyte/.ssh

ls -l /home/airbyte/.ssh/authorized\_keys

1. You should see:
   * .ssh directory with drwx------ permissions.
   * authorized\_keys file with -rw------- permissions.

Test SSH access to the airbyte user:  
bash  
CopyEdit  
ssh airbyte@<vm-ip>

1. If the setup is correct, you should log in without being prompted for a password (assuming the private key is properly configured locally).

### **Troubleshooting**

* **If the directory still doesn’t exist**:

Ensure the airbyte user was created:  
bash  
CopyEdit  
sudo adduser airbyte

* + Then repeat the steps above.
* **If permissions are incorrect**:
  + Reapply the commands for chmod and chown carefully, as improper permissions can block SSH access.

Let me know if you encounter further issues! 😊

4. Deploying and Configuring Airflow

### **4. Deploying and Configuring Airflow**

In this step, we’ll install and configure **Apache Airflow** on the GCP VM under the airflow user. This includes creating its environment, configuring necessary services, and testing the installation.

### **Step 1: Switch to the Airflow User**

1. **Log in to the VM as the airflow user**:

If you’ve set up SSH keys, you can directly log in:  
bash  
CopyEdit  
ssh airflow@<vm-ip>  
  
In our case: ssh airflow-vm

**Switch from your current user to the airflow user**:  
bash  
CopyEdit  
sudo su - airflow

1. **Why?**
   * This ensures that all Airflow-specific files, configurations, and environments are managed by the airflow user.

### **Step 2: Set Up Airflow Environment**

**Install Python and Virtual Environment Tools**:  
bash  
CopyEdit  
sudo apt update && sudo apt install -y python3 python3-pip python3-venv

1. **Why?**
   * Airflow requires Python 3 and pip to run. A virtual environment isolates the Airflow installation from the system environment.

**Create and Activate a Virtual Environment**:  
bash  
CopyEdit  
python3 -m venv airflow-venv

source airflow-venv/bin/activate

1. **Why?**
   * Using a virtual environment prevents dependency conflicts with other Python applications on the system.

**Upgrade pip and Install Airflow**:  
bash  
CopyEdit  
pip install --upgrade pip

pip install apache-airflow

1. **Why?**
   * Installing Airflow in the virtual environment ensures that all required dependencies are installed in an isolated manner.

### **Step 3: Configure Airflow**

1. **Set the Airflow Home Directory**:

Define AIRFLOW\_HOME to specify where Airflow files and logs will be stored:  
bash  
CopyEdit  
export AIRFLOW\_HOME=~/airflow

mkdir -p $AIRFLOW\_HOME

Add this to your shell configuration (~/.bashrc) for persistence:  
bash  
CopyEdit  
echo "export AIRFLOW\_HOME=~/airflow" >> ~/.bashrc

source ~/.bashrc

1. **Why?**
   * This centralizes all Airflow-related files in a dedicated directory for better organization and easier troubleshooting.

**Initialize the Airflow Database**:  
bash  
CopyEdit  
airflow db init

1. **What It Does**:
   * Creates the metadata database where Airflow stores information about DAGs, tasks, and their states.

**Create an Admin User**:  
bash  
CopyEdit  
airflow users create \

--username admin \

--firstname Admin \

--lastname User \

--role Admin \

--email admin@example.com

1. **Why?**
   * This user will be used to log in to the Airflow web interface.

### **Step 4: Start Airflow Services**

**Start the Webserver**:  
bash  
CopyEdit  
airflow webserver -p 8080

1. **What It Does**:
   * Starts the Airflow web interface, accessible at http://<vm-ip>:8080.
2. **Start the Scheduler**:

Open a new terminal window (or use tmux/screen to run both processes in parallel):  
bash  
CopyEdit  
airflow scheduler

1. **Why?**
   * The scheduler is responsible for running tasks according to the DAG schedule.

### **Step 5: Configure Airflow for Airbyte Integration**

1. **Install Additional Airflow Providers**:

For API-based integration:  
bash  
CopyEdit  
pip install apache-airflow-providers-http

1. **Why?**
   * The http provider allows you to make API calls from Airflow, which will be used to trigger Airbyte jobs.
2. **Create a DAG for Airbyte**:

Navigate to the dags/ directory:  
bash  
CopyEdit  
mkdir -p $AIRFLOW\_HOME/dags

Create a new DAG file (e.g., airbyte\_integration.py):  
bash  
CopyEdit  
nano $AIRFLOW\_HOME/dags/airbyte\_integration.py

Example DAG:  
python  
CopyEdit  
from airflow import DAG

from airflow.providers.http.operators.http import SimpleHttpOperator

from datetime import datetime

with DAG(

"airbyte\_integration",

default\_args={"retries": 1},

schedule\_interval=None,

start\_date=datetime(2023, 1, 1),

catchup=False,

) as dag:

trigger\_airbyte = SimpleHttpOperator(

task\_id="trigger\_airbyte\_sync",

http\_conn\_id="airbyte\_api",

endpoint="api/v1/jobs/sync",

method="POST",

headers={"Content-Type": "application/json"},

data='{"connectionId": "your-airbyte-connection-id"}',

)

1. **Why?**
   * This DAG triggers Airbyte sync jobs via its API and integrates them into your larger workflows.
2. **Configure Airflow HTTP Connection**:
   * In the Airflow web UI (http://<vm-ip>:8080), navigate to **Admin → Connections** and add an HTTP connection:
     + **Conn ID**: airbyte\_api
     + **Host**: http://localhost:8000 (Airbyte’s API endpoint)
     + **Extra**: {"Authorization": "Bearer your-auth-token"} (if authentication is required)
3. **Why?**
   * This sets up the connection Airflow uses to communicate with Airbyte.

### **Step 6: Test the Setup**

1. **Access the Airflow Web Interface**:
   * Open http://<vm-ip>:8080 in your browser and log in using the admin credentials.
2. **Trigger the Airbyte DAG**:
   * In the Airflow web interface, locate the airbyte\_integration DAG and trigger it manually.
3. **Verify Logs**:
   * Check the Airflow task logs to ensure the API call was successful and the Airbyte job was triggered.

### **Best Practices for Airflow Setup**

1. **Resource Allocation**:
   * Monitor resource usage on the VM (CPU, memory) and adjust the instance size if necessary.
2. **Logging**:
   * Ensure Airflow logs are stored in a persistent location ($AIRFLOW\_HOME/logs) for troubleshooting.
3. **Security**:
   * Use Airflow’s built-in RBAC system to control access to the web interface.
   * Never expose Airflow’s API or webserver to public networks without proper authentication.

The next step will cover **deploying and configuring Airbyte** on the VM, followed by integration between the two services. Let me know if you’re ready to continue! 😊

5. Production-Grade Deployment of Apache Airflow

### **Production-Grade Deployment of Apache Airflow**

A production-grade Airflow deployment requires considerations for **scalability**, **security**, and **reliability**. Below is a comprehensive guide to deploying Airflow in a production environment on a GCP VM, focusing on best practices and robust configurations.

### **1. Architectural Overview**

For a production setup, Airflow should:

1. Run on a **dedicated GCP VM** with sufficient resources.
2. Use a **PostgreSQL database** as the metadata database (instead of SQLite, which is for testing only).
3. Leverage a **CeleryExecutor** for distributed task execution.
4. Store logs in **Google Cloud Storage** (GCS) or another external storage for persistence.
5. Expose the **webserver** securely with proper authentication (e.g., using RBAC).

### **2. Prerequisites**

#### **VM Specifications:**

* **Machine type**: e2-standard-4 (4 vCPUs, 16 GB RAM) or higher.
* **Disk**: At least 50GB to handle logs and temporary data.
* **OS**: Debian 11 or Ubuntu 20.04 LTS.

#### **Software and Tools:**

* Python 3.7 or higher.
* PostgreSQL database (managed or self-hosted on another GCP VM).
* Docker and Docker Compose (optional, but recommended for easier deployment).

### **3. Deployment Steps**

### **Step 1: Create the GCP VM**

1. **Go to GCP Console** → **Compute Engine** → **VM Instances**.
2. **Configure the VM**:
   * Name: airflow-production
   * Machine Type: e2-standard-4 (4 vCPUs, 16 GB RAM)
   * Boot Disk: Ubuntu 20.04 LTS, 50GB
   * Firewall: Allow HTTP and HTTPS traffic.

**SSH into the VM**:  
bash  
CopyEdit  
gcloud compute ssh airflow-production --zone=us-central1-a

### **Step 2: Install System Dependencies**

Update the system and install required packages:

bash

CopyEdit

sudo apt update && sudo apt upgrade -y

sudo apt install -y python3 python3-pip python3-venv libpq-dev postgresql-client rabbitmq-server nginx

### **Step 3: Set Up the Metadata Database (PostgreSQL)**

Use **Google Cloud SQL** for the metadata database:  
bash  
CopyEdit  
gcloud sql instances create airflow-db --tier=db-f1-micro --region=us-central1

=> ( gcloud sql instances create airflow-db --tier=db-f1-micro --region=northamerica-northeast2 )

WEll this created MysSQL

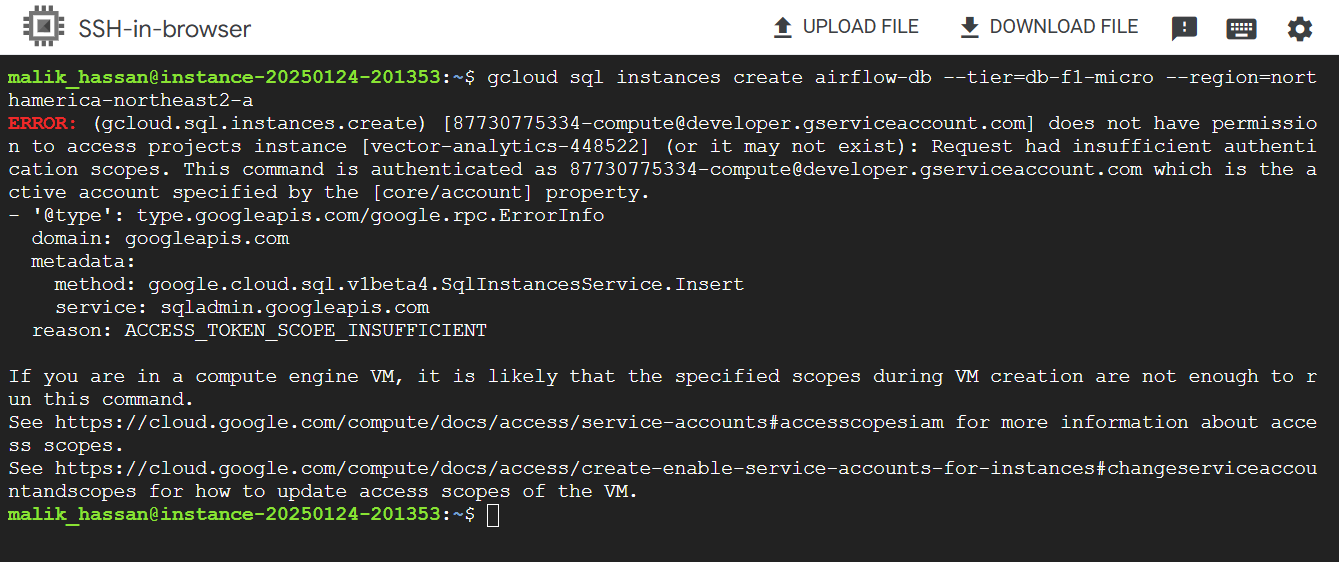
Use this for Postgrres

gcloud sql instances create airflow-db --database-version=POSTGRES\_14 --tier=db-f1-micro --region=northamerica-northeast2

gcloud sql databases create airflow\_metadata --instance=airflow-db

gcloud sql users set-password postgres --host=% --instance=airflow-db --password=StrongPass@123

If not able to create GCP database then visit this:   
<https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.yltkqg1l4ioh>



—

Retrieve the connection string for PostgreSQL:  
plaintext  
CopyEdit  
postgresql+psycopg2://postgres:StrongPass@123/<CLOUD\_SQL\_IP>:5432/airflow\_metadata

The error occurs because you are trying to execute a **connection string** as if it were a shell command. This is not how connection strings are used. A **connection string** like postgresql+psycopg2://... is meant to be used within an application or configuration file, not directly in the terminal.

Test connectivity:  
bash  
CopyEdit  
psql -h <CLOUD\_SQL\_IP> -U postgres -d airflow\_metadata

psql -h 34.130.5.9 -U postgres -d airflow\_metadata

### **Step 4: Install Airflow**

Create a dedicated airflow user:  
bash  
CopyEdit  
sudo adduser airflow

sudo su - airflow

Set up a Python virtual environment:  
bash  
CopyEdit  
python3 -m venv airflow-venv

source airflow-venv/bin/activate

Install Airflow and dependencies:  
bash

[Release Notes — Airflow Documentation](https://airflow.apache.org/docs/apache-airflow/stable/release_notes.html)  
Airflow release version

CopyEdit  
export AIRFLOW\_VERSION=2.10.4

pip install "apache-airflow==${AIRFLOW\_VERSION}" "apache-airflow[celery,postgres]==${AIRFLOW\_VERSION}"

### **Step 5: Configure Airflow**

Define the Airflow home directory:  
bash  
CopyEdit  
export AIRFLOW\_HOME=~/airflow

mkdir -p $AIRFLOW\_HOME

echo "export AIRFLOW\_HOME=~/airflow" >> ~/.bashrc

source ~/.bashrc

<https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.hufmvurkbp75>

Initialize the database:  
bash  
CopyEdit  
airflow db init

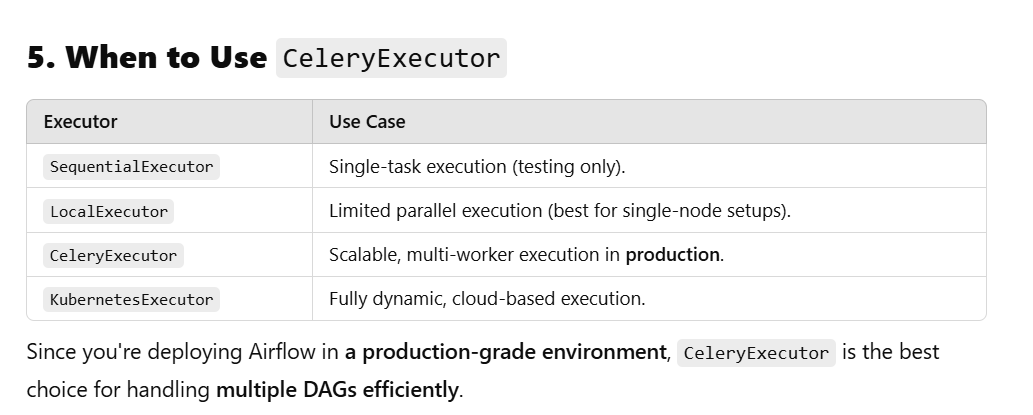
How to init: <https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.t9bfvrokkpgm>  
  
Options configure DAtabase connection with VM:

1: Through External Ip and port adding in the network firewall: https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.tj3fzjysinpk

2: Through proxy tunnel: <https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.59xg0td231ib>  
  
I think option 2 is a lot better than option one. As the database would not need to access through the internet! Just by the vm for airflow only. If the tunnel works then why not.

Update airflow.cfg:  
bash  
CopyEdit  
nano $AIRFLOW\_HOME/airflow.cfg

Set **executor** to CeleryExecutor:  
ini  
CopyEdit  
executor = CeleryExecutor



Set the **database connection string**:  
ini  
CopyEdit  
sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass@123@<CLOUD\_SQL\_IP>:5432/airflow\_metadata

Configure remote logging:  
ini  
CopyEdit  
remote\_logging = True

remote\_log\_conn\_id = gcs\_default

remote\_base\_log\_folder = gs://airflow-logs

### **Step 6: Set Up Celery**

https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.p3xs894k4exh

Configure RabbitMQ as the broker:  
bash  
CopyEdit  
sudo systemctl enable rabbitmq-server

sudo systemctl start rabbitmq-server

sudo systemctl status rabbitmq-server

Update airflow.cfg:  
ini  
CopyEdit  
broker\_url = pyamqp://guest:guest@localhost:5672/

result\_backend = db+postgresql://postgres:StrongPass@123@<CLOUD\_SQL\_IP>:5432/airflow\_metadata

<https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.87prbt5dvzk0>  
**result\_backend vs sql\_alchemy\_conn in Airflow**

### **Step 7: Secure Airflow**

**Create an admin user**:  
bash  
CopyEdit

airflow users create \

--username admin \

--firstname Admin \

--lastname User \

--role Admin \

--email [admin@example.com](mailto:admin@example.com)

Password: admin

1. **Secure the Webserver with HTTPS**:
   * Install SSL certificates using Let’s Encrypt or self-signed certs.

Set up NGINX to proxy the Airflow webserver securely:  
bash  
CopyEdit  
sudo nano /etc/nginx/sites-available/airflow

Add:  
nginx  
CopyEdit  
server {

listen 443 ssl;

server\_name airflow.yourdomain.com;

ssl\_certificate /etc/letsencrypt/live/airflow.yourdomain.com/fullchain.pem;

ssl\_certificate\_key /etc/letsencrypt/live/airflow.yourdomain.com/privkey.pem;

location / {

proxy\_pass http://localhost:8080;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

}

}

Restart NGINX:  
bash  
CopyEdit  
sudo ln -s /etc/nginx/sites-available/airflow /etc/nginx/sites-enabled/

sudo systemctl restart nginx

### **Step 8: Start Airflow Services**

Start the Airflow webserver:  
bash  
CopyEdit  
airflow webserver -p 8080

Start the scheduler:  
bash  
CopyEdit  
airflow scheduler

Start the Celery worker:  
bash  
CopyEdit  
airflow celery worker

### **Step 9: Test and Validate**

1. Access the Airflow web UI:
   * Open https://airflow.yourdomain.com in a browser.
2. Verify that tasks are executed using the Celery worker.

### **Step 10: Monitor and Scale**

Enable **Google Cloud Monitoring** to track resources:  
bash  
CopyEdit  
gcloud monitoring dashboards create

1. Add more workers for scaling:
   * Create additional VMs for workers.
   * Connect them to RabbitMQ and the metadata database.

### **Best Practices**

* **Database**: Use a managed Cloud SQL instance for high availability.
* **Logging**: Store logs in GCS for durability.
* **Security**: Protect web server access using RBAC and HTTPS.
* **Backup**: Regularly back up the metadata database.

This setup ensures Airflow is secure, scalable, and production-ready. Next, we can move to **deploying Airbyte** or address any questions about Airflow. Let me know! 😊

Explanation of Airflow Environment Setup Commands

### **Explanation of Airflow Environment Setup Commands**

These commands are setting up the **Airflow home directory**, which is the central location where Apache Airflow stores configuration files, logs, DAGs, and other metadata.

## **1. Set Airflow Home Environment Variable**

bash

CopyEdit

export AIRFLOW\_HOME=~/airflow

### **What This Does**

* **export AIRFLOW\_HOME=~/airflow** sets the AIRFLOW\_HOME environment variable to ~/airflow (the airflow directory inside the user's home folder).
* This means that Airflow will store its configuration files, logs, and DAGs inside ~/airflow.

### **Why Is This Needed?**

* By default, Airflow looks for configurations in ~/airflow.
* Explicitly setting this variable ensures that all Airflow commands use this directory as the base.

## **2. Create the Airflow Home Directory**

bash

CopyEdit

mkdir -p $AIRFLOW\_HOME

### **What This Does**

* **mkdir -p** ensures that the directory exists.
* The -p flag prevents errors if the directory already exists.

$AIRFLOW\_HOME expands to ~/airflow, so this command is the same as:  
bash  
CopyEdit  
mkdir -p ~/airflow

### **Why Is This Needed?**

* Airflow requires a directory to store its configurations, logs, and DAGs.
* If the directory does not exist, Airflow commands may fail.

## **3. Persist the Airflow Home Path**

bash

CopyEdit

echo "export AIRFLOW\_HOME=~/airflow" >> ~/.bashrc

### **What This Does**

* **echo "export AIRFLOW\_HOME=~/airflow" >> ~/.bashrc"** appends the AIRFLOW\_HOME variable to ~/.bashrc, which is a configuration file that runs every time a new terminal session starts.
* This ensures that the AIRFLOW\_HOME variable is **set automatically** every time you log in or open a new terminal session.

### **Why Is This Needed?**

* If you only use export AIRFLOW\_HOME=~/airflow in the terminal, the setting is **temporary** and will be lost when you close the terminal.
* Adding it to ~/.bashrc makes it **persistent**, so every new session automatically loads the AIRFLOW\_HOME variable.

## **4. Apply the Changes**

bash

CopyEdit

source ~/.bashrc

### **What This Does**

* **source ~/.bashrc** reloads the ~/.bashrc file so that the changes take effect **immediately** without requiring a logout or terminal restart.

### **Why Is This Needed?**

* After updating ~/.bashrc, you need to reload it for the changes to take effect.
* Running source ~/.bashrc ensures the AIRFLOW\_HOME variable is available in the current session.

## **Summary of What These Commands Do**

| **Command** | **Purpose** |
| --- | --- |
| export AIRFLOW\_HOME=~/airflow | Sets Airflow's home directory (temporary). |
| mkdir -p $AIRFLOW\_HOME | Ensures the Airflow directory exists. |
| echo "export AIRFLOW\_HOME=~/airflow" >> ~/.bashrc | Makes the Airflow home directory setting permanent. |
| source ~/.bashrc | Applies the changes immediately without restarting the terminal. |

### **Next Steps After Setting Up AIRFLOW\_HOME**

**Initialize the Airflow Database**:  
bash  
CopyEdit  
airflow db init

* + This sets up the Airflow metadata database inside $AIRFLOW\_HOME.

**Start Airflow Webserver**:  
bash  
CopyEdit  
airflow webserver -p 8080

* + Runs the Airflow web interface.

**Start the Scheduler**:  
bash  
CopyEdit  
airflow scheduler

* + Manages DAG execution.

This setup ensures that Airflow runs smoothly in a **dedicated directory** and is configured persistently across terminal sessions. 🚀 Let me know if you need further clarification! 😊

O

ChatGPT can make mistake

Connect to CLoud SQL Instance

### **How Does airflow db init Connect with the Cloud SQL Instance?**

When you run:

bash

CopyEdit

airflow db init

it initializes the **metadata database** for Apache Airflow. The way it connects to **Cloud SQL (PostgreSQL or MySQL)** depends on the **sql\_alchemy\_conn** setting inside airflow.cfg.

## **1. How airflow db init Works**

* airflow db init **creates the necessary tables and schema** in the metadata database.
* It reads the database connection string from the **Airflow configuration file** (airflow.cfg).
* The connection string is set in the **sql\_alchemy\_conn** parameter.

## **2. Configuring Airflow to Connect to Cloud SQL**

For Airflow to use **Cloud SQL** as the backend metadata database, follow these steps:

### **Step 1: Update sql\_alchemy\_conn in airflow.cfg**

**Edit the Airflow configuration file**:  
bash  
CopyEdit  
nano ~/airflow/airflow.cfg  
  
Nano is to eidt, I can use VScode editing the file for myself.

**Find and update the sql\_alchemy\_conn parameter**:  
ini  
CopyEdit  
sql\_alchemy\_conn = postgresql+psycopg2://postgres:YourPassword@34.130.5.9:5432/airflow\_metadata

sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass@123@34.130.5.9:5432/airflow\_metadata  
  
Error

sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass%%40123@34.130.5.9:5432/airflow\_metadata

%%40 as encoded version of @ symbol

* + Replace:
    - **YourPassword**: The password for your Cloud SQL postgres user.
    - **34.130.5.9**: The **public IP** of your Cloud SQL instance.
    - **5432**: The default PostgreSQL port.
    - **airflow\_metadata**: The Cloud SQL database name.

1. **Save and exit (Ctrl+O, Enter, Ctrl+X).**

**bytheway : I can the default here: sql\_alchemy\_conn = sqlite:////home/airflow/airflow/airflow.db**

Which is SQL light

### **Step 2: Test Connectivity**

Before running airflow db init, test if Airflow can connect to Cloud SQL:

bash

CopyEdit

psql -h 34.130.5.9 -U postgres -d airflow\_metadata

* If you can log in successfully, your connection settings are correct.

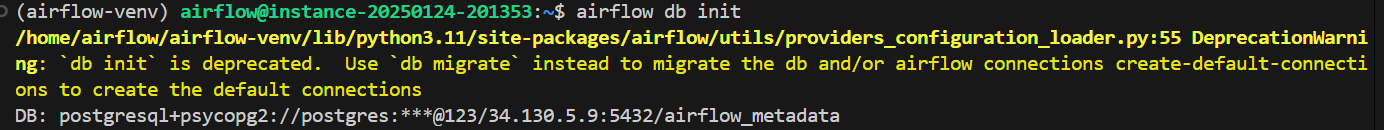
### **Step 3: Run Airflow DB Initialization**

Now, run:

bash

CopyEdit

airflow db init



airflow db migrate

( for airflow 2.10.4 that i am currently running)

* Airflow will **connect to Cloud SQL** using the sql\_alchemy\_conn string.
* It will **create tables** required for DAG execution, logging, and scheduling.

## 

## 

Not Tried Yet: But I think this would be better for Airflow database as it woud only be used by air flow it self.

## 

## **3. Alternative: Using Cloud SQL Proxy for Secure Connection**

If you **don’t want to expose Cloud SQL via a public IP**, use the **Cloud SQL Proxy**.

### **Step 1: Install Cloud SQL Proxy**

bash

CopyEdit

wget https://dl.google.com/cloudsql/cloud\_sql\_proxy.linux.amd64 -O cloud\_sql\_proxy

chmod +x cloud\_sql\_proxy

sudo mv cloud\_sql\_proxy /usr/local/bin/



This does not work: Because my user airflow does not have root access and /usr/local/bin/ is root level permission.

Instead, we will setup a folder for the service user.

[VM Deployment](https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.crwvkgv0g04z)

### **Step 2: Start the Proxy**

bash

CopyEdit

cloud\_sql\_proxy -instances=<PROJECT\_ID>:<REGION>:airflow-db=tcp:5432

cloud\_sql\_proxy -instances=vector-analytics-448522:northamerica-northeast2:airflow-db=tcp:5432

|  |
| --- |

Replace:

* **<PROJECT\_ID>**: Your GCP project ID. vector-analytics-448522
* **<REGION>**: The region where your Cloud SQL instance is deployed. northamerica-northeast2-a
* **airflow-db**: The Cloud SQL instance name.

### **Step 3: Update sql\_alchemy\_conn**

If using Cloud SQL Proxy, modify airflow.cfg:

ini

CopyEdit

sql\_alchemy\_conn = postgresql+psycopg2://postgres:YourPassword@127.0.0.1:5432/airflow\_metadata

### **Step 4: Run airflow db init**

bash

CopyEdit

airflow db init

* Now, Airflow will **connect securely** through the proxy instead of using a public IP.

## **4. Verify That Airflow Is Using Cloud SQL**

After initializing the database, check the tables in Cloud SQL:

bash

CopyEdit

psql -h 34.130.5.9 -U postgres -d airflow\_metadata -c "\dt"

If Airflow is correctly connected, you'll see tables like:

plaintext

CopyEdit

List of relations

Schema | Name | Type | Owner

--------+-------------------+-------+--------

public | alembic\_version | table | postgres

public | dag | table | postgres

public | log | table | postgres

Option 1: How Does airflow cont with the Cloud SQL

### **How Does airflow db init Connect with the Cloud SQL Instance?**

When you run:

bash

CopyEdit

airflow db init

it initializes the **metadata database** for Apache Airflow. The way it connects to **Cloud SQL (PostgreSQL or MySQL)** depends on the **sql\_alchemy\_conn** setting inside airflow.cfg.

## **1. How airflow db init Works**

* airflow db init **creates the necessary tables and schema** in the metadata database.
* It reads the database connection string from the **Airflow configuration file** (airflow.cfg).
* The connection string is set in the **sql\_alchemy\_conn** parameter.

## **2. Configuring Airflow to Connect to Cloud SQL**

For Airflow to use **Cloud SQL** as the backend metadata database, follow these steps:

### **Step 1: Update sql\_alchemy\_conn in airflow.cfg**

**Edit the Airflow configuration file**:  
bash  
CopyEdit  
nano ~/airflow/airflow.cfg  
  
Nano is to eidt, I can use VScode editing the file for myself.

**Find and update the sql\_alchemy\_conn parameter**:  
ini  
CopyEdit  
sql\_alchemy\_conn = postgresql+psycopg2://postgres:YourPassword@34.130.5.9:5432/airflow\_metadata

sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass@123@34.130.5.9:5432/airflow\_metadata  
  
Error

sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass%%40123@34.130.5.9:5432/airflow\_metadata

%%40 as encoded version of @ symbol

* + Replace:
    - **YourPassword**: The password for your Cloud SQL postgres user.
    - **34.130.5.9**: The **public IP** of your Cloud SQL instance.
    - **5432**: The default PostgreSQL port.
    - **airflow\_metadata**: The Cloud SQL database name.

1. **Save and exit (Ctrl+O, Enter, Ctrl+X).**

**bytheway : I can the default here: sql\_alchemy\_conn = sqlite:////home/airflow/airflow/airflow.db**

Which is SQL light

### **Step 2: Test Connectivity**

Before running airflow db init, test if Airflow can connect to Cloud SQL:

bash

CopyEdit

psql -h 34.130.5.9 -U postgres -d airflow\_metadata

* If you can log in successfully, your connection settings are correct.

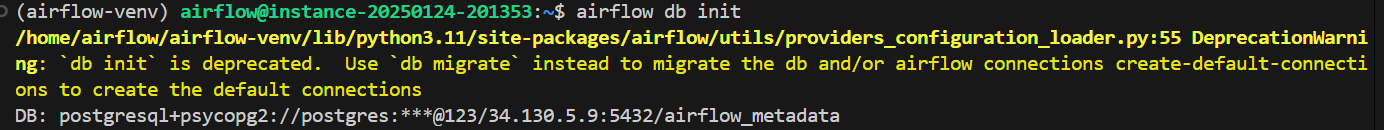
### **Step 3: Run Airflow DB Initialization**

Now, run:

bash

CopyEdit

airflow db init



airflow db migrate

( for airflow 2.10.4 that i am currently running)

* Airflow will **connect to Cloud SQL** using the sql\_alchemy\_conn string.
* It will **create tables** required for DAG execution, logging, and scheduling.

## 

DAtabase Test taking long

psql -h 34.130.5.9 -U postgres -d airflow\_metadata is taking long time!

###### **ChatGPT said:**

ChatGPT

If the psql command is taking too long to connect, it's likely due to **network configuration issues** or **firewall rules** blocking the connection to your Cloud SQL instance. Here's a step-by-step troubleshooting guide:

### **1. Check Cloud SQL Instance Accessibility**

#### **Step 1.1: Verify Cloud SQL Instance is Running**

* Go to the Google Cloud SQL Console.
* Ensure the instance (airflow-db) is in the **RUNNING** state.

#### **Step 1.2: Check the Public IP Address**

* On the instance details page, confirm the **Public IP Address** matches the one you're using in the psql command (34.130.5.9).

### **2. Ensure Authorized Networks are Configured**

Cloud SQL requires you to explicitly allow external connections.

#### **Step 2.1: Add the IP Address of the Connecting Machine**

1. Go to the **Cloud SQL Instance** details page.
2. Navigate to the **Connections** tab.
3. Under **Authorized Networks**, add the external IP address of the machine where you're running the psql command (e.g., your local machine or GCP VM):
   * You can find your machine’s external IP at<https://whatismyipaddress.com/>.
4. Save the changes.

#### **Step 2.2: Restart the SQL Instance (if needed)**

Some configuration changes may require a restart:

* Go to the **Instance Details** page → Click **Restart**.

### **3. Test Connectivity from the GCP VM**

**Ensure the VM Can Access the Cloud SQL Public IP** From your VM, run:  
bash  
CopyEdit  
ping 34.130.5.9

* + If the ping succeeds, the VM can reach the SQL instance.

**Verify Port 5432 is Open** Use telnet or nc to test connectivity to port 5432:  
bash  
CopyEdit  
nc -zv 34.130.5.9 5432

ORRR

telnet 34.130.5.9 5432

* + A successful response confirms the port is open.

**Retry the psql Command**:  
bash  
CopyEdit  
psql -h 34.130.5.9 -U postgres -d airflow\_metadata

NOT Installedn telent or NC then ===>

Install Netcat (nc):  
bash  
CopyEdit  
sudo apt update

sudo apt install netcat -y

Alternatively, install Telnet:  
bash  
CopyEdit  
sudo apt install telnet -y

### **4. If Using a Private IP Address**

If your Cloud SQL instance is configured with **Private IP**:

1. Ensure the GCP VM and the Cloud SQL instance are in the **same VPC network**.
2. Use the **Private IP Address** of the Cloud SQL instance instead of the Public IP.

To find the private IP:

* Go to the **Instance Details** page → Look for **Private IP Address** under the **Connections** tab.

### **5. Common Issues and Fixes**

#### **Issue 1: Connection Timeout**

* **Cause**: The VM or local machine can't reach the Cloud SQL instance.
* **Fix**: Ensure the public IP of the machine is added to the authorized networks.

#### **Issue 2: Firewall Rules**

* **Cause**: Outbound traffic on port 5432 is blocked.
* **Fix**: Ensure no firewall rules block traffic on port 5432.
* https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.76hv4sk0l33l

#### **Issue 3: PostgreSQL Client Version**

* **Cause**: An incompatible PostgreSQL client version.
* **Fix**: Ensure you're using a compatible client version (e.g., PostgreSQL 13+ for Cloud SQL PostgreSQL 13).

#### **Issue 4: DNS Issues**

* **Cause**: DNS resolution may fail in some environments.

**Fix**: Use the psql command with --set=sslmode=disable temporarily to test:  
bash  
CopyEdit  
psql -h 34.130.5.9 -U postgres -d airflow\_metadata --set=sslmode=disable

### **6. Debugging with Logs**

If the issue persists, check the **Cloud SQL logs**:

1. Go to the **Cloud SQL Instance Details** page.
2. Navigate to the **Logs** tab.
3. Look for connection errors or failed authentication logs.

If you’ve tried the above and it’s still slow, let me know the output of your telnet or nc command, and I’ll help troubleshoot further! 😊

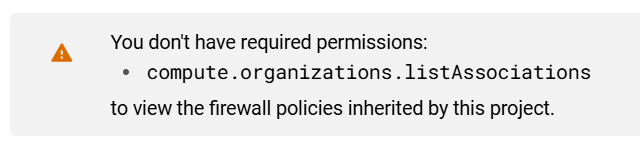
Configure IP - PostgresSQL

here's how to check and configure **firewall rules** in Google Cloud Platform (GCP) to ensure that port 5432 (used by PostgreSQL) is open for your Cloud SQL instance:

### **Step 1: Check Firewall Rules in GCP**

#### **1.1: Navigate to Firewall Rules**

1. Open the Google Cloud Console Firewall Rules Page.
2. Review the list of existing firewall rules in your project.



#### **1.2: Identify Relevant Rules**

* Look for a rule that:
  + Applies to the VPC where your Cloud SQL instance is hosted.
  + Allows ingress traffic on port 5432 for PostgreSQL.
* If no such rule exists, you'll need to create one (details in Step 2).

### **Step 2: Create a Firewall Rule for Port 5432**

1. **Go to the Firewall Rules Page**:
   * Navigate to **VPC Network → Firewall Rules** in the GCP Console.
2. **Click on "Create Firewall Rule"**:
   * Set the following details:
     + **Name**: allow-postgresql-ingress
     + **Network**: Choose the VPC where your Cloud SQL instance is deployed (default is default).
     + **Priority**: Leave as default (1000).
     + **Direction of Traffic**: **Ingress** (traffic coming into the instance).
     + **Action**: **Allow**.
     + **Targets**: Select **Specified target tags** (use a tag like cloud-sql).
     + **Source Filter**: Choose one of the following based on your setup:
       - **IP ranges**: Add the IP address of the machine or VM connecting to Cloud SQL (e.g., 203.0.113.45/32).
       - For testing purposes (not recommended for production), use 0.0.0.0/0 to allow all sources.
     + **Protocols and Ports**: Select **Specified protocols and ports**, then enable **TCP** and enter 5432.
3. **Save the Rule**:
   * Click **Create** to apply the rule.

### **Step 3: Attach a Firewall Tag to the Cloud SQL Instance**

1. **Go to the Cloud SQL Instance Page**:
   * Navigate to **SQL → Instances** in the GCP Console.
   * Select your Cloud SQL instance.
2. **Attach the Firewall Tag**:
   * In the **Instance Details** section, add the target tag (e.g., cloud-sql) to the instance to ensure the firewall rule applies.

### **Step 4: Check Outbound Rules on the VM or Local Machine**

To connect to the Cloud SQL instance, your **local machine** or **GCP VM** must allow outbound traffic on port 5432.

#### **4.1: Check GCP VM Firewall Rules**

* Go to **VPC Network → Firewall Rules** in the GCP Console.
* Look for a rule allowing **egress** (outbound) traffic:
  + **Destination**: 0.0.0.0/0.
  + **Ports/Protocols**: TCP 5432.

If no such rule exists, create one:

1. **Create a New Firewall Rule**:
   * **Name**: allow-postgresql-egress.
   * **Network**: Select your VM's VPC.
   * **Priority**: 1000 (default).
   * **Direction**: **Egress**.
   * **Action**: **Allow**.
   * **Targets**: Leave as **All instances in the network**.
   * **Destination Filter**: Set to **IP ranges**, and add 0.0.0.0/0.
   * **Protocols and Ports**: Enable **TCP** and set 5432.
2. **Save the Rule**:
   * Click **Create** to apply the changes.

### **Step 5: Verify Firewall Rules**

**List All Firewall Rules (Optional)**: Use the gcloud CLI to check all rules:  
bash  
CopyEdit  
gcloud compute firewall-rules list

**Verify Specific Rules**: To verify the specific rule for port 5432:  
bash  
CopyEdit  
gcloud compute firewall-rules describe allow-postgresql-ingress

gcloud compute firewall-rules describe [airflow-vm-n-postgresdb](https://console.cloud.google.com/net-security/firewall-manager/firewall-policies/details/airflow-vm-n-postgresdb?project=vector-analytics-448522&hl=en&invt=AboJzw)

1. The output should show:
   * **Direction**: INGRESS.
   * **Ports**: 5432.
   * **Source IP ranges**: Your machine's external IP or 0.0.0.0/0.

yes

### **Step 6: Retest Connection**

After confirming firewall rules:

Retry the psql connection:  
bash  
CopyEdit  
psql -h 34.130.5.9 -U postgres -d airflow\_metadata

1. If you still experience issues, check:
   * The Cloud SQL **Authorized Networks**.
   * Logs in **Cloud SQL → Logs** for connection errors.

Let me know if you encounter any issues during the process!

Add Your Machine's IP to Authorized Networks

### **Fixing Connection Timeout Issue for Cloud SQL (PostgreSQL) in GCP**

A **connection timeout** when trying to connect to Cloud SQL means your VM or local machine **cannot reach the database**. The most common cause is that your machine's **public IP is not added to the authorized networks**.

## **Step 1: Add Your Machine's IP to Authorized Networks**

By default, Cloud SQL does **not** allow connections from external networks unless explicitly permitted.

### **1.1: Find Your Public IP**

If connecting from your **local machine** (laptop/desktop):

bash

CopyEdit

curl ifconfig.me

This will return your public IP (e.g., 203.0.113.45).

If connecting from a **GCP VM**:

* Go to GCP Compute Engine.
* Locate your VM and note the **External IP**.

### **1.2: Add the IP to Cloud SQL Authorized Networks**

1. **Go to Cloud SQL Console**:
   * Open Google Cloud SQL.
   * Click on your **PostgreSQL instance** (airflow-db).
2. **Navigate to the "Connections" Tab**.
3. **Under "Authorized Networks", Click "Add Network"**:
   * **Name**: my-machine

**Network**: Your public IP (from Step 1.1), in CIDR format:  
CopyEdit  
203.0.113.45/32

### **Why Do We Add /32?**

* **/32 means a single IP address**.
* It tells Cloud SQL to allow only that **one specific IP** (your VM’s external IP) to connect.
* Add it by yourself

If connecting from a GCP VM, use its **External IP** instead.

1. **Click "Save"**.
2. **Wait a Few Minutes** for the changes to apply.

## **Step 2: Verify Cloud SQL Connection Again**

Once the authorized network is set up, try connecting again.

### **2.1: Using psql from a Local Machine**

bash

CopyEdit

psql -h 34.130.5.9 -U postgres -d airflow\_metadata

If prompted, enter your PostgreSQL **password**.

### **2.2: Using gcloud to Test Connection**

bash

CopyEdit

gcloud sql connect airflow-db --user=postgres

## **Step 3: Troubleshooting If Connection Still Fails**

### **Issue: Connection Still Timing Out**

1. **Confirm the Public IP of the Machine**
   * Run curl ifconfig.me again and ensure it's added to the **Authorized Networks**.
2. **Ensure PostgreSQL is Running**
   * Open **Cloud SQL → Instances** in GCP.
   * The status should be RUNNING.
   * Restart the instance if necessary.

**Check If Port 5432 is Reachable** Run:  
bash  
CopyEdit  
nc -zv 34.130.5.9 5432

1. If this fails, Cloud SQL **is not reachable**, and you need to check firewall settings (next step).

## **Step 4: Verify GCP Firewall Rules**

Your VM or local machine **must** allow outbound traffic on **port 5432**.

### **4.1: List Firewall Rules**

Check if a rule exists for PostgreSQL:

bash

CopyEdit

gcloud compute firewall-rules list --filter="name~postgres"

### **4.2: Create a Firewall Rule for PostgreSQL (If Missing)**

If no rule exists, **create one**:

bash

CopyEdit

gcloud compute firewall-rules create allow-postgresql-ingress \

--direction=INGRESS \

--priority=1000 \

--network=default \

--action=ALLOW \

--rules=tcp:5432 \

--source-ranges=203.0.113.45/32

* Replace **203.0.113.45/32** with your **machine's external IP**.

## **Step 5: Use Private IP Instead (If Within GCP)**

If your VM is inside **GCP**, it's best to use a **Private IP** connection instead of a Public IP.

### **5.1: Enable Private IP**

1. Go to **Cloud SQL Console → Connections**.
2. Under **Private IP**, enable it.
3. Note the **Private IP Address**.

### **5.2: Connect Using Private IP**

If your **VM and Cloud SQL are in the same VPC**, use the Private IP:

bash

CopyEdit

psql -h 10.128.0.5 -U postgres -d airflow\_metadata

(Replace 10.128.0.5 with your actual private IP.)

## **Final Steps**

After completing the steps above: ✅ **Retry psql connection**✅ **Ensure IP is authorized**✅ **Verify firewall settings**

If you're still having issues, let me know what error messages you're getting! 🚀

Option 2: SQL Proxy for Secure Connection

## 

## **3. Alternative: Using Cloud SQL Proxy for Secure Connection**

If you **don’t want to expose Cloud SQL via a public IP**, use the **Cloud SQL Proxy**.

### **Step 1: Install Cloud SQL Proxy**

bash

CopyEdit

wget https://dl.google.com/cloudsql/cloud\_sql\_proxy.linux.amd64 -O cloud\_sql\_proxy

chmod +x cloud\_sql\_proxy

sudo mv cloud\_sql\_proxy /usr/local/bin/



This does not work: Because my user airflow does not have root access and /usr/local/bin/ is root level permission.

Instead, we will setup a folder for the service user.

[VM Deployment](https://docs.google.com/document/d/1MVANqgz9mpLMWMVRim0skjXTXlVtTmqKNMoIRv9Qfqs/edit?tab=t.crwvkgv0g04z)

### **Step 2: Start the Proxy**

bash

CopyEdit

cloud\_sql\_proxy -instances=<PROJECT\_ID>:<REGION>:airflow-db=tcp:5432

cloud\_sql\_proxy -instances=instance-20250124-201353:northamerica-northeast2:airflow-db=tcp:5432

|  |
| --- |

Replace:

* **<PROJECT\_ID>**: Your GCP project ID.[Instance-20250124-201353](https://console.cloud.google.com/compute/instancesDetail/zones/northamerica-northeast2-a/instances/instance-20250124-201353?hl=en&inv=1&invt=AboJ6Q&project=vector-analytics-448522)
* **<REGION>**: The region where your Cloud SQL instance is deployed. northamerica-northeast2-a
* **airflow-db**: The Cloud SQL instance name.

### **Step 3: Update sql\_alchemy\_conn**

If using Cloud SQL Proxy, modify airflow.cfg:

ini

CopyEdit

sql\_alchemy\_conn = postgresql+psycopg2://postgres:YourPassword@127.0.0.1:5432/airflow\_metadata

### **Step 4: Run airflow db init**

bash

CopyEdit

airflow db init

* Now, Airflow will **connect securely** through the proxy instead of using a public IP.

## **4. Verify That Airflow Is Using Cloud SQL**

After initializing the database, check the tables in Cloud SQL:

bash

CopyEdit

psql -h 34.130.5.9 -U postgres -d airflow\_metadata -c "\dt"

If Airflow is correctly connected, you'll see tables like:

plaintext

CopyEdit

List of relations

Schema | Name | Type | Owner

--------+-------------------+-------+--------

public | alembic\_version | table | postgres

public | dag | table | postgres

public | log | table | postgres

Fix: Service User Does Not Have Access

### **Fix: Service User Does Not Have Access to /usr/local/bin/**

If the **service user** (e.g., airflow or another non-root user) **does not have access to /usr/local/bin/**, it means the system permissions restrict modifications to this directory. This is **expected behavior** for security reasons.

### **Solution 1: Use the Home Directory Instead**

If /usr/local/bin/ is restricted, you can store cloud\_sql\_proxy in your **home directory** instead:

wget https://dl.google.com/cloudsql/cloud\_sql\_proxy.linux.amd64 -O cloud\_sql\_proxy

chmod +x cloud\_sql\_proxy

#### **1. Move Cloud SQL Proxy to the User's bin Directory**

bash

CopyEdit

mkdir -p ~/bin

mv cloud\_sql\_proxy ~/bin/

chmod +x ~/bin/cloud\_sql\_proxy

#### **2. Add ~/bin to the User's PATH**

bash

CopyEdit

echo 'export PATH=$HOME/bin:$PATH' >> ~/.bashrc

source ~/.bashrc

* This ensures cloud\_sql\_proxy can be run from anywhere without specifying its full path.

#### **3. Verify the Installation**

bash

CopyEdit

cloud\_sql\_proxy --version

### **=> Discarded not needed for now.**

### **Solution 2: Use sudo to Move the Binary to a Shared Directory**

If you want all users to access cloud\_sql\_proxy, move it to /usr/local/bin/, but **change the ownership** so a service user (e.g., airflow) can use it.

Move it with correct permissions:  
bash  
CopyEdit  
sudo mv cloud\_sql\_proxy /usr/local/bin/

sudo chmod +x /usr/local/bin/cloud\_sql\_proxy

sudo chown airflow:airflow /usr/local/bin/cloud\_sql\_proxy

Verify the ownership:  
bash  
CopyEdit  
ls -l /usr/local/bin/cloud\_sql\_proxy

Expected output:  
plaintext  
CopyEdit  
-rwxr-xr-x 1 airflow airflow 12345678 Jan 29 12:34 /usr/local/bin/cloud\_sql\_proxy

### **Solution 3: Run Cloud SQL Proxy Without Moving It**

If you don’t want to modify permissions, just **run it from the directory where it was downloaded**:

bash

CopyEdit

./cloud\_sql\_proxy -instances=instance-20250124-201353:northamerica-northeast2:airflow-db=tcp:5432

(Assuming you're in the directory where cloud\_sql\_proxy was downloaded.)

### **Which Solution Should You Use?**

| **Scenario** | **Solution** |
| --- | --- |
| You want only **your user** to use cloud\_sql\_proxy | Use **Solution 1** (Move to ~/bin) |
| You want **all users** to use cloud\_sql\_proxy | Use **Solution 2** (Move to /usr/local/bin/ & change ownership) |
| You don’t want to move it | Use **Solution 3** (Run it directly from its directory) |

After implementing any of the solutions, **try running Cloud SQL Proxy again**:

bash

CopyEdit

cloud\_sql\_proxy --version

If the issue persists, let me know what error you get! 🚀

Not able to create GCP Database

#### **Validate API Access for Cloud SQL**

Sometimes, the sqladmin.googleapis.com API might not be enabled for your project. Ensure it’s active:

**List All Enabled APIs**:  
bash  
CopyEdit  
gcloud services list --enabled

**Enable the SQL Admin API (if missing)**:  
bash  
CopyEdit  
gcloud services enable sqladmin.googleapis.com

**Retry Creating the Cloud SQL Instance**:  
bash  
CopyEdit  
gcloud sql instances create airflow-db --tier=db-f1-micro --region=northamerica-northeast2

—---------------------------------------

—-----------------------------------------

—--------------------------------------

The error occurs because the **service account** associated with your VM does not have sufficient permissions (authentication scopes) to create a Cloud SQL instance. By default, Compute Engine VMs use a service account with limited scopes, so you'll need to fix this in one of two ways:

### **Solution 1: Update the VM's Service Account and Scopes**

The easiest way to resolve this is to give your VM's service account the correct permissions and scopes for managing Cloud SQL.

#### **Step 1: Verify the VM's Service Account**

Run the following command to confirm the service account associated with your VM:

bash

CopyEdit

gcloud compute instances describe instance-20250124-201353 --zone=us-central1-a --format="get(serviceAccounts.email)"

I used:   
gcloud compute instances describe instance-20250124-201353 --zone=northamerica-northeast2-a --format="get(serviceAccounts.email)"

You'll see something like:

plaintext

CopyEdit

[87730775334-compute@developer.gserviceaccount.com](mailto:87730775334-compute@developer.gserviceaccount.com)

87730775334-compute@developer.gserviceaccount.com

#### **Step 2: Update the VM's Scopes**

The service account needs the **cloud-platform scope** to manage Cloud SQL. Update the VM's access scopes:

**Stop the VM**:  
bash  
CopyEdit  
gcloud compute instances stop instance-20250124-201353 --zone=northamerica-northeast2-a

**Update Scopes**: Add the cloud-platform scope to your VM:  
bash  
CopyEdit  
gcloud compute instances set-service-account instance-20250124-201353 \

--zone=northamerica-northeast2-a \

--scopes=https://www.googleapis.com/auth/cloud-platform

**Start the VM**:  
bash  
CopyEdit  
gcloud compute instances start instance-20250124-201353 --zone=northamerica-northeast2-a

### **Solution 2: Grant IAM Permissions to the Service Account**

Ensure the service account has the necessary IAM role to manage Cloud SQL.

#### **Step 1: Grant Cloud SQL Admin Role**

Run the following command to grant the Cloud SQL Admin role to the service account:  
bash  
CopyEdit  
gcloud projects add-iam-policy-binding vector-analytics-448522 \

--member="serviceAccount:87730775334-compute@developer.gserviceaccount.com" \

--role="roles/cloudsql.admin"

1. **What This Does**:
   * Grants the service account full administrative access to Cloud SQL resources in the project.

Verify the new IAM role:  
bash  
CopyEdit  
gcloud projects get-iam-policy vector-analytics-448522 --flatten="bindings[].members" --filter="bindings.members:87730775334-compute@developer.gserviceaccount.com"

### **Solution 3: Use a Personal Google Cloud Account**

If you’re unable to modify the VM's service account or scopes, you can switch to using a personal Google Cloud account with sufficient permissions.

#### **Step 1: Authenticate as Yourself**

Log in to your Google account:  
bash  
CopyEdit  
gcloud auth login

Set your account as the active account:  
bash  
CopyEdit  
gcloud config set account your-email@example.com

Verify the active account:  
bash  
CopyEdit  
gcloud config get-value account

#### **Step 2: Run the Cloud SQL Command**

Create the Cloud SQL instance:

bash

CopyEdit

gcloud sql instances create airflow-db --tier=db-f1-micro --region=us-central1

### **Important Notes**

1. **VM Restart**: If you update the VM's scopes (Solution 1), ensure you restart the VM for the changes to take effect.
2. **IAM Roles**:
   * The service account needs both the **Cloud SQL Admin** role and the **Compute Admin** role for managing Cloud SQL and interacting with other resources.
3. **Scope Details**: The cloud-platform scope allows the VM to manage all Google Cloud APIs. For security, you can restrict this to only the scopes required for Cloud SQL (https://www.googleapis.com/auth/sqlservice.admin).

### **Verify the Fix**

Once you've made the changes, re-run the command to create the Cloud SQL instance:

bash

CopyEdit

gcloud sql instances create airflow-db --tier=db-f1-micro --region=us-central1

gcloud sql instances create airflow-db --tier=db-f1-micro --region=northamerica-northeast2-a

Let me know if this resolves the issue or if you encounter any additional errors!

What is RabbitMQ and Why Are We Using It?

### **What is RabbitMQ and Why Are We Using It?**

RabbitMQ is a **message broker** that enables **asynchronous communication** between different components of a system. In Apache Airflow, we use RabbitMQ as the **task queue** for **CeleryExecutor**, which allows multiple workers to execute DAG tasks in parallel.

## **1. What is a Message Broker?**

A **message broker** is a system that: ✅ **Receives tasks from one component** (Airflow scheduler).  
✅ **Queues the tasks** until they can be processed.  
✅ **Distributes tasks** to available workers.

### **Why Do We Need RabbitMQ?**

* **Airflow needs to execute tasks in parallel** → RabbitMQ helps distribute them.
* **Celery (task execution engine) requires a broker** → RabbitMQ acts as the broker.
* **Ensures reliable task execution** → Tasks are queued and retried if workers fail.

## **2. Understanding the Commands**

### **Step 1: Enable RabbitMQ on System Boot**

bash

CopyEdit

sudo systemctl enable rabbitmq-server

#### **What This Does**

* systemctl enable ensures that RabbitMQ **automatically starts** when the system reboots.
* This prevents manual restarts after every reboot.
* It creates a **systemd service** that manages RabbitMQ.

#### **Why Are We Doing This?**

* In a production system, you don’t want RabbitMQ to stop if the VM is restarted.
* Ensures **continuous execution of Airflow DAGs**.

### **Step 2: Start RabbitMQ Immediately**

bash

CopyEdit

sudo systemctl start rabbitmq-server

#### **What This Does**

* Starts the **RabbitMQ service right away** (without a reboot).
* If RabbitMQ is already running, this command does nothing.

#### **Why Are We Doing This?**

* Airflow workers need RabbitMQ running before they can receive tasks.
* Without it, **CeleryExecutor will fail** because there’s no broker to queue tasks.

## **3. How RabbitMQ Works in Airflow**

| **Component** | **Function** |
| --- | --- |
| **Airflow Scheduler** | Sends tasks to RabbitMQ. |
| **RabbitMQ (Message Broker)** | Holds tasks in a queue and distributes them to workers. |
| **Celery Workers** | Pick up tasks from RabbitMQ and execute them. |
| **PostgreSQL (Result Backend)** | Stores task execution results for Airflow. |

### **Flow of Task Execution in Airflow**

1️⃣ **Airflow Scheduler** triggers a DAG and sends tasks to **RabbitMQ**.  
2️⃣ **RabbitMQ** holds the tasks in a queue.  
3️⃣ **Celery Workers** check RabbitMQ and execute pending tasks.  
4️⃣ **Results are stored in PostgreSQL** (so Airflow UI can show task status).

## **4. How to Verify That RabbitMQ is Running**

After running the commands, check:

bash

CopyEdit

sudo systemctl status rabbitmq-server

Expected output:

plaintext

CopyEdit

● rabbitmq-server.service - RabbitMQ Broker

Loaded: loaded (/lib/systemd/system/rabbitmq-server.service; enabled; vendor preset: enabled)

Active: active (running) since Tue 2025-01-29 18:00:00 UTC; 2min ago

If it says **"active (running)"**, RabbitMQ is working.

### **Check RabbitMQ Queues**

To check Celery task queues:

bash

CopyEdit

sudo rabbitmqctl list\_queues

If RabbitMQ is correctly configured, you will see:

plaintext

CopyEdit

celery 10

(10 represents the number of queued tasks waiting to be processed.)

## **5. Updating Airflow to Use RabbitMQ**

After starting RabbitMQ, **configure Airflow to use it**:

**Edit airflow.cfg**:  
bash  
CopyEdit  
nano $AIRFLOW\_HOME/airflow.cfg

**Find the [celery] section** and update:  
ini  
CopyEdit  
[celery]

broker\_url = amqp://guest:guest@localhost:5672//

result\_backend = db+postgresql://postgres:YourPassword@127.0.0.1:5432/airflow\_metadata

**Restart Airflow**:  
bash  
CopyEdit  
airflow scheduler &

airflow webserver -p 8080 &

airflow celery worker &

## **6. Why Are We Doing This?**

✅ **Parallel Execution** → Airflow can distribute tasks among multiple workers.  
✅ **Task Reliability** → If a worker fails, RabbitMQ holds the task until another worker picks it up.  
✅ **Scalability** → Can handle hundreds of DAGs efficiently.

### **Without RabbitMQ (SequentialExecutor vs CeleryExecutor)**

| **Executor Type** | **Number of Tasks at a Time** | **Best For** |
| --- | --- | --- |
| **SequentialExecutor** | **1** (No parallel execution) | Local testing |
| **CeleryExecutor + RabbitMQ** | **Multiple** (Parallel execution) | Production |

Since you are setting up a **production-grade Airflow deployment**, **CeleryExecutor with RabbitMQ** is the best choice.

### **Final Summary**

| **Command** | **What It Does** | **Why It’s Needed** |
| --- | --- | --- |
| sudo systemctl enable rabbitmq-server | Ensures RabbitMQ starts on boot | Prevents Airflow failures after reboot |
| sudo systemctl start rabbitmq-server | Starts RabbitMQ immediately | Allows Airflow to send tasks to workers |
| **Update airflow.cfg** | Configures Celery to use RabbitMQ | Enables parallel task execution |

Now, Airflow can **efficiently schedule, queue, and execute tasks** in a distributed system. 🚀

Let me know if you need any more clarification! 😊

O

ChatGPT can make mistakes. Check im

result\_backend vs sql\_alchemy\_conn in Airflow

### **result\_backend vs sql\_alchemy\_conn in Airflow**

Both result\_backend and sql\_alchemy\_conn are used for **database connections in Airflow**, but they serve **different purposes**.

## **1. sql\_alchemy\_conn (Airflow Metadata Database)**

### **Purpose**

* Stores **Airflow metadata** (DAGs, tasks, execution history, logs).
* Tracks the **state** of DAGs and tasks.
* Required for **all Airflow executors**.

### **Example Configuration (airflow.cfg)**

ini

CopyEdit

sql\_alchemy\_conn = postgresql+psycopg2://postgres:YourPassword@<CLOUD\_SQL\_IP>:5432/airflow\_metadata

### **Where It Is Used**

* Airflow **scheduler**, **webserver**, and **workers** use this database.
* Stores **DAG definitions, task instances, logs, and execution history**.

## **2. result\_backend (Celery Task State Storage)**

### **Purpose**

* Stores **Celery task execution results**.
* Required **only for CeleryExecutor**.
* Helps **workers fetch task results** from completed executions.

### **Example Configuration (airflow.cfg)**

ini

CopyEdit

result\_backend = db+postgresql://postgres:YourPassword@<CLOUD\_SQL\_IP>:5432/airflow\_metadata

(Here, we store task results in the same PostgreSQL database as Airflow metadata.)

OR, if using Redis (faster but not persistent):

ini

CopyEdit

result\_backend = redis://localhost:6379/0

### **Where It Is Used**

* Used **only when running CeleryExecutor**.
* Celery workers use it to store the **state of each task execution**.
* Can be configured with **PostgreSQL, MySQL, or Redis**.

## **3. Key Differences**

| **Feature** | **sql\_alchemy\_conn** | **result\_backend** |
| --- | --- | --- |
| **Purpose** | Stores **Airflow metadata** (DAGs, tasks, logs) | Stores **Celery task execution results** |
| **Required?** | ✅ Yes (for all executors) | ❌ No (only for CeleryExecutor) |
| **Used By** | Scheduler, Webserver, Workers | Celery Workers |
| **Supported Backends** | PostgreSQL, MySQL, SQLite (for testing) | PostgreSQL, MySQL, Redis, RPC |
| **Performance Impact** | Moderate (since it tracks everything in Airflow) | Can be high if many Celery tasks are running |

## **4. How They Work Together in CeleryExecutor**

1️⃣ **Airflow Scheduler** schedules a DAG and adds a task to the queue (stored in RabbitMQ).  
2️⃣ **Celery Worker** picks up the task and runs it.  
3️⃣ **Celery Worker** stores the task execution **result** in result\_backend.  
4️⃣ **Airflow Metadata DB (sql\_alchemy\_conn)** is updated to reflect the task state (e.g., running, success).  
5️⃣ **Airflow Webserver** fetches metadata from sql\_alchemy\_conn and task results from result\_backend.

### **5. Which One Should You Use?**

✅ **Use sql\_alchemy\_conn** **always** (required for Airflow).  
✅ **Use result\_backend** **only if using CeleryExecutor**.  
✅ If performance is critical, use **Redis for result\_backend** (faster than PostgreSQL).

### **Final Summary**

| **Setting** | **Purpose** | **Required?** | **Example Backend** |
| --- | --- | --- | --- |
| sql\_alchemy\_conn | Stores Airflow metadata (DAGs, logs, task history) | ✅ Yes (for all executors) | PostgreSQL, MySQL |
| result\_backend | Stores Celery task execution results | ❌ No (only for CeleryExecutor) | PostgreSQL, Redis, MySQL |

Now, you can configure your Airflow setup efficiently! 🚀 Let me know if you need more details. 😊

O

Chat

Airflow running issues

### **Issues Identified and How to Fix Them**

Your logs indicate **two major issues**:

**Airflow Webserver PID Conflict**:  
csharp  
CopyEdit  
Error: Already running on PID 93867 (or pid file '/home/airflow/airflow/airflow-webserver.pid' is stale)

* + This means another Airflow webserver instance is already running.
  + You need to **kill stale processes** and restart the webserver.

**Redis Connection Failure in Celery**:  
vbnet  
CopyEdit  
consumer: Cannot connect to redis://redis:6379/0: Error -2 connecting to redis:6379. Name or service not known.

* + The Celery workers **cannot connect to Redis**, which is needed as a message broker.
  + You need to **ensure Redis is running** and correctly configured.

## **Solution 1: Fix Airflow Webserver PID Conflict**

### **Step 1: Find Running Airflow Webserver Processes**

Run:

bash

CopyEdit

ps aux | grep airflow-webserver

If processes are running (example output):

plaintext

CopyEdit

airflow 93867 1.0 1.8 1662008 276696 ? Sl 18:29 0:06 gunicorn: master [airflow-webserver]

airflow 94404 0.0 1.3 1662008 206024 ? S 18:37 0:00 [ready] gunicorn: worker [airflow-webserver]

**Step 2: Kill the Running Processes**

bash

CopyEdit

sudo kill -9 93867 94404

**Step 3: Delete Stale PID File**

bash

CopyEdit

rm -f $AIRFLOW\_HOME/airflow-webserver.pid

**Step 4: Restart the Webserver**

bash

CopyEdit

airflow webserver -p 8080 &

## **Solution 2: Fix Redis Connection for Celery**

### **Step 1: Check If Redis is Running**

bash

CopyEdit

sudo systemctl status redis

If Redis is **not running**, start it:  
bash  
CopyEdit  
sudo systemctl start redis

To **enable it on boot**:  
bash  
CopyEdit  
sudo systemctl enable redis

### **Step 2: Verify Redis is Listening on Port 6379**

Run:

bash

CopyEdit

sudo netstat -tlnp | grep 6379

If Redis is running, you should see:

plaintext

CopyEdit

tcp 0 0 127.0.0.1:6379 0.0.0.0:\* LISTEN 1234/redis-server

If no output appears, **restart Redis**.

### **Step 3: Update Airflow Configuration (airflow.cfg)**

If Redis is running but Airflow **still cannot connect**, update the **broker URL** in airflow.cfg:

bash

CopyEdit

nano $AIRFLOW\_HOME/airflow.cfg

Find the [celery] section and update:

ini

CopyEdit

[celery]

broker\_url = redis://127.0.0.1:6379/0

result\_backend = db+postgresql://postgres:YourPassword@34.130.5.9:5432/airflow\_metadata

* **Fixes:**
  + Change redis://redis:6379/0 → redis://127.0.0.1:6379/0 (use **localhost** instead of the hostname).

Save and exit (Ctrl+O, Enter, Ctrl+X).

### **Step 4: Restart Celery Workers**

bash

CopyEdit

airflow celery worker &

If the issue persists, restart all Airflow services:

bash

CopyEdit

airflow db migrate

airflow scheduler &

airflow webserver -p 8080 &

airflow celery worker &

## **Final Checklist**

✅ **Fix Airflow webserver conflicts (kill -9 <PID> & delete airflow-webserver.pid)**✅ **Ensure Redis is running (sudo systemctl start redis)**✅ **Use 127.0.0.1 instead of redis in broker\_url**✅ **Restart Celery workers (airflow celery worker &)**

Now, your Airflow instance should work correctly! 🚀 Let me know if you need more help. 😊

airflow.cfg (only for reference)

[core]

# The folder where your airflow pipelines live, most likely a

# subfolder in a code repository. This path must be absolute.

#

# Variable: AIRFLOW\_\_CORE\_\_DAGS\_FOLDER

#

dags\_folder = /home/airflow/airflow/dags

# Hostname by providing a path to a callable, which will resolve the hostname.

# The format is "package.function".

#

# For example, default value ``airflow.utils.net.getfqdn`` means that result from patched

# version of `socket.getfqdn() <https://docs.python.org/3/library/socket.html#socket.getfqdn>`\_\_,

# see related `CPython Issue <https://github.com/python/cpython/issues/49254>`\_\_.

#

# No argument should be required in the function specified.

# If using IP address as hostname is preferred, use value ``airflow.utils.net.get\_host\_ip\_address``

#

# Variable: AIRFLOW\_\_CORE\_\_HOSTNAME\_CALLABLE

#

hostname\_callable = airflow.utils.net.getfqdn

# A callable to check if a python file has airflow dags defined or not and should

# return ``True`` if it has dags otherwise ``False``.

# If this is not provided, Airflow uses its own heuristic rules.

#

# The function should have the following signature

#

# .. code-block:: python

#

# def func\_name(file\_path: str, zip\_file: zipfile.ZipFile | None = None) -> bool: ...

#

# Variable: AIRFLOW\_\_CORE\_\_MIGHT\_CONTAIN\_DAG\_CALLABLE

#

might\_contain\_dag\_callable = airflow.utils.file.might\_contain\_dag\_via\_default\_heuristic

# Default timezone in case supplied date times are naive

# can be `UTC` (default), `system`, or any `IANA <https://www.iana.org/time-zones>`

# timezone string (e.g. Europe/Amsterdam)

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_TIMEZONE

#

default\_timezone = utc

# The executor class that airflow should use. Choices include

# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``,

# ``KubernetesExecutor``, ``CeleryKubernetesExecutor``, ``LocalKubernetesExecutor`` or the

# full import path to the class when using a custom executor.

#

# Variable: AIRFLOW\_\_CORE\_\_EXECUTOR

#

executor = CeleryExecutor

# The auth manager class that airflow should use. Full import path to the auth manager class.

#

# Variable: AIRFLOW\_\_CORE\_\_AUTH\_MANAGER

#

auth\_manager = airflow.providers.fab.auth\_manager.fab\_auth\_manager.FabAuthManager

# This defines the maximum number of task instances that can run concurrently per scheduler in

# Airflow, regardless of the worker count. Generally this value, multiplied by the number of

# schedulers in your cluster, is the maximum number of task instances with the running

# state in the metadata database. Setting this value to zero allows unlimited parallelism.

#

# Variable: AIRFLOW\_\_CORE\_\_PARALLELISM

#

parallelism = 32

# The maximum number of task instances allowed to run concurrently in each DAG. To calculate

# the number of tasks that is running concurrently for a DAG, add up the number of running

# tasks for all DAG runs of the DAG. This is configurable at the DAG level with ``max\_active\_tasks``,

# which is defaulted as ``[core] max\_active\_tasks\_per\_dag``.

#

# An example scenario when this would be useful is when you want to stop a new dag with an early

# start date from stealing all the executor slots in a cluster.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_ACTIVE\_TASKS\_PER\_DAG

#

max\_active\_tasks\_per\_dag = 16

# Are DAGs paused by default at creation

#

# Variable: AIRFLOW\_\_CORE\_\_DAGS\_ARE\_PAUSED\_AT\_CREATION

#

dags\_are\_paused\_at\_creation = True

# The maximum number of active DAG runs per DAG. The scheduler will not create more DAG runs

# if it reaches the limit. This is configurable at the DAG level with ``max\_active\_runs``,

# which is defaulted as ``[core] max\_active\_runs\_per\_dag``.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_ACTIVE\_RUNS\_PER\_DAG

#

max\_active\_runs\_per\_dag = 16

# (experimental) The maximum number of consecutive DAG failures before DAG is automatically paused.

# This is also configurable per DAG level with ``max\_consecutive\_failed\_dag\_runs``,

# which is defaulted as ``[core] max\_consecutive\_failed\_dag\_runs\_per\_dag``.

# If not specified, then the value is considered as 0,

# meaning that the dags are never paused out by default.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_CONSECUTIVE\_FAILED\_DAG\_RUNS\_PER\_DAG

#

max\_consecutive\_failed\_dag\_runs\_per\_dag = 0

# The name of the method used in order to start Python processes via the multiprocessing module.

# This corresponds directly with the options available in the Python docs:

# `multiprocessing.set\_start\_method

# <https://docs.python.org/3/library/multiprocessing.html#multiprocessing.set\_start\_method>`\_\_

# must be one of the values returned by `multiprocessing.get\_all\_start\_methods()

# <https://docs.python.org/3/library/multiprocessing.html#multiprocessing.get\_all\_start\_methods>`\_\_.

#

# Example: mp\_start\_method = fork

#

# Variable: AIRFLOW\_\_CORE\_\_MP\_START\_METHOD

#

# mp\_start\_method =

# Whether to load the DAG examples that ship with Airflow. It's good to

# get started, but you probably want to set this to ``False`` in a production

# environment

#

# Variable: AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES

#

load\_examples = True

# Path to the folder containing Airflow plugins

#

# Variable: AIRFLOW\_\_CORE\_\_PLUGINS\_FOLDER

#

plugins\_folder = /home/airflow/airflow/plugins

# Should tasks be executed via forking of the parent process

#

# \* ``False``: Execute via forking of the parent process

# \* ``True``: Spawning a new python process, slower than fork, but means plugin changes picked

# up by tasks straight away

#

# Variable: AIRFLOW\_\_CORE\_\_EXECUTE\_TASKS\_NEW\_PYTHON\_INTERPRETER

#

execute\_tasks\_new\_python\_interpreter = False

# Secret key to save connection passwords in the db

#

# Variable: AIRFLOW\_\_CORE\_\_FERNET\_KEY

#

fernet\_key =

# Whether to disable pickling dags

#

# Variable: AIRFLOW\_\_CORE\_\_DONOT\_PICKLE

#

donot\_pickle = True

# How long before timing out a python file import

#

# Variable: AIRFLOW\_\_CORE\_\_DAGBAG\_IMPORT\_TIMEOUT

#

dagbag\_import\_timeout = 30.0

# Should a traceback be shown in the UI for dagbag import errors,

# instead of just the exception message

#

# Variable: AIRFLOW\_\_CORE\_\_DAGBAG\_IMPORT\_ERROR\_TRACEBACKS

#

dagbag\_import\_error\_tracebacks = True

# If tracebacks are shown, how many entries from the traceback should be shown

#

# Variable: AIRFLOW\_\_CORE\_\_DAGBAG\_IMPORT\_ERROR\_TRACEBACK\_DEPTH

#

dagbag\_import\_error\_traceback\_depth = 2

# How long before timing out a DagFileProcessor, which processes a dag file

#

# Variable: AIRFLOW\_\_CORE\_\_DAG\_FILE\_PROCESSOR\_TIMEOUT

#

dag\_file\_processor\_timeout = 50

# The class to use for running task instances in a subprocess.

# Choices include StandardTaskRunner, CgroupTaskRunner or the full import path to the class

# when using a custom task runner.

#

# Variable: AIRFLOW\_\_CORE\_\_TASK\_RUNNER

#

task\_runner = StandardTaskRunner

# If set, tasks without a ``run\_as\_user`` argument will be run with this user

# Can be used to de-elevate a sudo user running Airflow when executing tasks

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_IMPERSONATION

#

default\_impersonation =

# What security module to use (for example kerberos)

#

# Variable: AIRFLOW\_\_CORE\_\_SECURITY

#

security =

# Turn unit test mode on (overwrites many configuration options with test

# values at runtime)

#

# Variable: AIRFLOW\_\_CORE\_\_UNIT\_TEST\_MODE

#

unit\_test\_mode = False

# Whether to enable pickling for xcom (note that this is insecure and allows for

# RCE exploits).

#

# Variable: AIRFLOW\_\_CORE\_\_ENABLE\_XCOM\_PICKLING

#

enable\_xcom\_pickling = False

# What classes can be imported during deserialization. This is a multi line value.

# The individual items will be parsed as a pattern to a glob function.

# Python built-in classes (like dict) are always allowed.

#

# Variable: AIRFLOW\_\_CORE\_\_ALLOWED\_DESERIALIZATION\_CLASSES

#

allowed\_deserialization\_classes = airflow.\*

# What classes can be imported during deserialization. This is a multi line value.

# The individual items will be parsed as regexp patterns.

# This is a secondary option to ``[core] allowed\_deserialization\_classes``.

#

# Variable: AIRFLOW\_\_CORE\_\_ALLOWED\_DESERIALIZATION\_CLASSES\_REGEXP

#

allowed\_deserialization\_classes\_regexp =

# When a task is killed forcefully, this is the amount of time in seconds that

# it has to cleanup after it is sent a SIGTERM, before it is SIGKILLED

#

# Variable: AIRFLOW\_\_CORE\_\_KILLED\_TASK\_CLEANUP\_TIME

#

killed\_task\_cleanup\_time = 60

# Whether to override params with dag\_run.conf. If you pass some key-value pairs

# through ``airflow dags backfill -c`` or

# ``airflow dags trigger -c``, the key-value pairs will override the existing ones in params.

#

# Variable: AIRFLOW\_\_CORE\_\_DAG\_RUN\_CONF\_OVERRIDES\_PARAMS

#

dag\_run\_conf\_overrides\_params = True

# If enabled, Airflow will only scan files containing both ``DAG`` and ``airflow`` (case-insensitive).

#

# Variable: AIRFLOW\_\_CORE\_\_DAG\_DISCOVERY\_SAFE\_MODE

#

dag\_discovery\_safe\_mode = True

# The pattern syntax used in the

# `.airflowignore

# <https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html#airflowignore>`\_\_

# files in the DAG directories. Valid values are ``regexp`` or ``glob``.

#

# Variable: AIRFLOW\_\_CORE\_\_DAG\_IGNORE\_FILE\_SYNTAX

#

dag\_ignore\_file\_syntax = regexp

# The number of retries each task is going to have by default. Can be overridden at dag or task level.

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_TASK\_RETRIES

#

default\_task\_retries = 0

# The number of seconds each task is going to wait by default between retries. Can be overridden at

# dag or task level.

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_TASK\_RETRY\_DELAY

#

default\_task\_retry\_delay = 300

# The maximum delay (in seconds) each task is going to wait by default between retries.

# This is a global setting and cannot be overridden at task or DAG level.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_TASK\_RETRY\_DELAY

#

max\_task\_retry\_delay = 86400

# The weighting method used for the effective total priority weight of the task

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_TASK\_WEIGHT\_RULE

#

default\_task\_weight\_rule = downstream

# Maximum possible time (in seconds) that task will have for execution of auxiliary processes

# (like listeners, mini scheduler...) after task is marked as success..

#

# Variable: AIRFLOW\_\_CORE\_\_TASK\_SUCCESS\_OVERTIME

#

task\_success\_overtime = 20

# The default task execution\_timeout value for the operators. Expected an integer value to

# be passed into timedelta as seconds. If not specified, then the value is considered as None,

# meaning that the operators are never timed out by default.

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_TASK\_EXECUTION\_TIMEOUT

#

default\_task\_execution\_timeout =

# Updating serialized DAG can not be faster than a minimum interval to reduce database write rate.

#

# Variable: AIRFLOW\_\_CORE\_\_MIN\_SERIALIZED\_DAG\_UPDATE\_INTERVAL

#

min\_serialized\_dag\_update\_interval = 30

# If ``True``, serialized DAGs are compressed before writing to DB.

#

# .. note::

#

# This will disable the DAG dependencies view

#

# Variable: AIRFLOW\_\_CORE\_\_COMPRESS\_SERIALIZED\_DAGS

#

compress\_serialized\_dags = False

# Fetching serialized DAG can not be faster than a minimum interval to reduce database

# read rate. This config controls when your DAGs are updated in the Webserver

#

# Variable: AIRFLOW\_\_CORE\_\_MIN\_SERIALIZED\_DAG\_FETCH\_INTERVAL

#

min\_serialized\_dag\_fetch\_interval = 10

# Maximum number of Rendered Task Instance Fields (Template Fields) per task to store

# in the Database.

# All the template\_fields for each of Task Instance are stored in the Database.

# Keeping this number small may cause an error when you try to view ``Rendered`` tab in

# TaskInstance view for older tasks.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_NUM\_RENDERED\_TI\_FIELDS\_PER\_TASK

#

max\_num\_rendered\_ti\_fields\_per\_task = 30

# On each dagrun check against defined SLAs

#

# Variable: AIRFLOW\_\_CORE\_\_CHECK\_SLAS

#

check\_slas = True

# Path to custom XCom class that will be used to store and resolve operators results

#

# Example: xcom\_backend = path.to.CustomXCom

#

# Variable: AIRFLOW\_\_CORE\_\_XCOM\_BACKEND

#

xcom\_backend = airflow.models.xcom.BaseXCom

# By default Airflow plugins are lazily-loaded (only loaded when required). Set it to ``False``,

# if you want to load plugins whenever 'airflow' is invoked via cli or loaded from module.

#

# Variable: AIRFLOW\_\_CORE\_\_LAZY\_LOAD\_PLUGINS

#

lazy\_load\_plugins = True

# By default Airflow providers are lazily-discovered (discovery and imports happen only when required).

# Set it to ``False``, if you want to discover providers whenever 'airflow' is invoked via cli or

# loaded from module.

#

# Variable: AIRFLOW\_\_CORE\_\_LAZY\_DISCOVER\_PROVIDERS

#

lazy\_discover\_providers = True

# Hide sensitive \*\*Variables\*\* or \*\*Connection extra json keys\*\* from UI

# and task logs when set to ``True``

#

# .. note::

#

# Connection passwords are always hidden in logs

#

# Variable: AIRFLOW\_\_CORE\_\_HIDE\_SENSITIVE\_VAR\_CONN\_FIELDS

#

hide\_sensitive\_var\_conn\_fields = True

# A comma-separated list of extra sensitive keywords to look for in variables names or connection's

# extra JSON.

#

# Variable: AIRFLOW\_\_CORE\_\_SENSITIVE\_VAR\_CONN\_NAMES

#

sensitive\_var\_conn\_names =

# Task Slot counts for ``default\_pool``. This setting would not have any effect in an existing

# deployment where the ``default\_pool`` is already created. For existing deployments, users can

# change the number of slots using Webserver, API or the CLI

#

# Variable: AIRFLOW\_\_CORE\_\_DEFAULT\_POOL\_TASK\_SLOT\_COUNT

#

default\_pool\_task\_slot\_count = 128

# The maximum list/dict length an XCom can push to trigger task mapping. If the pushed list/dict has a

# length exceeding this value, the task pushing the XCom will be failed automatically to prevent the

# mapped tasks from clogging the scheduler.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_MAP\_LENGTH

#

max\_map\_length = 1024

# The default umask to use for process when run in daemon mode (scheduler, worker, etc.)

#

# This controls the file-creation mode mask which determines the initial value of file permission bits

# for newly created files.

#

# This value is treated as an octal-integer.

#

# Variable: AIRFLOW\_\_CORE\_\_DAEMON\_UMASK

#

daemon\_umask = 0o077

# Class to use as dataset manager.

#

# Example: dataset\_manager\_class = airflow.datasets.manager.DatasetManager

#

# Variable: AIRFLOW\_\_CORE\_\_DATASET\_MANAGER\_CLASS

#

# dataset\_manager\_class =

# Kwargs to supply to dataset manager.

#

# Example: dataset\_manager\_kwargs = {"some\_param": "some\_value"}

#

# Variable: AIRFLOW\_\_CORE\_\_DATASET\_MANAGER\_KWARGS

#

# dataset\_manager\_kwargs =

# Dataset URI validation should raise an exception if it is not compliant with AIP-60.

# By default this configuration is false, meaning that Airflow 2.x only warns the user.

# In Airflow 3, this configuration will be removed, unconditionally enabling strict validation.

#

# Variable: AIRFLOW\_\_CORE\_\_STRICT\_DATASET\_URI\_VALIDATION

#

strict\_dataset\_uri\_validation = False

# (experimental) Whether components should use Airflow Internal API for DB connectivity.

#

# Variable: AIRFLOW\_\_CORE\_\_DATABASE\_ACCESS\_ISOLATION

#

database\_access\_isolation = False

# (experimental) Airflow Internal API url.

# Only used if ``[core] database\_access\_isolation`` is ``True``.

#

# Example: internal\_api\_url = http://localhost:8080

#

# Variable: AIRFLOW\_\_CORE\_\_INTERNAL\_API\_URL

#

# internal\_api\_url =

# Secret key used to authenticate internal API clients to core. It should be as random as possible.

# However, when running more than 1 instances of webserver / internal API services, make sure all

# of them use the same ``secret\_key`` otherwise calls will fail on authentication.

# The authentication token generated using the secret key has a short expiry time though - make

# sure that time on ALL the machines that you run airflow components on is synchronized

# (for example using ntpd) otherwise you might get "forbidden" errors when the logs are accessed.

#

# Variable: AIRFLOW\_\_CORE\_\_INTERNAL\_API\_SECRET\_KEY

#

internal\_api\_secret\_key = 3dDCLAL4X63Zd1a//soPMQ==

# The ability to allow testing connections across Airflow UI, API and CLI.

# Supported options: ``Disabled``, ``Enabled``, ``Hidden``. Default: Disabled

# Disabled - Disables the test connection functionality and disables the Test Connection button in UI.

# Enabled - Enables the test connection functionality and shows the Test Connection button in UI.

# Hidden - Disables the test connection functionality and hides the Test Connection button in UI.

# Before setting this to Enabled, make sure that you review the users who are able to add/edit

# connections and ensure they are trusted. Connection testing can be done maliciously leading to

# undesired and insecure outcomes.

# See `Airflow Security Model: Capabilities of authenticated UI users

# <https://airflow.apache.org/docs/apache-airflow/stable/security/security\_model.html#capabilities-of-authenticated-ui-users>`\_\_

# for more details.

#

# Variable: AIRFLOW\_\_CORE\_\_TEST\_CONNECTION

#

test\_connection = Disabled

# The maximum length of the rendered template field. If the value to be stored in the

# rendered template field exceeds this size, it's redacted.

#

# Variable: AIRFLOW\_\_CORE\_\_MAX\_TEMPLATED\_FIELD\_LENGTH

#

max\_templated\_field\_length = 4096

[database]

# Path to the ``alembic.ini`` file. You can either provide the file path relative

# to the Airflow home directory or the absolute path if it is located elsewhere.

#

# Variable: AIRFLOW\_\_DATABASE\_\_ALEMBIC\_INI\_FILE\_PATH

#

alembic\_ini\_file\_path = alembic.ini

# The SQLAlchemy connection string to the metadata database.

# SQLAlchemy supports many different database engines.

# See: `Set up a Database Backend: Database URI

# <https://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#database-uri>`\_\_

# for more details.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN

#

# sql\_alchemy\_conn = sqlite:////home/airflow/airflow/airflow.db

sql\_alchemy\_conn = postgresql+psycopg2://postgres:StrongPass%%40123@34.130.5.9:5432/airflow\_metadata

#Manuuly added by malik

broker\_url = pyamqp://guest:guest@localhost:5672/

result\_backend = postgresql+psycopg2://postgres:StrongPass%%40123@34.130.5.9:5432/airflow\_metadata

# Extra engine specific keyword args passed to SQLAlchemy's create\_engine, as a JSON-encoded value

#

# Example: sql\_alchemy\_engine\_args = {"arg1": true}

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_ENGINE\_ARGS

#

# sql\_alchemy\_engine\_args =

# The encoding for the databases

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ENGINE\_ENCODING

#

sql\_engine\_encoding = utf-8

# Collation for ``dag\_id``, ``task\_id``, ``key``, ``external\_executor\_id`` columns

# in case they have different encoding.

# By default this collation is the same as the database collation, however for ``mysql`` and ``mariadb``

# the default is ``utf8mb3\_bin`` so that the index sizes of our index keys will not exceed

# the maximum size of allowed index when collation is set to ``utf8mb4`` variant, see

# `GitHub Issue Comment <https://github.com/apache/airflow/pull/17603#issuecomment-901121618>`\_\_

# for more details.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ENGINE\_COLLATION\_FOR\_IDS

#

# sql\_engine\_collation\_for\_ids =

# If SQLAlchemy should pool database connections.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_POOL\_ENABLED

#

sql\_alchemy\_pool\_enabled = True

# The SQLAlchemy pool size is the maximum number of database connections

# in the pool. 0 indicates no limit.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_POOL\_SIZE

#

sql\_alchemy\_pool\_size = 5

# The maximum overflow size of the pool.

# When the number of checked-out connections reaches the size set in pool\_size,

# additional connections will be returned up to this limit.

# When those additional connections are returned to the pool, they are disconnected and discarded.

# It follows then that the total number of simultaneous connections the pool will allow

# is \*\*pool\_size\*\* + \*\*max\_overflow\*\*,

# and the total number of "sleeping" connections the pool will allow is pool\_size.

# max\_overflow can be set to ``-1`` to indicate no overflow limit;

# no limit will be placed on the total number of concurrent connections. Defaults to ``10``.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_MAX\_OVERFLOW

#

sql\_alchemy\_max\_overflow = 10

# The SQLAlchemy pool recycle is the number of seconds a connection

# can be idle in the pool before it is invalidated. This config does

# not apply to sqlite. If the number of DB connections is ever exceeded,

# a lower config value will allow the system to recover faster.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_POOL\_RECYCLE

#

sql\_alchemy\_pool\_recycle = 1800

# Check connection at the start of each connection pool checkout.

# Typically, this is a simple statement like "SELECT 1".

# See `SQLAlchemy Pooling: Disconnect Handling - Pessimistic

# <https://docs.sqlalchemy.org/en/14/core/pooling.html#disconnect-handling-pessimistic>`\_\_

# for more details.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_POOL\_PRE\_PING

#

sql\_alchemy\_pool\_pre\_ping = True

# The schema to use for the metadata database.

# SQLAlchemy supports databases with the concept of multiple schemas.

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_SCHEMA

#

sql\_alchemy\_schema =

# Import path for connect args in SQLAlchemy. Defaults to an empty dict.

# This is useful when you want to configure db engine args that SQLAlchemy won't parse

# in connection string. This can be set by passing a dictionary containing the create engine parameters.

# For more details about passing create engine parameters (keepalives variables, timeout etc)

# in Postgres DB Backend see `Setting up a PostgreSQL Database

# <https://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#setting-up-a-postgresql-database>`\_\_

# e.g ``connect\_args={"timeout":30}`` can be defined in ``airflow\_local\_settings.py`` and

# can be imported as shown below

#

# Example: sql\_alchemy\_connect\_args = airflow\_local\_settings.connect\_args

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONNECT\_ARGS

#

# sql\_alchemy\_connect\_args =

# Important Warning: Use of sql\_alchemy\_session\_maker Highly Discouraged

# Import path for function which returns 'sqlalchemy.orm.sessionmaker'.

# Improper configuration of sql\_alchemy\_session\_maker can lead to serious issues,

# including data corruption, unrecoverable application crashes. Please review the SQLAlchemy

# documentation for detailed guidance on proper configuration and best practices.

#

# Example: sql\_alchemy\_session\_maker = airflow\_local\_settings.\_sessionmaker

#

# Variable: AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_SESSION\_MAKER

#

# sql\_alchemy\_session\_maker =

# Whether to load the default connections that ship with Airflow when ``airflow db init`` is called.

# It's good to get started, but you probably want to set this to ``False`` in a production environment.

#

# Variable: AIRFLOW\_\_DATABASE\_\_LOAD\_DEFAULT\_CONNECTIONS

#

load\_default\_connections = True

# Number of times the code should be retried in case of DB Operational Errors.

# Not all transactions will be retried as it can cause undesired state.

# Currently it is only used in ``DagFileProcessor.process\_file`` to retry ``dagbag.sync\_to\_db``.

#

# Variable: AIRFLOW\_\_DATABASE\_\_MAX\_DB\_RETRIES

#

max\_db\_retries = 3

# Whether to run alembic migrations during Airflow start up. Sometimes this operation can be expensive,

# and the users can assert the correct version through other means (e.g. through a Helm chart).

# Accepts ``True`` or ``False``.

#

# Variable: AIRFLOW\_\_DATABASE\_\_CHECK\_MIGRATIONS

#

check\_migrations = True

[logging]

# The folder where airflow should store its log files.

# This path must be absolute.

# There are a few existing configurations that assume this is set to the default.

# If you choose to override this you may need to update the

# ``[logging] dag\_processor\_manager\_log\_location`` and

# ``[logging] child\_process\_log\_directory settings`` as well.

#

# Variable: AIRFLOW\_\_LOGGING\_\_BASE\_LOG\_FOLDER

#

base\_log\_folder = /home/airflow/airflow/logs

# Airflow can store logs remotely in AWS S3, Google Cloud Storage or Elastic Search.

# Set this to ``True`` if you want to enable remote logging.

#

# Variable: AIRFLOW\_\_LOGGING\_\_REMOTE\_LOGGING

#

remote\_logging = True

# Users must supply an Airflow connection id that provides access to the storage

# location. Depending on your remote logging service, this may only be used for

# reading logs, not writing them.

#

# Variable: AIRFLOW\_\_LOGGING\_\_REMOTE\_LOG\_CONN\_ID

#

remote\_log\_conn\_id = gcs\_default

# Whether the local log files for GCS, S3, WASB and OSS remote logging should be deleted after

# they are uploaded to the remote location.

#

# Variable: AIRFLOW\_\_LOGGING\_\_DELETE\_LOCAL\_LOGS

#

delete\_local\_logs = False

# Path to Google Credential JSON file. If omitted, authorization based on `the Application Default

# Credentials

# <https://cloud.google.com/docs/authentication/application-default-credentials>`\_\_ will

# be used.

#

# Variable: AIRFLOW\_\_LOGGING\_\_GOOGLE\_KEY\_PATH

#

google\_key\_path =

# Storage bucket URL for remote logging

# S3 buckets should start with \*\*s3://\*\*

# Cloudwatch log groups should start with \*\*cloudwatch://\*\*

# GCS buckets should start with \*\*gs://\*\*

# WASB buckets should start with \*\*wasb\*\* just to help Airflow select correct handler

# Stackdriver logs should start with \*\*stackdriver://\*\*

#

# Variable: AIRFLOW\_\_LOGGING\_\_REMOTE\_BASE\_LOG\_FOLDER

#

remote\_base\_log\_folder = gs://airflow-logs

# The remote\_task\_handler\_kwargs param is loaded into a dictionary and passed to the ``\_\_init\_\_``

# of remote task handler and it overrides the values provided by Airflow config. For example if you set

# ``delete\_local\_logs=False`` and you provide ``{"delete\_local\_copy": true}``, then the local

# log files will be deleted after they are uploaded to remote location.

#

# Example: remote\_task\_handler\_kwargs = {"delete\_local\_copy": true}

#

# Variable: AIRFLOW\_\_LOGGING\_\_REMOTE\_TASK\_HANDLER\_KWARGS

#

remote\_task\_handler\_kwargs =

# Use server-side encryption for logs stored in S3

#

# Variable: AIRFLOW\_\_LOGGING\_\_ENCRYPT\_S3\_LOGS

#

encrypt\_s3\_logs = False

# Logging level.

#

# Supported values: ``CRITICAL``, ``ERROR``, ``WARNING``, ``INFO``, ``DEBUG``.

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOGGING\_LEVEL

#

logging\_level = INFO

# Logging level for celery. If not set, it uses the value of logging\_level

#

# Supported values: ``CRITICAL``, ``ERROR``, ``WARNING``, ``INFO``, ``DEBUG``.

#

# Variable: AIRFLOW\_\_LOGGING\_\_CELERY\_LOGGING\_LEVEL

#

celery\_logging\_level =

# Logging level for Flask-appbuilder UI.

#

# Supported values: ``CRITICAL``, ``ERROR``, ``WARNING``, ``INFO``, ``DEBUG``.

#

# Variable: AIRFLOW\_\_LOGGING\_\_FAB\_LOGGING\_LEVEL

#

fab\_logging\_level = WARNING

# Logging class

# Specify the class that will specify the logging configuration

# This class has to be on the python classpath

#

# Example: logging\_config\_class = my.path.default\_local\_settings.LOGGING\_CONFIG

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOGGING\_CONFIG\_CLASS

#

logging\_config\_class =

# Flag to enable/disable Colored logs in Console

# Colour the logs when the controlling terminal is a TTY.

#

# Variable: AIRFLOW\_\_LOGGING\_\_COLORED\_CONSOLE\_LOG

#

colored\_console\_log = True

# Log format for when Colored logs is enabled

#

# Variable: AIRFLOW\_\_LOGGING\_\_COLORED\_LOG\_FORMAT

#

colored\_log\_format = [%%(blue)s%%(asctime)s%%(reset)s] {%%(blue)s%%(filename)s:%%(reset)s%%(lineno)d} %%(log\_color)s%%(levelname)s%%(reset)s - %%(log\_color)s%%(message)s%%(reset)s

# Specifies the class utilized by Airflow to implement colored logging

#

# Variable: AIRFLOW\_\_LOGGING\_\_COLORED\_FORMATTER\_CLASS

#

colored\_formatter\_class = airflow.utils.log.colored\_log.CustomTTYColoredFormatter

# Format of Log line

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOG\_FORMAT

#

log\_format = [%%(asctime)s] {%%(filename)s:%%(lineno)d} %%(levelname)s - %%(message)s

# Defines the format of log messages for simple logging configuration

#

# Variable: AIRFLOW\_\_LOGGING\_\_SIMPLE\_LOG\_FORMAT

#

simple\_log\_format = %%(asctime)s %%(levelname)s - %%(message)s

# Where to send dag parser logs. If "file", logs are sent to log files defined by child\_process\_log\_directory.

#

# Variable: AIRFLOW\_\_LOGGING\_\_DAG\_PROCESSOR\_LOG\_TARGET

#

dag\_processor\_log\_target = file

# Format of Dag Processor Log line

#

# Variable: AIRFLOW\_\_LOGGING\_\_DAG\_PROCESSOR\_LOG\_FORMAT

#

dag\_processor\_log\_format = [%%(asctime)s] [SOURCE:DAG\_PROCESSOR] {%%(filename)s:%%(lineno)d} %%(levelname)s - %%(message)s

# Determines the formatter class used by Airflow for structuring its log messages

# The default formatter class is timezone-aware, which means that timestamps attached to log entries

# will be adjusted to reflect the local timezone of the Airflow instance

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOG\_FORMATTER\_CLASS

#

log\_formatter\_class = airflow.utils.log.timezone\_aware.TimezoneAware

# An import path to a function to add adaptations of each secret added with

# ``airflow.utils.log.secrets\_masker.mask\_secret`` to be masked in log messages. The given function

# is expected to require a single parameter: the secret to be adapted. It may return a

# single adaptation of the secret or an iterable of adaptations to each be masked as secrets.

# The original secret will be masked as well as any adaptations returned.

#

# Example: secret\_mask\_adapter = urllib.parse.quote

#

# Variable: AIRFLOW\_\_LOGGING\_\_SECRET\_MASK\_ADAPTER

#

secret\_mask\_adapter =

# Specify prefix pattern like mentioned below with stream handler ``TaskHandlerWithCustomFormatter``

#

# Example: task\_log\_prefix\_template = {{ti.dag\_id}}-{{ti.task\_id}}-{{execution\_date}}-{{ti.try\_number}}

#

# Variable: AIRFLOW\_\_LOGGING\_\_TASK\_LOG\_PREFIX\_TEMPLATE

#

task\_log\_prefix\_template =

# Formatting for how airflow generates file names/paths for each task run.

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOG\_FILENAME\_TEMPLATE

#

log\_filename\_template = dag\_id={{ ti.dag\_id }}/run\_id={{ ti.run\_id }}/task\_id={{ ti.task\_id }}/{%% if ti.map\_index >= 0 %%}map\_index={{ ti.map\_index }}/{%% endif %%}attempt={{ try\_number }}.log

# Formatting for how airflow generates file names for log

#

# Variable: AIRFLOW\_\_LOGGING\_\_LOG\_PROCESSOR\_FILENAME\_TEMPLATE

#

log\_processor\_filename\_template = {{ filename }}.log

# Full path of dag\_processor\_manager logfile.

#

# Variable: AIRFLOW\_\_LOGGING\_\_DAG\_PROCESSOR\_MANAGER\_LOG\_LOCATION

#

dag\_processor\_manager\_log\_location = /home/airflow/airflow/logs/dag\_processor\_manager/dag\_processor\_manager.log

# Whether DAG processor manager will write logs to stdout

#

# Variable: AIRFLOW\_\_LOGGING\_\_DAG\_PROCESSOR\_MANAGER\_LOG\_STDOUT

#

dag\_processor\_manager\_log\_stdout = False

# Name of handler to read task instance logs.

# Defaults to use ``task`` handler.

#

# Variable: AIRFLOW\_\_LOGGING\_\_TASK\_LOG\_READER

#

task\_log\_reader = task

# A comma\-separated list of third-party logger names that will be configured to print messages to

# consoles\.

#

# Example: extra\_logger\_names = connexion,sqlalchemy

#

# Variable: AIRFLOW\_\_LOGGING\_\_EXTRA\_LOGGER\_NAMES

#

extra\_logger\_names =

# When you start an Airflow worker, Airflow starts a tiny web server

# subprocess to serve the workers local log files to the airflow main

# web server, who then builds pages and sends them to users. This defines

# the port on which the logs are served. It needs to be unused, and open

# visible from the main web server to connect into the workers.

#

# Variable: AIRFLOW\_\_LOGGING\_\_WORKER\_LOG\_SERVER\_PORT

#

worker\_log\_server\_port = 8793

# Port to serve logs from for triggerer.

# See ``[logging] worker\_log\_server\_port`` description for more info.

#

# Variable: AIRFLOW\_\_LOGGING\_\_TRIGGER\_LOG\_SERVER\_PORT

#

trigger\_log\_server\_port = 8794

# We must parse timestamps to interleave logs between trigger and task. To do so,

# we need to parse timestamps in log files. In case your log format is non-standard,

# you may provide import path to callable which takes a string log line and returns

# the timestamp (datetime.datetime compatible).

#

# Example: interleave\_timestamp\_parser = path.to.my\_func

#

# Variable: AIRFLOW\_\_LOGGING\_\_INTERLEAVE\_TIMESTAMP\_PARSER

#

# interleave\_timestamp\_parser =

# Permissions in the form or of octal string as understood by chmod. The permissions are important

# when you use impersonation, when logs are written by a different user than airflow. The most secure

# way of configuring it in this case is to add both users to the same group and make it the default

# group of both users. Group-writeable logs are default in airflow, but you might decide that you are

# OK with having the logs other-writeable, in which case you should set it to ``0o777``. You might

# decide to add more security if you do not use impersonation and change it to ``0o755`` to make it

# only owner-writeable. You can also make it just readable only for owner by changing it to ``0o700``

# if all the access (read/write) for your logs happens from the same user.

#

# Example: file\_task\_handler\_new\_folder\_permissions = 0o775

#

# Variable: AIRFLOW\_\_LOGGING\_\_FILE\_TASK\_HANDLER\_NEW\_FOLDER\_PERMISSIONS

#

file\_task\_handler\_new\_folder\_permissions = 0o775

# Permissions in the form or of octal string as understood by chmod. The permissions are important

# when you use impersonation, when logs are written by a different user than airflow. The most secure

# way of configuring it in this case is to add both users to the same group and make it the default

# group of both users. Group-writeable logs are default in airflow, but you might decide that you are

# OK with having the logs other-writeable, in which case you should set it to ``0o666``. You might

# decide to add more security if you do not use impersonation and change it to ``0o644`` to make it

# only owner-writeable. You can also make it just readable only for owner by changing it to ``0o600``

# if all the access (read/write) for your logs happens from the same user.

#

# Example: file\_task\_handler\_new\_file\_permissions = 0o664

#

# Variable: AIRFLOW\_\_LOGGING\_\_FILE\_TASK\_HANDLER\_NEW\_FILE\_PERMISSIONS

#

file\_task\_handler\_new\_file\_permissions = 0o664

# By default Celery sends all logs into stderr.

# If enabled any previous logging handlers will get \*removed\*.

# With this option AirFlow will create new handlers

# and send low level logs like INFO and WARNING to stdout,

# while sending higher severity logs to stderr.

#

# Variable: AIRFLOW\_\_LOGGING\_\_CELERY\_STDOUT\_STDERR\_SEPARATION

#

celery\_stdout\_stderr\_separation = False

# If enabled, Airflow may ship messages to task logs from outside the task run context, e.g. from

# the scheduler, executor, or callback execution context. This can help in circumstances such as

# when there's something blocking the execution of the task and ordinarily there may be no task

# logs at all.

# This is set to ``True`` by default. If you encounter issues with this feature

# (e.g. scheduler performance issues) it can be disabled.

#

# Variable: AIRFLOW\_\_LOGGING\_\_ENABLE\_TASK\_CONTEXT\_LOGGER

#

enable\_task\_context\_logger = True

# A comma separated list of keywords related to errors whose presence should display the line in red

# color in UI

#

# Variable: AIRFLOW\_\_LOGGING\_\_COLOR\_LOG\_ERROR\_KEYWORDS

#

color\_log\_error\_keywords = error,exception

# A comma separated list of keywords related to warning whose presence should display the line in yellow

# color in UI

#

# Variable: AIRFLOW\_\_LOGGING\_\_COLOR\_LOG\_WARNING\_KEYWORDS

#

color\_log\_warning\_keywords = warn

[metrics]

# `StatsD <https://github.com/statsd/statsd>`\_\_ integration settings.

# If true, ``[metrics] metrics\_allow\_list`` and ``[metrics] metrics\_block\_list`` will use

# regex pattern matching anywhere within the metric name instead of only prefix matching

# at the start of the name.

#

# Variable: AIRFLOW\_\_METRICS\_\_METRICS\_USE\_PATTERN\_MATCH

#

metrics\_use\_pattern\_match = False

# Configure an allow list (comma separated string) to send only certain metrics.

# If ``[metrics] metrics\_use\_pattern\_match`` is ``false``, match only the exact metric name prefix.

# If ``[metrics] metrics\_use\_pattern\_match`` is ``true``, provide regex patterns to match.

#

# Example: metrics\_allow\_list = "scheduler,executor,dagrun,pool,triggerer,celery" or "^scheduler,^executor,heartbeat|timeout"

#

# Variable: AIRFLOW\_\_METRICS\_\_METRICS\_ALLOW\_LIST

#

metrics\_allow\_list =

# Configure a block list (comma separated string) to block certain metrics from being emitted.

# If ``[metrics] metrics\_allow\_list`` and ``[metrics] metrics\_block\_list`` are both configured,

# ``[metrics] metrics\_block\_list`` is ignored.

#

# If ``[metrics] metrics\_use\_pattern\_match`` is ``false``, match only the exact metric name prefix.

#

# If ``[metrics] metrics\_use\_pattern\_match`` is ``true``, provide regex patterns to match.

#

# Example: metrics\_block\_list = "scheduler,executor,dagrun,pool,triggerer,celery" or "^scheduler,^executor,heartbeat|timeout"

#

# Variable: AIRFLOW\_\_METRICS\_\_METRICS\_BLOCK\_LIST

#

metrics\_block\_list =

# Enables sending metrics to StatsD.

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_ON

#

statsd\_on = False

# Specifies the host address where the StatsD daemon (or server) is running

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_HOST

#

statsd\_host = localhost

# Specifies the port on which the StatsD daemon (or server) is listening to

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_PORT

#

statsd\_port = 8125

# Defines the namespace for all metrics sent from Airflow to StatsD

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_PREFIX

#

statsd\_prefix = airflow

# A function that validate the StatsD stat name, apply changes to the stat name if necessary and return

# the transformed stat name.

#

# The function should have the following signature

#

# .. code-block:: python

#

# def func\_name(stat\_name: str) -> str: ...

#

# Variable: AIRFLOW\_\_METRICS\_\_STAT\_NAME\_HANDLER

#

stat\_name\_handler =

# To enable datadog integration to send airflow metrics.

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_DATADOG\_ENABLED

#

statsd\_datadog\_enabled = False

# List of datadog tags attached to all metrics(e.g: ``key1:value1,key2:value2``)

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_DATADOG\_TAGS

#

statsd\_datadog\_tags =

# Set to ``False`` to disable metadata tags for some of the emitted metrics

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_DATADOG\_METRICS\_TAGS

#

statsd\_datadog\_metrics\_tags = True

# If you want to utilise your own custom StatsD client set the relevant

# module path below.

# Note: The module path must exist on your

# `PYTHONPATH <https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH>`

# for Airflow to pick it up

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_CUSTOM\_CLIENT\_PATH

#

# statsd\_custom\_client\_path =

# If you want to avoid sending all the available metrics tags to StatsD,

# you can configure a block list of prefixes (comma separated) to filter out metric tags

# that start with the elements of the list (e.g: ``job\_id,run\_id``)

#

# Example: statsd\_disabled\_tags = job\_id,run\_id,dag\_id,task\_id

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_DISABLED\_TAGS

#

statsd\_disabled\_tags = job\_id,run\_id

# To enable sending Airflow metrics with StatsD-Influxdb tagging convention.

#

# Variable: AIRFLOW\_\_METRICS\_\_STATSD\_INFLUXDB\_ENABLED

#

statsd\_influxdb\_enabled = False

# Enables sending metrics to OpenTelemetry.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_ON

#

otel\_on = False

# Specifies the hostname or IP address of the OpenTelemetry Collector to which Airflow sends

# metrics and traces.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_HOST

#

otel\_host = localhost

# Specifies the port of the OpenTelemetry Collector that is listening to.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_PORT

#

otel\_port = 8889

# The prefix for the Airflow metrics.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_PREFIX

#

otel\_prefix = airflow

# Defines the interval, in milliseconds, at which Airflow sends batches of metrics and traces

# to the configured OpenTelemetry Collector.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_INTERVAL\_MILLISECONDS

#

otel\_interval\_milliseconds = 60000

# If ``True``, all metrics are also emitted to the console. Defaults to ``False``.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_DEBUGGING\_ON

#

otel\_debugging\_on = False

# The default service name of traces.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_SERVICE

#

otel\_service = Airflow

# If ``True``, SSL will be enabled. Defaults to ``False``.

# To establish an HTTPS connection to the OpenTelemetry collector,

# you need to configure the SSL certificate and key within the OpenTelemetry collector's

# ``config.yml`` file.

#

# Variable: AIRFLOW\_\_METRICS\_\_OTEL\_SSL\_ACTIVE

#

otel\_ssl\_active = False

[traces]

# Distributed traces integration settings.

# Enables sending traces to OpenTelemetry.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_ON

#

otel\_on = False

# Specifies the hostname or IP address of the OpenTelemetry Collector to which Airflow sends

# traces.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_HOST

#

otel\_host = localhost

# Specifies the port of the OpenTelemetry Collector that is listening to.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_PORT

#

otel\_port = 8889

# The default service name of traces.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_SERVICE

#

otel\_service = Airflow

# If True, all traces are also emitted to the console. Defaults to False.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_DEBUGGING\_ON

#

otel\_debugging\_on = False

# If True, SSL will be enabled. Defaults to False.

# To establish an HTTPS connection to the OpenTelemetry collector,

# you need to configure the SSL certificate and key within the OpenTelemetry collector's

# config.yml file.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_SSL\_ACTIVE

#

otel\_ssl\_active = False

# If True, after the task is complete, the full task log messages will be added as the

# span events, chunked by 64k size. defaults to False.

#

# Variable: AIRFLOW\_\_TRACES\_\_OTEL\_TASK\_LOG\_EVENT

#

otel\_task\_log\_event = False

[secrets]

# Full class name of secrets backend to enable (will precede env vars and metastore in search path)

#

# Example: backend = airflow.providers.amazon.aws.secrets.systems\_manager.SystemsManagerParameterStoreBackend

#

# Variable: AIRFLOW\_\_SECRETS\_\_BACKEND

#

backend =

# The backend\_kwargs param is loaded into a dictionary and passed to ``\_\_init\_\_``

# of secrets backend class. See documentation for the secrets backend you are using.

# JSON is expected.

#

# Example for AWS Systems Manager ParameterStore:

# ``{"connections\_prefix": "/airflow/connections", "profile\_name": "default"}``

#

# Variable: AIRFLOW\_\_SECRETS\_\_BACKEND\_KWARGS

#

backend\_kwargs =

# .. note:: |experimental|

#

# Enables local caching of Variables, when parsing DAGs only.

# Using this option can make dag parsing faster if Variables are used in top level code, at the expense

# of longer propagation time for changes.

# Please note that this cache concerns only the DAG parsing step. There is no caching in place when DAG

# tasks are run.

#

# Variable: AIRFLOW\_\_SECRETS\_\_USE\_CACHE

#

use\_cache = False

# .. note:: |experimental|

#

# When the cache is enabled, this is the duration for which we consider an entry in the cache to be

# valid. Entries are refreshed if they are older than this many seconds.

# It means that when the cache is enabled, this is the maximum amount of time you need to wait to see a

# Variable change take effect.

#

# Variable: AIRFLOW\_\_SECRETS\_\_CACHE\_TTL\_SECONDS

#

cache\_ttl\_seconds = 900

[cli]

# In what way should the cli access the API. The LocalClient will use the

# database directly, while the json\_client will use the api running on the

# webserver

#

# Variable: AIRFLOW\_\_CLI\_\_API\_CLIENT

#

api\_client = airflow.api.client.local\_client

# If you set web\_server\_url\_prefix, do NOT forget to append it here, ex:

# ``endpoint\_url = http://localhost:8080/myroot``

# So api will look like: ``http://localhost:8080/myroot/api/experimental/...``

#

# Variable: AIRFLOW\_\_CLI\_\_ENDPOINT\_URL

#

endpoint\_url = http://localhost:8080

[debug]

# Used only with ``DebugExecutor``. If set to ``True`` DAG will fail with first

# failed task. Helpful for debugging purposes.

#

# Variable: AIRFLOW\_\_DEBUG\_\_FAIL\_FAST

#

fail\_fast = False

[api]

# Enables the deprecated experimental API. Please note that these API endpoints do not have

# access control. An authenticated user has full access.

#

# .. warning::

#

# This `Experimental REST API

# <https://airflow.apache.org/docs/apache-airflow/stable/deprecated-rest-api-ref.html>`\_\_ is

# deprecated since version 2.0. Please consider using

# `the Stable REST API

# <https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html>`\_\_.

# For more information on migration, see

# `RELEASE\_NOTES.rst <https://github.com/apache/airflow/blob/main/RELEASE\_NOTES.rst>`\_

#

# Variable: AIRFLOW\_\_API\_\_ENABLE\_EXPERIMENTAL\_API

#

enable\_experimental\_api = False

# Comma separated list of auth backends to authenticate users of the API. See

# `Security: API

# <https://airflow.apache.org/docs/apache-airflow/stable/security/api.html>`\_\_ for possible values.

# ("airflow.api.auth.backend.default" allows all requests for historic reasons)

#

# Variable: AIRFLOW\_\_API\_\_AUTH\_BACKENDS

#

auth\_backends = airflow.api.auth.backend.session

# Used to set the maximum page limit for API requests. If limit passed as param

# is greater than maximum page limit, it will be ignored and maximum page limit value

# will be set as the limit

#

# Variable: AIRFLOW\_\_API\_\_MAXIMUM\_PAGE\_LIMIT

#

maximum\_page\_limit = 100

# Used to set the default page limit when limit param is zero or not provided in API

# requests. Otherwise if positive integer is passed in the API requests as limit, the

# smallest number of user given limit or maximum page limit is taken as limit.

#

# Variable: AIRFLOW\_\_API\_\_FALLBACK\_PAGE\_LIMIT

#

fallback\_page\_limit = 100

# The intended audience for JWT token credentials used for authorization. This value must match on the client and server sides. If empty, audience will not be tested.

#

# Example: google\_oauth2\_audience = project-id-random-value.apps.googleusercontent.com

#

# Variable: AIRFLOW\_\_API\_\_GOOGLE\_OAUTH2\_AUDIENCE

#

google\_oauth2\_audience =

# Path to Google Cloud Service Account key file (JSON). If omitted, authorization based on

# `the Application Default Credentials

# <https://cloud.google.com/docs/authentication/production#finding\_credentials\_automatically>`\_\_ will

# be used.

#

# Example: google\_key\_path = /files/service-account-json

#

# Variable: AIRFLOW\_\_API\_\_GOOGLE\_KEY\_PATH

#

google\_key\_path =

# Used in response to a preflight request to indicate which HTTP

# headers can be used when making the actual request. This header is

# the server side response to the browser's

# Access-Control-Request-Headers header.

#

# Variable: AIRFLOW\_\_API\_\_ACCESS\_CONTROL\_ALLOW\_HEADERS

#

access\_control\_allow\_headers =

# Specifies the method or methods allowed when accessing the resource.

#

# Variable: AIRFLOW\_\_API\_\_ACCESS\_CONTROL\_ALLOW\_METHODS

#

access\_control\_allow\_methods =

# Indicates whether the response can be shared with requesting code from the given origins.

# Separate URLs with space.

#

# Variable: AIRFLOW\_\_API\_\_ACCESS\_CONTROL\_ALLOW\_ORIGINS

#

access\_control\_allow\_origins =

# Indicates whether the \*\*xcomEntries\*\* endpoint supports the \*\*deserialize\*\*

# flag. If set to ``False``, setting this flag in a request would result in a

# 400 Bad Request error.

#

# Variable: AIRFLOW\_\_API\_\_ENABLE\_XCOM\_DESERIALIZE\_SUPPORT

#

enable\_xcom\_deserialize\_support = False

[lineage]

# what lineage backend to use

#

# Variable: AIRFLOW\_\_LINEAGE\_\_BACKEND

#

backend =

[operators]

# The default owner assigned to each new operator, unless

# provided explicitly or passed via ``default\_args``

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_OWNER

#

default\_owner = airflow

# The default value of attribute "deferrable" in operators and sensors.

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_DEFERRABLE

#

default\_deferrable = false

# Indicates the default number of CPU units allocated to each operator when no specific CPU request

# is specified in the operator's configuration

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_CPUS

#

default\_cpus = 1

# Indicates the default number of RAM allocated to each operator when no specific RAM request

# is specified in the operator's configuration

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_RAM

#

default\_ram = 512

# Indicates the default number of disk storage allocated to each operator when no specific disk request

# is specified in the operator's configuration

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_DISK

#

default\_disk = 512

# Indicates the default number of GPUs allocated to each operator when no specific GPUs request

# is specified in the operator's configuration

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_GPUS

#

default\_gpus = 0

# Default queue that tasks get assigned to and that worker listen on.

#

# Variable: AIRFLOW\_\_OPERATORS\_\_DEFAULT\_QUEUE

#

default\_queue = default

# Is allowed to pass additional/unused arguments (args, kwargs) to the BaseOperator operator.

# If set to ``False``, an exception will be thrown,

# otherwise only the console message will be displayed.

#

# Variable: AIRFLOW\_\_OPERATORS\_\_ALLOW\_ILLEGAL\_ARGUMENTS

#

allow\_illegal\_arguments = False

[webserver]

# The message displayed when a user attempts to execute actions beyond their authorised privileges.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ACCESS\_DENIED\_MESSAGE

#

access\_denied\_message = Access is Denied

# Path of webserver config file used for configuring the webserver parameters

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_CONFIG\_FILE

#

config\_file = /home/airflow/airflow/webserver\_config.py

# The base url of your website: Airflow cannot guess what domain or CNAME you are using.

# This is used to create links in the Log Url column in the Browse - Task Instances menu,

# as well as in any automated emails sent by Airflow that contain links to your webserver.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_BASE\_URL

#

base\_url = http://localhost:8080

# Default timezone to display all dates in the UI, can be UTC, system, or

# any IANA timezone string (e.g. \*\*Europe/Amsterdam\*\*). If left empty the

# default value of core/default\_timezone will be used

#

# Example: default\_ui\_timezone = America/New\_York

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_DEFAULT\_UI\_TIMEZONE

#

default\_ui\_timezone = UTC

# The ip specified when starting the web server

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_HOST

#

web\_server\_host = 0.0.0.0

# The port on which to run the web server

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_PORT

#

web\_server\_port = 8080

# Paths to the SSL certificate and key for the web server. When both are

# provided SSL will be enabled. This does not change the web server port.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_SSL\_CERT

#

web\_server\_ssl\_cert =

# Paths to the SSL certificate and key for the web server. When both are

# provided SSL will be enabled. This does not change the web server port.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_SSL\_KEY

#

web\_server\_ssl\_key =

# The type of backend used to store web session data, can be ``database`` or ``securecookie``. For the

# ``database`` backend, sessions are store in the database and they can be

# managed there (for example when you reset password of the user, all sessions for that user are

# deleted). For the ``securecookie`` backend, sessions are stored in encrypted cookies on the client

# side. The ``securecookie`` mechanism is 'lighter' than database backend, but sessions are not deleted

# when you reset password of the user, which means that other than waiting for expiry time, the only

# way to invalidate all sessions for a user is to change secret\_key and restart webserver (which

# also invalidates and logs out all other user's sessions).

#

# When you are using ``database`` backend, make sure to keep your database session table small

# by periodically running ``airflow db clean --table session`` command, especially if you have

# automated API calls that will create a new session for each call rather than reuse the sessions

# stored in browser cookies.

#

# Example: session\_backend = securecookie

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_SESSION\_BACKEND

#

session\_backend = database

# Number of seconds the webserver waits before killing gunicorn master that doesn't respond

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_MASTER\_TIMEOUT

#

web\_server\_master\_timeout = 120

# Number of seconds the gunicorn webserver waits before timing out on a worker

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_WORKER\_TIMEOUT

#

web\_server\_worker\_timeout = 120

# Number of workers to refresh at a time. When set to 0, worker refresh is

# disabled. When nonzero, airflow periodically refreshes webserver workers by

# bringing up new ones and killing old ones.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WORKER\_REFRESH\_BATCH\_SIZE

#

worker\_refresh\_batch\_size = 1

# Number of seconds to wait before refreshing a batch of workers.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WORKER\_REFRESH\_INTERVAL

#

worker\_refresh\_interval = 6000

# If set to ``True``, Airflow will track files in plugins\_folder directory. When it detects changes,

# then reload the gunicorn. If set to ``True``, gunicorn starts without preloading, which is slower,

# uses more memory, and may cause race conditions. Avoid setting this to ``True`` in production.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_RELOAD\_ON\_PLUGIN\_CHANGE

#

reload\_on\_plugin\_change = False

# Secret key used to run your flask app. It should be as random as possible. However, when running

# more than 1 instances of webserver, make sure all of them use the same ``secret\_key`` otherwise

# one of them will error with "CSRF session token is missing".

# The webserver key is also used to authorize requests to Celery workers when logs are retrieved.

# The token generated using the secret key has a short expiry time though - make sure that time on

# ALL the machines that you run airflow components on is synchronized (for example using ntpd)

# otherwise you might get "forbidden" errors when the logs are accessed.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_SECRET\_KEY

#

secret\_key = 3dDCLAL4X63Zd1a//soPMQ==

# Number of workers to run the Gunicorn web server

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WORKERS

#

workers = 4

# The worker class gunicorn should use. Choices include

# ``sync`` (default), ``eventlet``, ``gevent``.

#

# .. warning::

#

# When using ``gevent`` you might also want to set the ``\_AIRFLOW\_PATCH\_GEVENT``

# environment variable to ``"1"`` to make sure gevent patching is done as early as possible.

#

# Be careful to set ``\_AIRFLOW\_PATCH\_GEVENT`` only on the web server as gevent patching may

# affect the scheduler behavior via the ``multiprocessing`` sockets module and cause crash.

#

# See related Issues / PRs for more details:

#

# \* https://github.com/benoitc/gunicorn/issues/2796

# \* https://github.com/apache/airflow/issues/8212

# \* https://github.com/apache/airflow/pull/28283

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WORKER\_CLASS

#

worker\_class = sync

# Log files for the gunicorn webserver. '-' means log to stderr.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ACCESS\_LOGFILE

#

access\_logfile = -

# Log files for the gunicorn webserver. '-' means log to stderr.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ERROR\_LOGFILE

#

error\_logfile = -

# Access log format for gunicorn webserver.

# default format is ``%%(h)s %%(l)s %%(u)s %%(t)s "%%(r)s" %%(s)s %%(b)s "%%(f)s" "%%(a)s"``

# See `Gunicorn Settings: 'access\_log\_format' Reference

# <https://docs.gunicorn.org/en/stable/settings.html#access-log-format>`\_\_ for more details

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ACCESS\_LOGFORMAT

#

access\_logformat =

# Expose the configuration file in the web server. Set to ``non-sensitive-only`` to show all values

# except those that have security implications. ``True`` shows all values. ``False`` hides the

# configuration completely.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_EXPOSE\_CONFIG

#

expose\_config = False

# Expose hostname in the web server

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_EXPOSE\_HOSTNAME

#

expose\_hostname = False

# Expose stacktrace in the web server

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_EXPOSE\_STACKTRACE

#

expose\_stacktrace = False

# Default DAG view. Valid values are: ``grid``, ``graph``, ``duration``, ``gantt``, ``landing\_times``

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_DAG\_DEFAULT\_VIEW

#

dag\_default\_view = grid

# Default DAG orientation. Valid values are:

# ``LR`` (Left->Right), ``TB`` (Top->Bottom), ``RL`` (Right->Left), ``BT`` (Bottom->Top)

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_DAG\_ORIENTATION

#

dag\_orientation = LR

# Sorting order in grid view. Valid values are: ``topological``, ``hierarchical\_alphabetical``

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_GRID\_VIEW\_SORTING\_ORDER

#

grid\_view\_sorting\_order = topological

# The amount of time (in secs) webserver will wait for initial handshake

# while fetching logs from other worker machine

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_LOG\_FETCH\_TIMEOUT\_SEC

#

log\_fetch\_timeout\_sec = 5

# Time interval (in secs) to wait before next log fetching.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_LOG\_FETCH\_DELAY\_SEC

#

log\_fetch\_delay\_sec = 2

# Distance away from page bottom to enable auto tailing.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_LOG\_AUTO\_TAILING\_OFFSET

#

log\_auto\_tailing\_offset = 30

# Animation speed for auto tailing log display.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_LOG\_ANIMATION\_SPEED

#

log\_animation\_speed = 1000

# By default, the webserver shows paused DAGs. Flip this to hide paused

# DAGs by default

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_HIDE\_PAUSED\_DAGS\_BY\_DEFAULT

#

hide\_paused\_dags\_by\_default = False

# Consistent page size across all listing views in the UI

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PAGE\_SIZE

#

page\_size = 100

# Define the color of navigation bar

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NAVBAR\_COLOR

#

navbar\_color = #fff

# Define the color of text in the navigation bar

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NAVBAR\_TEXT\_COLOR

#

navbar\_text\_color = #51504f

# Define the color of navigation bar links when hovered

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NAVBAR\_HOVER\_COLOR

#

navbar\_hover\_color = #eee

# Define the color of text in the navigation bar when hovered

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NAVBAR\_TEXT\_HOVER\_COLOR

#

navbar\_text\_hover\_color = #51504f

# Define the color of the logo text

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NAVBAR\_LOGO\_TEXT\_COLOR

#

navbar\_logo\_text\_color = #51504f

# Default dagrun to show in UI

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_DEFAULT\_DAG\_RUN\_DISPLAY\_NUMBER

#

default\_dag\_run\_display\_number = 25

# Enable werkzeug ``ProxyFix`` middleware for reverse proxy

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ENABLE\_PROXY\_FIX

#

enable\_proxy\_fix = False

# Number of values to trust for ``X-Forwarded-For``.

# See `Werkzeug: X-Forwarded-For Proxy Fix

# <https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\_fix/>`\_\_ for more details.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PROXY\_FIX\_X\_FOR

#

proxy\_fix\_x\_for = 1

# Number of values to trust for ``X-Forwarded-Proto``.

# See `Werkzeug: X-Forwarded-For Proxy Fix

# <https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\_fix/>`\_\_ for more details.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PROXY\_FIX\_X\_PROTO

#

proxy\_fix\_x\_proto = 1

# Number of values to trust for ``X-Forwarded-Host``.

# See `Werkzeug: X-Forwarded-For Proxy Fix

# <https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\_fix/>`\_\_ for more details.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PROXY\_FIX\_X\_HOST

#

proxy\_fix\_x\_host = 1

# Number of values to trust for ``X-Forwarded-Port``.

# See `Werkzeug: X-Forwarded-For Proxy Fix

# <https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\_fix/>`\_\_ for more details.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PROXY\_FIX\_X\_PORT

#

proxy\_fix\_x\_port = 1

# Number of values to trust for ``X-Forwarded-Prefix``.

# See `Werkzeug: X-Forwarded-For Proxy Fix

# <https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\_fix/>`\_\_ for more details.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_PROXY\_FIX\_X\_PREFIX

#

proxy\_fix\_x\_prefix = 1

# Set secure flag on session cookie

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_COOKIE\_SECURE

#

cookie\_secure = False

# Set samesite policy on session cookie

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_COOKIE\_SAMESITE

#

cookie\_samesite = Lax

# Default setting for wrap toggle on DAG code and TI log views.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_DEFAULT\_WRAP

#

default\_wrap = False

# Allow the UI to be rendered in a frame

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_X\_FRAME\_ENABLED

#

x\_frame\_enabled = True

# Send anonymous user activity to your analytics tool

# choose from ``google\_analytics``, ``segment``, ``metarouter``, or ``matomo``

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ANALYTICS\_TOOL

#

# analytics\_tool =

# Unique ID of your account in the analytics tool

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ANALYTICS\_ID

#

# analytics\_id =

# Your instances url, only applicable to Matomo.

#

# Example: analytics\_url = https://your.matomo.instance.com/

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ANALYTICS\_URL

#

# analytics\_url =

# 'Recent Tasks' stats will show for old DagRuns if set

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_SHOW\_RECENT\_STATS\_FOR\_COMPLETED\_RUNS

#

show\_recent\_stats\_for\_completed\_runs = True

# The UI cookie lifetime in minutes. User will be logged out from UI after

# ``[webserver] session\_lifetime\_minutes`` of non-activity

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_SESSION\_LIFETIME\_MINUTES

#

session\_lifetime\_minutes = 43200

# Sets a custom page title for the DAGs overview page and site title for all pages

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_INSTANCE\_NAME

#

# instance\_name =

# Whether the custom page title for the DAGs overview page contains any Markup language

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_INSTANCE\_NAME\_HAS\_MARKUP

#

instance\_name\_has\_markup = False

# How frequently, in seconds, the DAG data will auto-refresh in graph or grid view

# when auto-refresh is turned on

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_AUTO\_REFRESH\_INTERVAL

#

auto\_refresh\_interval = 3

# Boolean for displaying warning for publicly viewable deployment

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_WARN\_DEPLOYMENT\_EXPOSURE

#

warn\_deployment\_exposure = True

# Comma separated string of view events to exclude from dag audit view.

# All other events will be added minus the ones passed here.

# The audit logs in the db will not be affected by this parameter.

#

# Example: audit\_view\_excluded\_events = cli\_task\_run,running,success

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_AUDIT\_VIEW\_EXCLUDED\_EVENTS

#

# audit\_view\_excluded\_events =

# Comma separated string of view events to include in dag audit view.

# If passed, only these events will populate the dag audit view.

# The audit logs in the db will not be affected by this parameter.

#

# Example: audit\_view\_included\_events = dagrun\_cleared,failed

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_AUDIT\_VIEW\_INCLUDED\_EVENTS

#

# audit\_view\_included\_events =

# Boolean for running SwaggerUI in the webserver.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ENABLE\_SWAGGER\_UI

#

enable\_swagger\_ui = True

# Boolean for running Internal API in the webserver.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_RUN\_INTERNAL\_API

#

run\_internal\_api = False

# The caching algorithm used by the webserver. Must be a valid hashlib function name.

#

# Example: caching\_hash\_method = sha256

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_CACHING\_HASH\_METHOD

#

caching\_hash\_method = md5

# Behavior of the trigger DAG run button for DAGs without params. ``False`` to skip and trigger

# without displaying a form to add a \*\*dag\_run.conf\*\*, ``True`` to always display the form.

# The form is displayed always if parameters are defined.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_SHOW\_TRIGGER\_FORM\_IF\_NO\_PARAMS

#

show\_trigger\_form\_if\_no\_params = False

# Number of recent DAG run configurations in the selector on the trigger web form.

#

# Example: num\_recent\_configurations\_for\_trigger = 10

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_NUM\_RECENT\_CONFIGURATIONS\_FOR\_TRIGGER

#

num\_recent\_configurations\_for\_trigger = 5

# A DAG author is able to provide any raw HTML into ``doc\_md`` or params description in

# ``description\_md`` for text formatting. This is including potentially unsafe javascript.

# Displaying the DAG or trigger form in web UI provides the DAG author the potential to

# inject malicious code into clients browsers. To ensure the web UI is safe by default,

# raw HTML is disabled by default. If you trust your DAG authors, you can enable HTML

# support in markdown by setting this option to ``True``.

#

# This parameter also enables the deprecated fields ``description\_html`` and

# ``custom\_html\_form`` in DAG params until the feature is removed in a future version.

#

# Example: allow\_raw\_html\_descriptions = False

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ALLOW\_RAW\_HTML\_DESCRIPTIONS

#

allow\_raw\_html\_descriptions = False

# The maximum size of the request payload (in MB) that can be sent.

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_ALLOWED\_PAYLOAD\_SIZE

#

allowed\_payload\_size = 1.0

# Require confirmation when changing a DAG in the web UI. This is to prevent accidental changes

# to a DAG that may be running on sensitive environments like production.

# When set to ``True``, confirmation dialog will be shown when a user tries to Pause/Unpause,

# Trigger a DAG

#

# Variable: AIRFLOW\_\_WEBSERVER\_\_REQUIRE\_CONFIRMATION\_DAG\_CHANGE

#

require\_confirmation\_dag\_change = False

[email]

# Configuration email backend and whether to

# send email alerts on retry or failure

# Email backend to use

#

# Variable: AIRFLOW\_\_EMAIL\_\_EMAIL\_BACKEND

#

email\_backend = airflow.utils.email.send\_email\_smtp

# Email connection to use

#

# Variable: AIRFLOW\_\_EMAIL\_\_EMAIL\_CONN\_ID

#

email\_conn\_id = smtp\_default

# Whether email alerts should be sent when a task is retried

#

# Variable: AIRFLOW\_\_EMAIL\_\_DEFAULT\_EMAIL\_ON\_RETRY

#

default\_email\_on\_retry = True

# Whether email alerts should be sent when a task failed

#

# Variable: AIRFLOW\_\_EMAIL\_\_DEFAULT\_EMAIL\_ON\_FAILURE

#

default\_email\_on\_failure = True

# File that will be used as the template for Email subject (which will be rendered using Jinja2).

# If not set, Airflow uses a base template.

#

# Example: subject\_template = /path/to/my\_subject\_template\_file

#

# Variable: AIRFLOW\_\_EMAIL\_\_SUBJECT\_TEMPLATE

#

# subject\_template =

# File that will be used as the template for Email content (which will be rendered using Jinja2).

# If not set, Airflow uses a base template.

#

# Example: html\_content\_template = /path/to/my\_html\_content\_template\_file

#

# Variable: AIRFLOW\_\_EMAIL\_\_HTML\_CONTENT\_TEMPLATE

#

# html\_content\_template =

# Email address that will be used as sender address.

# It can either be raw email or the complete address in a format ``Sender Name <sender@email.com>``

#

# Example: from\_email = Airflow <airflow@example.com>

#

# Variable: AIRFLOW\_\_EMAIL\_\_FROM\_EMAIL

#

# from\_email =

# ssl context to use when using SMTP and IMAP SSL connections. By default, the context is "default"

# which sets it to ``ssl.create\_default\_context()`` which provides the right balance between

# compatibility and security, it however requires that certificates in your operating system are

# updated and that SMTP/IMAP servers of yours have valid certificates that have corresponding public

# keys installed on your machines. You can switch it to "none" if you want to disable checking

# of the certificates, but it is not recommended as it allows MITM (man-in-the-middle) attacks

# if your infrastructure is not sufficiently secured. It should only be set temporarily while you

# are fixing your certificate configuration. This can be typically done by upgrading to newer

# version of the operating system you run Airflow components on,by upgrading/refreshing proper

# certificates in the OS or by updating certificates for your mail servers.

#

# Example: ssl\_context = default

#

# Variable: AIRFLOW\_\_EMAIL\_\_SSL\_CONTEXT

#

ssl\_context = default

[smtp]

# If you want airflow to send emails on retries, failure, and you want to use

# the airflow.utils.email.send\_email\_smtp function, you have to configure an

# smtp server here

# Specifies the host server address used by Airflow when sending out email notifications via SMTP.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_HOST

#

smtp\_host = localhost

# Determines whether to use the STARTTLS command when connecting to the SMTP server.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_STARTTLS

#

smtp\_starttls = True

# Determines whether to use an SSL connection when talking to the SMTP server.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_SSL

#

smtp\_ssl = False

# Username to authenticate when connecting to smtp server.

#

# Example: smtp\_user = airflow

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_USER

#

# smtp\_user =

# Password to authenticate when connecting to smtp server.

#

# Example: smtp\_password = airflow

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_PASSWORD

#

# smtp\_password =

# Defines the port number on which Airflow connects to the SMTP server to send email notifications.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_PORT

#

smtp\_port = 25

# Specifies the default \*\*from\*\* email address used when Airflow sends email notifications.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_MAIL\_FROM

#

smtp\_mail\_from = airflow@example.com

# Determines the maximum time (in seconds) the Apache Airflow system will wait for a

# connection to the SMTP server to be established.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_TIMEOUT

#

smtp\_timeout = 30

# Defines the maximum number of times Airflow will attempt to connect to the SMTP server.

#

# Variable: AIRFLOW\_\_SMTP\_\_SMTP\_RETRY\_LIMIT

#

smtp\_retry\_limit = 5

[sentry]

# `Sentry <https://docs.sentry.io>`\_\_ integration. Here you can supply

# additional configuration options based on the Python platform.

# See `Python / Configuration / Basic Options

# <https://docs.sentry.io/platforms/python/configuration/options/>`\_\_ for more details.

# Unsupported options: ``integrations``, ``in\_app\_include``, ``in\_app\_exclude``,

# ``ignore\_errors``, ``before\_breadcrumb``, ``transport``.

# Enable error reporting to Sentry

#

# Variable: AIRFLOW\_\_SENTRY\_\_SENTRY\_ON

#

sentry\_on = false

#

# Variable: AIRFLOW\_\_SENTRY\_\_SENTRY\_DSN

#

sentry\_dsn =

# Dotted path to a before\_send function that the sentry SDK should be configured to use.

#

# Variable: AIRFLOW\_\_SENTRY\_\_BEFORE\_SEND

#

# before\_send =

[scheduler]

# Task instances listen for external kill signal (when you clear tasks

# from the CLI or the UI), this defines the frequency at which they should

# listen (in seconds).

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_JOB\_HEARTBEAT\_SEC

#

job\_heartbeat\_sec = 5

# The scheduler constantly tries to trigger new tasks (look at the

# scheduler section in the docs for more information). This defines

# how often the scheduler should run (in seconds).

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_HEARTBEAT\_SEC

#

scheduler\_heartbeat\_sec = 5

# The frequency (in seconds) at which the LocalTaskJob should send heartbeat signals to the

# scheduler to notify it's still alive. If this value is set to 0, the heartbeat interval will default

# to the value of ``[scheduler] scheduler\_zombie\_task\_threshold``.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_LOCAL\_TASK\_JOB\_HEARTBEAT\_SEC

#

local\_task\_job\_heartbeat\_sec = 0

# The number of times to try to schedule each DAG file

# -1 indicates unlimited number

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_NUM\_RUNS

#

num\_runs = -1

# Controls how long the scheduler will sleep between loops, but if there was nothing to do

# in the loop. i.e. if it scheduled something then it will start the next loop

# iteration straight away.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_IDLE\_SLEEP\_TIME

#

scheduler\_idle\_sleep\_time = 1

# Number of seconds after which a DAG file is parsed. The DAG file is parsed every

# ``[scheduler] min\_file\_process\_interval`` number of seconds. Updates to DAGs are reflected after

# this interval. Keeping this number low will increase CPU usage.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_MIN\_FILE\_PROCESS\_INTERVAL

#

min\_file\_process\_interval = 30

# How often (in seconds) to check for stale DAGs (DAGs which are no longer present in

# the expected files) which should be deactivated, as well as datasets that are no longer

# referenced and should be marked as orphaned.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_PARSING\_CLEANUP\_INTERVAL

#

parsing\_cleanup\_interval = 60

# How long (in seconds) to wait after we have re-parsed a DAG file before deactivating stale

# DAGs (DAGs which are no longer present in the expected files). The reason why we need

# this threshold is to account for the time between when the file is parsed and when the

# DAG is loaded. The absolute maximum that this could take is ``[core] dag\_file\_processor\_timeout``,

# but when you have a long timeout configured, it results in a significant delay in the

# deactivation of stale dags.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_STALE\_DAG\_THRESHOLD

#

stale\_dag\_threshold = 50

# How often (in seconds) to scan the DAGs directory for new files. Default to 5 minutes.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_DAG\_DIR\_LIST\_INTERVAL

#

dag\_dir\_list\_interval = 300

# How often should stats be printed to the logs. Setting to 0 will disable printing stats

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_PRINT\_STATS\_INTERVAL

#

print\_stats\_interval = 30

# How often (in seconds) should pool usage stats be sent to StatsD (if statsd\_on is enabled)

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_POOL\_METRICS\_INTERVAL

#

pool\_metrics\_interval = 5.0

# If the last scheduler heartbeat happened more than ``[scheduler] scheduler\_health\_check\_threshold``

# ago (in seconds), scheduler is considered unhealthy.

# This is used by the health check in the \*\*/health\*\* endpoint and in ``airflow jobs check`` CLI

# for SchedulerJob.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_HEALTH\_CHECK\_THRESHOLD

#

scheduler\_health\_check\_threshold = 30

# When you start a scheduler, airflow starts a tiny web server

# subprocess to serve a health check if this is set to ``True``

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_ENABLE\_HEALTH\_CHECK

#

enable\_health\_check = False

# When you start a scheduler, airflow starts a tiny web server

# subprocess to serve a health check on this host

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_HEALTH\_CHECK\_SERVER\_HOST

#

scheduler\_health\_check\_server\_host = 0.0.0.0

# When you start a scheduler, airflow starts a tiny web server

# subprocess to serve a health check on this port

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_HEALTH\_CHECK\_SERVER\_PORT

#

scheduler\_health\_check\_server\_port = 8974

# How often (in seconds) should the scheduler check for orphaned tasks and SchedulerJobs

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_ORPHANED\_TASKS\_CHECK\_INTERVAL

#

orphaned\_tasks\_check\_interval = 300.0

# Determines the directory where logs for the child processes of the scheduler will be stored

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_CHILD\_PROCESS\_LOG\_DIRECTORY

#

child\_process\_log\_directory = /home/airflow/airflow/logs/scheduler

# Local task jobs periodically heartbeat to the DB. If the job has

# not heartbeat in this many seconds, the scheduler will mark the

# associated task instance as failed and will re-schedule the task.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULER\_ZOMBIE\_TASK\_THRESHOLD

#

scheduler\_zombie\_task\_threshold = 300

# How often (in seconds) should the scheduler check for zombie tasks.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_ZOMBIE\_DETECTION\_INTERVAL

#

zombie\_detection\_interval = 10.0

# Turn off scheduler catchup by setting this to ``False``.

# Default behavior is unchanged and

# Command Line Backfills still work, but the scheduler

# will not do scheduler catchup if this is ``False``,

# however it can be set on a per DAG basis in the

# DAG definition (catchup)

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_CATCHUP\_BY\_DEFAULT

#

catchup\_by\_default = True

# Setting this to ``True`` will make first task instance of a task

# ignore depends\_on\_past setting. A task instance will be considered

# as the first task instance of a task when there is no task instance

# in the DB with an execution\_date earlier than it., i.e. no manual marking

# success will be needed for a newly added task to be scheduled.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_IGNORE\_FIRST\_DEPENDS\_ON\_PAST\_BY\_DEFAULT

#

ignore\_first\_depends\_on\_past\_by\_default = True

# This changes the batch size of queries in the scheduling main loop.

# This should not be greater than ``[core] parallelism``.

# If this is too high, SQL query performance may be impacted by

# complexity of query predicate, and/or excessive locking.

# Additionally, you may hit the maximum allowable query length for your db.

# Set this to 0 to use the value of ``[core] parallelism``

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_MAX\_TIS\_PER\_QUERY

#

max\_tis\_per\_query = 16

# Should the scheduler issue ``SELECT ... FOR UPDATE`` in relevant queries.

# If this is set to ``False`` then you should not run more than a single

# scheduler at once

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_USE\_ROW\_LEVEL\_LOCKING

#

use\_row\_level\_locking = True

# Max number of DAGs to create DagRuns for per scheduler loop.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_MAX\_DAGRUNS\_TO\_CREATE\_PER\_LOOP

#

max\_dagruns\_to\_create\_per\_loop = 10

# How many DagRuns should a scheduler examine (and lock) when scheduling

# and queuing tasks.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_MAX\_DAGRUNS\_PER\_LOOP\_TO\_SCHEDULE

#

max\_dagruns\_per\_loop\_to\_schedule = 20

# Should the Task supervisor process perform a "mini scheduler" to attempt to schedule more tasks of the

# same DAG. Leaving this on will mean tasks in the same DAG execute quicker, but might starve out other

# dags in some circumstances

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_SCHEDULE\_AFTER\_TASK\_EXECUTION

#

schedule\_after\_task\_execution = True

# The scheduler reads dag files to extract the airflow modules that are going to be used,

# and imports them ahead of time to avoid having to re-do it for each parsing process.

# This flag can be set to ``False`` to disable this behavior in case an airflow module needs

# to be freshly imported each time (at the cost of increased DAG parsing time).

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_PARSING\_PRE\_IMPORT\_MODULES

#

parsing\_pre\_import\_modules = True

# The scheduler can run multiple processes in parallel to parse dags.

# This defines how many processes will run.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_PARSING\_PROCESSES

#

parsing\_processes = 2

# One of ``modified\_time``, ``random\_seeded\_by\_host`` and ``alphabetical``.

# The scheduler will list and sort the dag files to decide the parsing order.

#

# \* ``modified\_time``: Sort by modified time of the files. This is useful on large scale to parse the

# recently modified DAGs first.

# \* ``random\_seeded\_by\_host``: Sort randomly across multiple Schedulers but with same order on the

# same host. This is useful when running with Scheduler in HA mode where each scheduler can

# parse different DAG files.

# \* ``alphabetical``: Sort by filename

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_FILE\_PARSING\_SORT\_MODE

#

file\_parsing\_sort\_mode = modified\_time

# Whether the dag processor is running as a standalone process or it is a subprocess of a scheduler

# job.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_STANDALONE\_DAG\_PROCESSOR

#

standalone\_dag\_processor = False

# Only applicable if ``[scheduler] standalone\_dag\_processor`` is true and callbacks are stored

# in database. Contains maximum number of callbacks that are fetched during a single loop.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_MAX\_CALLBACKS\_PER\_LOOP

#

max\_callbacks\_per\_loop = 20

# Only applicable if ``[scheduler] standalone\_dag\_processor`` is true.

# Time in seconds after which dags, which were not updated by Dag Processor are deactivated.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_DAG\_STALE\_NOT\_SEEN\_DURATION

#

dag\_stale\_not\_seen\_duration = 600

# Turn off scheduler use of cron intervals by setting this to ``False``.

# DAGs submitted manually in the web UI or with trigger\_dag will still run.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_USE\_JOB\_SCHEDULE

#

use\_job\_schedule = True

# Allow externally triggered DagRuns for Execution Dates in the future

# Only has effect if schedule\_interval is set to None in DAG

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_ALLOW\_TRIGGER\_IN\_FUTURE

#

allow\_trigger\_in\_future = False

# How often to check for expired trigger requests that have not run yet.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_TRIGGER\_TIMEOUT\_CHECK\_INTERVAL

#

trigger\_timeout\_check\_interval = 15

# Amount of time a task can be in the queued state before being retried or set to failed.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_TASK\_QUEUED\_TIMEOUT

#

task\_queued\_timeout = 600.0

# How often to check for tasks that have been in the queued state for

# longer than ``[scheduler] task\_queued\_timeout``.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_TASK\_QUEUED\_TIMEOUT\_CHECK\_INTERVAL

#

task\_queued\_timeout\_check\_interval = 120.0

# The run\_id pattern used to verify the validity of user input to the run\_id parameter when

# triggering a DAG. This pattern cannot change the pattern used by scheduler to generate run\_id

# for scheduled DAG runs or DAG runs triggered without changing the run\_id parameter.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_ALLOWED\_RUN\_ID\_PATTERN

#

allowed\_run\_id\_pattern = ^[A-Za-z0-9\_.~:+-]+$

# Whether to create DAG runs that span an interval or one single point in time for cron schedules, when

# a cron string is provided to ``schedule`` argument of a DAG.

#

# \* ``True``: \*\*CronDataIntervalTimetable\*\* is used, which is suitable

# for DAGs with well-defined data interval. You get contiguous intervals from the end of the previous

# interval up to the scheduled datetime.

# \* ``False``: \*\*CronTriggerTimetable\*\* is used, which is closer to the behavior of cron itself.

#

# Notably, for \*\*CronTriggerTimetable\*\*, the logical date is the same as the time the DAG Run will

# try to schedule, while for \*\*CronDataIntervalTimetable\*\*, the logical date is the beginning of

# the data interval, but the DAG Run will try to schedule at the end of the data interval.

#

# Variable: AIRFLOW\_\_SCHEDULER\_\_CREATE\_CRON\_DATA\_INTERVALS

#

create\_cron\_data\_intervals = True

[triggerer]

# How many triggers a single Triggerer will run at once, by default.

#

# Variable: AIRFLOW\_\_TRIGGERER\_\_DEFAULT\_CAPACITY

#

default\_capacity = 1000

# How often to heartbeat the Triggerer job to ensure it hasn't been killed.

#

# Variable: AIRFLOW\_\_TRIGGERER\_\_JOB\_HEARTBEAT\_SEC

#

job\_heartbeat\_sec = 5

# If the last triggerer heartbeat happened more than ``[triggerer] triggerer\_health\_check\_threshold``

# ago (in seconds), triggerer is considered unhealthy.

# This is used by the health check in the \*\*/health\*\* endpoint and in ``airflow jobs check`` CLI

# for TriggererJob.

#

# Variable: AIRFLOW\_\_TRIGGERER\_\_TRIGGERER\_HEALTH\_CHECK\_THRESHOLD

#

triggerer\_health\_check\_threshold = 30

[kerberos]

# Location of your ccache file once kinit has been performed.

#

# Variable: AIRFLOW\_\_KERBEROS\_\_CCACHE

#

ccache = /tmp/airflow\_krb5\_ccache

# gets augmented with fqdn

#

# Variable: AIRFLOW\_\_KERBEROS\_\_PRINCIPAL

#

principal = airflow

# Determines the frequency at which initialization or re-initialization processes occur.

#

# Variable: AIRFLOW\_\_KERBEROS\_\_REINIT\_FREQUENCY

#

reinit\_frequency = 3600

# Path to the kinit executable

#

# Variable: AIRFLOW\_\_KERBEROS\_\_KINIT\_PATH

#

kinit\_path = kinit

# Designates the path to the Kerberos keytab file for the Airflow user

#

# Variable: AIRFLOW\_\_KERBEROS\_\_KEYTAB

#

keytab = airflow.keytab

# Allow to disable ticket forwardability.

#

# Variable: AIRFLOW\_\_KERBEROS\_\_FORWARDABLE

#

forwardable = True

# Allow to remove source IP from token, useful when using token behind NATted Docker host.

#

# Variable: AIRFLOW\_\_KERBEROS\_\_INCLUDE\_IP

#

include\_ip = True

[sensors]

# Sensor default timeout, 7 days by default (7 \* 24 \* 60 \* 60).

#

# Variable: AIRFLOW\_\_SENSORS\_\_DEFAULT\_TIMEOUT

#

default\_timeout = 604800

[usage\_data\_collection]

# Airflow integrates `Scarf <https://about.scarf.sh/>`\_\_ to collect basic platform and usage data

# during operation. This data assists Airflow maintainers in better understanding how Airflow is used.

# Insights gained from this telemetry are critical for prioritizing patches, minor releases, and

# security fixes. Additionally, this information supports key decisions related to the development road map.

# Check the FAQ doc for more information on what data is collected.

#

# Deployments can opt-out of analytics by setting the ``enabled`` option

# to ``False``, or the ``SCARF\_ANALYTICS=false`` environment variable.

# Individual users can easily opt-out of analytics in various ways documented in the

# `Scarf Do Not Track docs <https://docs.scarf.sh/gateway/#do-not-track>`\_\_.

# Enable or disable usage data collection and sending.

#

# Variable: AIRFLOW\_\_USAGE\_DATA\_COLLECTION\_\_ENABLED

#

enabled = True

[common.io]

# Common IO configuration section

# Path to a location on object storage where XComs can be stored in url format.

#

# Example: xcom\_objectstorage\_path = s3://conn\_id@bucket/path

#

# Variable: AIRFLOW\_\_COMMON.IO\_\_XCOM\_OBJECTSTORAGE\_PATH

#

xcom\_objectstorage\_path =

# Threshold in bytes for storing XComs in object storage. -1 means always store in the

# database. 0 means always store in object storage. Any positive number means

# it will be stored in object storage if the size of the value is greater than the threshold.

#

# Example: xcom\_objectstorage\_threshold = 1000000

#

# Variable: AIRFLOW\_\_COMMON.IO\_\_XCOM\_OBJECTSTORAGE\_THRESHOLD

#

xcom\_objectstorage\_threshold = -1

# Compression algorithm to use when storing XComs in object storage. Supported algorithms

# are a.o.: snappy, zip, gzip, bz2, and lzma. If not specified, no compression will be used.

# Note that the compression algorithm must be available in the Python installation (e.g.

# python-snappy for snappy). Zip, gz, bz2 are available by default.

#

# Example: xcom\_objectstorage\_compression = gz

#

# Variable: AIRFLOW\_\_COMMON.IO\_\_XCOM\_OBJECTSTORAGE\_COMPRESSION

#

xcom\_objectstorage\_compression =

[fab]

# This section contains configs specific to FAB provider.

# Boolean for enabling rate limiting on authentication endpoints.

#

# Variable: AIRFLOW\_\_FAB\_\_AUTH\_RATE\_LIMITED

#

auth\_rate\_limited = True

# Rate limit for authentication endpoints.

#

# Variable: AIRFLOW\_\_FAB\_\_AUTH\_RATE\_LIMIT

#

auth\_rate\_limit = 5 per 40 second

# Update FAB permissions and sync security manager roles

# on webserver startup

#

# Variable: AIRFLOW\_\_FAB\_\_UPDATE\_FAB\_PERMS

#

update\_fab\_perms = True

[imap]

# Options for IMAP provider.

# ssl\_context =

[smtp\_provider]

# Options for SMTP provider.

# ssl context to use when using SMTP and IMAP SSL connections. By default, the context is "default"

# which sets it to ``ssl.create\_default\_context()`` which provides the right balance between

# compatibility and security, it however requires that certificates in your operating system are

# updated and that SMTP/IMAP servers of yours have valid certificates that have corresponding public

# keys installed on your machines. You can switch it to "none" if you want to disable checking

# of the certificates, but it is not recommended as it allows MITM (man-in-the-middle) attacks

# if your infrastructure is not sufficiently secured. It should only be set temporarily while you

# are fixing your certificate configuration. This can be typically done by upgrading to newer

# version of the operating system you run Airflow components on,by upgrading/refreshing proper

# certificates in the OS or by updating certificates for your mail servers.

#

# If you do not set this option explicitly, it will use Airflow "email.ssl\_context" configuration,

# but if this configuration is not present, it will use "default" value.

#

# Example: ssl\_context = default

#

# Variable: AIRFLOW\_\_SMTP\_PROVIDER\_\_SSL\_CONTEXT

#

# ssl\_context =

# Allows overriding of the standard templated email subject line when the SmtpNotifier is used.

# Must provide a path to the template.

#

# Example: templated\_email\_subject\_path = path/to/override/email\_subject.html

#

# Variable: AIRFLOW\_\_SMTP\_PROVIDER\_\_TEMPLATED\_EMAIL\_SUBJECT\_PATH

#

# templated\_email\_subject\_path =

# Allows overriding of the standard templated email path when the SmtpNotifier is used. Must provide

# a path to the template.

#

# Example: templated\_html\_content\_path = path/to/override/email.html

#

# Variable: AIRFLOW\_\_SMTP\_PROVIDER\_\_TEMPLATED\_HTML\_CONTENT\_PATH

#

# templated\_html\_content\_path =

Not Using Containers for Airflow

### **Why Are We Not Using Containers for Airflow in This Setup?**

There are a few key reasons why we are installing Airflow **directly on a GCP VM** rather than using **Docker or Kubernetes**:

## **1. Simple, Direct Deployment for a Single-VM Setup**

* Installing Airflow **natively on the VM** is **simpler** when using a **single-machine deployment**.
* If your workflow only requires **one VM**, managing Docker containers adds **unnecessary complexity**.

✅ **Best When**: You want **direct control** over Airflow and Celery workers without worrying about container orchestration.

❌ **Not Ideal When**: You need **high availability** and **scalability** across multiple nodes.

## **2. Avoiding Extra Overhead with Docker**

* Running Airflow **inside containers** requires managing:
  + **Volumes** for persistent storage (logs, DAGs).
  + **Networking** between Airflow services (scheduler, webserver, workers).
  + **Docker Compose** or **Kubernetes Helm charts** for orchestration.
* Since Airflow **already runs multiple processes (webserver, scheduler, Celery, workers)**, adding Docker can make **troubleshooting more complex**.

✅ **Best When**: You want **direct access** to system logs and configurations.

❌ **Not Ideal When**: You plan to scale Airflow **horizontally** across multiple machines.

## **3. Performance Considerations (Direct VM = Less Overhead)**

* Running Airflow **natively on a VM** avoids the **extra resource usage** that comes with Docker.
* In a **small-scale production setup**, avoiding containerized layers results in:
  + **Faster startup times** (no need to launch containers).
  + **Lower CPU/memory overhead**.

✅ **Best When**: Your VM is already optimized for **bare-metal performance**.

❌ **Not Ideal When**: You want **isolated dependency management** with Docker.

## **4. Easier Cloud SQL Integration**

* When Airflow runs **directly on a VM**, it can connect **directly** to **Cloud SQL** (PostgreSQL/MySQL).
* If we were using Docker, we would need:
  + **Cloud SQL Proxy** inside the container.
  + **Networking configurations** for containers to communicate with Cloud SQL.

✅ **Best When**: You want **simpler database connectivity**.

❌ **Not Ideal When**: You want **database connections isolated in containers**.

## **5. Easier Airflow + Airbyte Integration**

* We will install **Airbyte on the same VM**, and Airflow will trigger Airbyte API calls.
* If Airflow was inside **a container**, we would need:
  + **Docker networking configurations** to allow Airflow to talk to Airbyte.
  + **Additional volume mounts** for logs and DAGs.

✅ **Best When**: You want **Airbyte + Airflow running together easily**.

❌ **Not Ideal When**: You plan to **deploy Airbyte separately** on Kubernetes.

### **When Should You Use Containers for Airflow?**

| **Scenario** | **Install Airflow Directly on VM?** | **Use Docker or Kubernetes?** |
| --- | --- | --- |
| **Single GCP VM (low-scale production)** | ✅ Yes | ❌ No |
| **Need persistent logs/DAGs without extra setup** | ✅ Yes | ❌ No |
| **Cloud SQL integration without networking issues** | ✅ Yes | ❌ No |
| **Multi-node distributed setup** | ❌ No | ✅ Yes (Kubernetes) |
| **Scaling workers dynamically** | ❌ No | ✅ Yes (Kubernetes) |
| **Airflow CI/CD with different environments** | ❌ No | ✅ Yes (Docker) |

## **Final Decision: Why We Chose Direct Installation on a GCP VM**

✅ **We are running everything on a single VM**✅ **We want direct access to logs and configurations**✅ **We don’t need complex container networking**✅ **We are integrating Airflow + Airbyte on the same machine**

### **Next Steps**

Now that we understand why we’re **not containerizing Airflow**, we can move forward with **installing Airbyte on the same VM**.

Alternative Cloud Compute Options

### **Alternative Cloud Compute Options for Running Apache Airflow on GCP**

While we are currently deploying Airflow on a **Compute Engine VM**, there are several **alternative architectures** that provide different levels of **scalability, maintainability, and cost efficiency**.

## **1. Cloud Compute Alternatives for Running Airflow on GCP**

| **Option** | **Description** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Compute Engine (VM-based, our current setup)** | Direct installation on a GCP VM | ✅ Simple setup  ✅ Full control over OS & configs  ✅ No extra orchestration needed | ❌ Manual scaling  ❌ Limited HA (unless using a managed DB & backup) |
| **Google Kubernetes Engine (GKE - Containerized Airflow)** | Run Airflow as Docker containers in GKE | ✅ Auto-scaling  ✅ High availability  ✅ Easier multi-node setup | ❌ More complex setup  ❌ Needs Kubernetes expertise |
| **Cloud Composer (Managed Airflow by GCP)** | Fully managed Airflow service | ✅ No setup needed  ✅ Fully managed upgrades & security  ✅ Auto-scaling workers | ❌ Expensive  ❌ Limited customization |
| **Cloud Run (Serverless Airflow DAG Execution)** | Run Airflow DAG execution as **event-driven, serverless tasks** | ✅ No VM management  ✅ Pay-per-use  ✅ Scales instantly | ❌ Not ideal for full Airflow (no scheduler)  ❌ Requires external DB & trigger mechanism |
| **Bare Metal Solution (For High-Performance Workloads)** | Deploy Airflow on dedicated **bare-metal servers** in GCP | ✅ Best for heavy workloads  ✅ Full control over hardware | ❌ Expensive  ❌ Needs infrastructure management |

## **2. Compute Engine (VM) vs. Kubernetes vs. Cloud Composer**

### **Option 1: Compute Engine VM (What We Are Using)**

* **Use Case**: Best for **single-node deployments**, small/medium teams.
* **Deployment**: Airflow installed **natively on the VM**.
* **Scaling**: Manual or via **adding Celery workers**.
* **Database**: Cloud SQL (PostgreSQL or MySQL).

**✅ When to Use Compute Engine:**

* If you want **full control** over Airflow configurations.
* If you **don’t need Kubernetes complexity**.
* If you’re **deploying Airflow and Airbyte together on one VM**.

### **Option 2: Google Kubernetes Engine (GKE) - Scalable Containerized Airflow**

* **Use Case**: Best for **scalable** and **distributed** workflows.
* **Deployment**: Airflow runs as **Docker containers** inside a **Kubernetes cluster**.
* **Scaling**: Auto-scales Celery workers.
* **Database**: Cloud SQL or Kubernetes-native database.

**✅ When to Use GKE:**

* If you need **high availability** (HA) Airflow.
* If your DAGs require **dynamic worker scaling**.
* If your company **already uses Kubernetes**.

**🚀 How to Deploy Airflow on GKE:**

**Create a Kubernetes Cluster**:  
bash  
CopyEdit  
gcloud container clusters create airflow-cluster --num-nodes=3 --zone=us-central1-a

**Deploy Airflow Using Helm**:  
bash  
CopyEdit  
helm repo add apache-airflow https://airflow.apache.org

helm install airflow apache-airflow/airflow

1. **Expose the Web UI & Connect to Cloud SQL**.

### **Option 3: Cloud Composer (Fully Managed Airflow)**

* **Use Case**: Best for **teams who don’t want to manage Airflow infrastructure**.
* **Deployment**: Fully **managed by Google Cloud**.
* **Scaling**: Auto-managed **workers & scheduler**.
* **Database**: Uses **managed PostgreSQL (Cloud SQL)**.

**✅ When to Use Cloud Composer:**

* If you don’t want to manage **infrastructure, scaling, security**.
* If you want **fast Airflow deployment without setup**.
* If your company **is okay with higher costs**.

**🚀 How to Deploy Airflow with Cloud Composer:**

bash

CopyEdit

gcloud composer environments create airflow-env \

--location us-central1 \

--image-version composer-2-airflow-2

* GCP **fully manages** Airflow’s web server, scheduler, and workers.

### **Option 4: Cloud Run (Serverless Airflow DAG Execution)**

* **Use Case**: Best for **event-driven DAG execution** in **serverless environments**.
* **Deployment**: Run **Airflow workers only when needed** (DAG execution as serverless jobs).
* **Scaling**: Fully **serverless (no need to manage VMs)**.
* **Database**: Cloud SQL (external).

**✅ When to Use Cloud Run:**

* If you only need **Airflow DAG execution (not full Airflow UI)**.
* If you want **pay-per-execution pricing**.
* If you want **zero infrastructure management**.

**🚀 How to Deploy Airflow Worker Jobs on Cloud Run:**

**Package the Airflow DAG execution logic into a container**:  
dockerfile  
CopyEdit  
FROM apache/airflow:2.5.0

COPY my\_dag.py /opt/airflow/dags/

CMD ["airflow", "tasks", "run", "my\_dag"]

**Deploy the container to Cloud Run**:  
bash  
CopyEdit  
gcloud run deploy airflow-worker --image=gcr.io/my-project/airflow-dag-runner

### **3. Choosing the Best GCP Option for Airflow**

| **Scenario** | **Best Option** |
| --- | --- |
| **I want full control over Airflow** | Compute Engine (VM) |
| **I need scalable Airflow with auto-scaling workers** | Kubernetes (GKE) |
| **I don’t want to manage infrastructure** | Cloud Composer |
| **I only need Airflow DAG execution, not full UI** | Cloud Run |
| **I need maximum performance (bare metal)** | Bare Metal Solution |

## **4. Summary: Why We Chose Compute Engine for Now**

✅ **We are deploying Airflow and Airbyte together** on the same VM.  
✅ **We need full access** to Airflow configuration files and logs.  
✅ **We are not yet scaling to multiple nodes** (so no need for Kubernetes).  
✅ **We want direct Cloud SQL integration** without extra networking.  
✅ **We want an easy way to debug and manage Airflow** without adding containers.

### **Next Steps**

Since we’ve chosen **Compute Engine for now**, the next steps are:

1. **Install Airbyte on the VM**.
2. **Ensure Airbyte can connect to external sources** (PostgreSQL, APIs).
3. **Integrate Airbyte with Airflow (Triggering Airbyte Syncs from Airflow DAGs).**

Let me know if you’re ready to proceed! 🚀😊

Bare Metal Solution

### **What is a Bare Metal Solution in Cloud Computing?**

A **Bare Metal Solution (BMS)** is a cloud-based offering that provides **dedicated physical servers** without any virtualization overhead. Unlike traditional **virtual machines (VMs)**, where multiple virtual servers share a single physical machine, **bare metal servers** give you full access to the **physical hardware**.

## **1. Bare Metal vs. Virtual Machines**

| **Feature** | **Bare Metal** | **Virtual Machine (VM)** |
| --- | --- | --- |
| **Performance** | High (No virtualization overhead) | Lower (Hypervisor overhead) |
| **Customization** | Full hardware control | Limited by virtualization |
| **Scalability** | Manual scaling required | Auto-scaling possible |
| **Isolation** | Dedicated server, no shared resources | Shared with other tenants |
| **Security** | Higher (no multi-tenant risk) | Lower (depends on hypervisor security) |
| **Cost** | Higher (since it’s dedicated) | Lower (shared resources) |

## **2. Google Cloud Bare Metal Solution (BMS)**

### **How It Works**

* Instead of using **Compute Engine VMs**, you get **physical machines** with **no virtualization layer**.
* The servers are located in **Google Cloud data centers** but operate like **on-premises hardware**.
* You can run **Airflow, databases, Kubernetes, or high-performance workloads** directly on the hardware.

### **Who Uses It?**

* Enterprises needing **maximum performance** for big data, AI/ML, and databases.
* Companies migrating **on-premises workloads** that require **custom hardware**.
* Businesses running **Oracle databases, SAP, or latency-sensitive apps**.

## **3. When to Use Bare Metal for Airflow?**

| **Scenario** | **Use Bare Metal?** |
| --- | --- |
| **Running standard Airflow DAGs** | ❌ No |
| **Need high-performance DAG execution (ML, big data processing)** | ✅ Yes |
| **Want dedicated, powerful hardware without virtualization overhead** | ✅ Yes |
| **Need full control over networking, storage, and OS** | ✅ Yes |
| **Running databases like PostgreSQL, Oracle, or MySQL with Airflow** | ✅ Yes |

## **4. Deploying Airflow on Bare Metal in GCP**

To use **Google Cloud Bare Metal Solution**, follow these steps:

### **Step 1: Provision a Bare Metal Server**

1. Go to **Google Cloud Console** → **Bare Metal Solution**.
2. Click **"Create Instance"**.
3. Choose:
   * **Server type**: bm-standard-112 (112 vCPUs, 3TB RAM) or lower.
   * **Location**: Choose a Google Cloud region.
   * **Storage**: Attach NVMe SSDs for fast I/O.

### **Step 2: Install Airflow**

Connect to the Bare Metal instance:  
bash  
CopyEdit  
ssh user@<BARE\_METAL\_IP>

Install Airflow:  
bash  
CopyEdit  
sudo apt update && sudo apt install python3-pip python3-venv -y

python3 -m venv airflow-venv

source airflow-venv/bin/activate

pip install apache-airflow

### **Step 3: Configure Airflow for High Performance**

* Store DAGs and logs on **high-speed NVMe disks**.
* Optimize Celery workers for **parallel DAG execution**.
* Connect Airflow to **Cloud SQL (or a Bare Metal PostgreSQL/MySQL instance)**.

## **5. When NOT to Use Bare Metal**

❌ **If you don’t need extreme performance** – GCP **Compute Engine VMs** are sufficient.  
❌ **If you need auto-scaling** – Bare metal servers require **manual scaling**.  
❌ **If you want cost efficiency** – VMs are cheaper than dedicated hardware.

### **Conclusion: Should You Use Bare Metal for Airflow?**

✅ **Yes** → If you need **high-performance, low-latency Airflow DAG execution**.  
❌ **No** → If you just need a **normal Airflow setup, use Compute Engine VMs** instead.

Let me know if you want to explore this further! 🚀😊

Airbyte on Kubernetes

### **Complete Guide: Deploying Apache Airflow on Google Kubernetes Engine (GKE)**

Running **Apache Airflow on Kubernetes (GKE)** allows **auto-scaling, high availability, and better resource management** compared to a single VM-based deployment.

## **🚀 Why Deploy Airflow on Kubernetes?**

✅ **Auto-Scales Workers** dynamically based on DAG execution.  
✅ **High Availability (HA)** – If a worker or scheduler crashes, Kubernetes restarts it.  
✅ **Better Resource Isolation** – Each Airflow component runs in its own container.  
✅ **Easier CI/CD & Updates** – Deploy new Airflow versions with Helm and Kubernetes manifests.

# **🛠️ Step-by-Step Guide: Deploying Airflow on GKE**

We’ll follow these steps:

✅ **Step 1:** Set up Google Kubernetes Engine (GKE).  
✅ **Step 2:** Install Helm (package manager for Kubernetes).  
✅ **Step 3:** Deploy Airflow using the Helm Chart.  
✅ **Step 4:** Configure Cloud SQL for Airflow metadata.  
✅ **Step 5:** Access the Airflow web UI.  
✅ **Step 6:** Scale Airflow workers.  
✅ **Step 7:** Set up persistent logs and DAG storage.

## **Step 1: Create a Kubernetes Cluster on GKE**

### **1.1 Create the Cluster**

Run the following command to create a **GKE cluster** with 3 nodes:

bash

CopyEdit

gcloud container clusters create airflow-cluster \

--num-nodes=3 \

--machine-type=e2-standard-4 \

--zone=us-central1-a \

--enable-ip-alias

* **--num-nodes=3** → Creates 3 worker nodes.
* **--machine-type=e2-standard-4** → Each node gets 4 vCPUs and 16GB RAM.
* **--enable-ip-alias** → Allows Kubernetes pods to have unique IPs.

Zone: northamerica-northeast1

### **1.2 Authenticate Your Kubernetes Cluster**

After the cluster is created, authenticate kubectl (Kubernetes CLI):

bash

CopyEdit

~~gcloud container clusters get-credentials airflow-cluster --zone=us-central1-a~~

-zone in above gives error, using region:

gcloud container clusters get-credentials airflow-cluster --region=northamerica-northeast1

Copied form the CLoud Console:

gcloud container clusters get-credentials airflow-cluster --region northamerica-northeast1 --project vector-analytics-448522

* This allows kubectl to communicate with the GKE cluster.

### **What is kubectl?**

* + - kubectl is the **command-line tool** used to interact with **Kubernetes clusters**. It allows you to **deploy applications, manage resources, monitor the cluster, and troubleshoot issues**.

## **1. Why is kubectl Important?**

* + - ✅ **Control Kubernetes Clusters** – Manage nodes, pods, deployments, and services.  
      ✅ **Deploy Applications** – Create, update, and delete workloads.  
      ✅ **Monitor Cluster Health** – Check pod status, logs, and events.  
      ✅ **Scale Applications** – Increase or decrease replica counts.  
      ✅ **Manage Networking** – Expose services and handle ingress/egress traffic.

### **1.3 Verify Cluster Status**

Check if the cluster is running:

bash

CopyEdit

kubectl get nodes

You should see **3 nodes** in READY state.

## **Step 2: Install Helm (If Not Installed)**

Helm is a package manager for Kubernetes that simplifies Airflow deployment.

### **2.1 Install Helm (Linux/Mac)**

bash

CopyEdit

curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash

### **2.2 Verify Helm Installation**

bash

CopyEdit

helm version

## **Step 3: Deploy Airflow Using Helm**

Apache Airflow has an official **Helm Chart** for Kubernetes deployment.

### **3.1 Add the Airflow Helm Repository**

bash

CopyEdit

helm repo add apache-airflow https://airflow.apache.org

helm repo update

### **3.2 Create a Namespace for Airflow**

Namespaces isolate applications in Kubernetes.

bash

CopyEdit

kubectl create namespace airflow

### **3.3 Install Airflow with Helm**

Run the following command:

bash

CopyEdit

helm install airflow apache-airflow/airflow \

--namespace airflow \

--set executor=CeleryExecutor \

--set airflow.database.secretName=airflow-db-secret

* **executor=CeleryExecutor** → Uses CeleryExecutor for distributed task execution.
* **airflow.database.secretName=airflow-db-secret** → (We’ll create this secret in Step 4).

### **3.4 Verify Airflow Components**

bash

CopyEdit

kubectl get pods -n airflow

You should see:

plaintext

CopyEdit

NAME READY STATUS RESTARTS

airflow-webserver 1/1 Running 0

airflow-scheduler 1/1 Running 0

airflow-worker-0 1/1 Running 0

airflow-postgresql-0 1/1 Running 0

## **Step 4: Configure Cloud SQL for Airflow Metadata**

Instead of using the default **PostgreSQL inside Kubernetes**, we’ll use **Cloud SQL for Airflow’s metadata**.

### **4.1 Create a Cloud SQL Instance**

bash

CopyEdit

gcloud sql instances create airflow-db \

--database-version=POSTGRES\_13 \

--tier=db-f1-micro \

--region=us-central1

### **4.2 Create the Airflow Metadata Database**

bash

CopyEdit

gcloud sql databases create airflow\_metadata --instance=airflow-db

### **4.3 Set Up Cloud SQL Proxy**

Cloud SQL Proxy allows Kubernetes to connect securely to Cloud SQL.

bash

CopyEdit

kubectl create deployment cloudsql-proxy \

--image=gcr.io/cloudsql-docker/gce-proxy:latest \

--namespace airflow \

-- --dir=/cloudsql \

--instances=<PROJECT\_ID>:us-central1:airflow-db=tcp:5432

Replace **<PROJECT\_ID>** with your actual **Google Cloud project ID**.

### **4.4 Store Database Credentials in a Kubernetes Secret**

bash

CopyEdit

kubectl create secret generic airflow-db-secret --namespace airflow \

--from-literal=connection=postgresql+psycopg2://postgres:YOUR\_PASSWORD@airflow-db:5432/airflow\_metadata

* Replace **YOUR\_PASSWORD** with your Cloud SQL **PostgreSQL password**.

## **Step 5: Access the Airflow Web UI**

By default, Airflow is deployed inside Kubernetes. To expose it:

### **5.1 Port Forward to Your Local Machine**

bash

CopyEdit

kubectl port-forward svc/airflow-webserver 8080:8080 --namespace airflow

Now, open **http://localhost:8080** in your browser.

### **5.2 Get Airflow Admin Password**

bash

CopyEdit

kubectl get secret --namespace airflow airflow-webserver -o jsonpath="{.data.airflow-password}" | base64 --decode

Use this password to log in.

## **Step 6: Scale Airflow Workers**

If you need more workers, scale them dynamically:

### **6.1 Increase Worker Count**

bash

CopyEdit

kubectl scale statefulset airflow-worker --replicas=5 -n airflow

* This starts **5 Celery workers** instead of 1.

### **6.2 Verify Workers Are Running**

bash

CopyEdit

kubectl get pods -n airflow

## **Step 7: Store DAGs and Logs in Google Cloud Storage**

By default, Airflow stores DAGs in a local directory. Instead, store them in **GCS** for better durability.

### **7.1 Create a GCS Bucket**

bash

CopyEdit

gsutil mb gs://airflow-dags-bucket

### **7.2 Mount GCS Bucket in Airflow Configuration**

Edit airflow.cfg:

ini

CopyEdit

dags\_folder = gs://airflow-dags-bucket/dags

remote\_logging = True

remote\_log\_conn\_id = gcs\_default

remote\_base\_log\_folder = gs://airflow-logs

Apply changes:

bash

CopyEdit

kubectl rollout restart deployment airflow-webserver -n airflow

# **🎯 Summary: What We've Done**

✅ **Set up a Kubernetes cluster on GKE**✅ **Installed Helm and deployed Airflow**✅ **Configured Cloud SQL as the metadata database**✅ **Exposed the Airflow web UI**✅ **Scaled Celery workers**✅ **Stored DAGs and logs in Google Cloud Storage**

# **🎯 Next Steps**

1. **Deploy Airbyte on Kubernetes (for data extraction & ingestion).**
2. **Integrate Airflow with Airbyte (trigger syncs using DAGs).**
3. **Set up monitoring and alerting for production workflows.**

Let me know if you're ready to move to the next step! 🚀😊

(GKE) Cluster through UI

### **How to Create a Google Kubernetes Engine (GKE) Cluster from the Google Cloud Console (UI)**

You can create a **GKE cluster** directly from the **Google Cloud Console (UI)** without using the CLI. Here’s a step-by-step guide.

## **Step 1: Open Google Kubernetes Engine (GKE)**

1. Go to the **Google Cloud Console**:  
   👉 GKE Console
2. Click **"Create"** to start setting up a new Kubernetes cluster.

## **Step 2: Choose the Cluster Type**

GKE offers different cluster types. Select **"Standard"** or **"Autopilot"** depending on your needs.

| **Cluster Type** | **Best For** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Standard Cluster** | Full control over nodes and configurations | ✅ More flexibility  ✅ Customize machine types, networking | ❌ Requires more management |
| **Autopilot Cluster** | Fully managed GKE | ✅ Google manages everything  ✅ Auto-scales automatically | ❌ Less control over resources |

For **Airflow on Kubernetes**, choose **"Standard"**.

## **Step 3: Configure the Cluster**

1. **Cluster Basics:**
   * **Cluster Name**: airflow-cluster
   * **Location Type**: Choose **"Zonal"** (single zone) or **"Regional"** (high availability across multiple zones).
   * Recommended: Use **"Zonal"** (us-central1-a or any preferred zone) for lower costs.
2. **Node Pool Configuration** (For default-pool):
   * **Number of Nodes**: 3
   * **Machine Type**: e2-standard-4 (4 vCPUs, 16GB RAM)
   * **Boot Disk**: 50GB, SSD recommended for performance.
3. **Networking:**
   * **VPC Network**: default (or create a custom one).
   * **Networking Mode**: **"VPC-native"** (recommended).
   * **Enable Private Cluster** (if you want enhanced security).
4. **Access & Security:**
   * **Enable Workload Identity** (Recommended for IAM security).
   * **Enable Automatic Node Upgrades** (Ensures security updates).

## **Step 4: Enable Cluster Monitoring (Optional)**

* Under **Operations**, check: ✅ **Enable Cloud Monitoring**✅ **Enable Cloud Logging**

## **Step 5: Create the Cluster**

1. Click **"Create"**.
2. GCP will start provisioning your cluster (takes ~5-10 minutes).
3. Once the cluster is ready, you’ll see **"Running"** under its status.

## **Step 6: Connect to Your GKE Cluster**

After your cluster is created, you need to connect to it.

1. Click **"Connect"** on your cluster.

Copy the gcloud command shown and run it in Cloud Shell or your terminal:  
bash  
CopyEdit  
gcloud container clusters get-credentials airflow-cluster --zone=us-central1-a

Verify that your cluster is ready:  
bash  
CopyEdit  
kubectl get nodes

You should see output similar to:  
scss  
CopyEdit  
NAME STATUS ROLES AGE VERSION

gke-airflow-cluster-default-pool-12345 Ready <none> 5m v1.23.8-gke.1500

## **Step 7: Deploy Apache Airflow on GKE**

Now that your cluster is running, you can proceed with **installing Apache Airflow using Helm** (as outlined in the previous guide).

### **✅ Summary: What We've Done**

✅ Created a **GKE Standard cluster** from the UI  
✅ Configured **3 nodes, e2-standard-4 machines**✅ Enabled **VPC networking & security**✅ Connected to the cluster using kubectl

### **🚀 Next Steps**

1. **Deploy Airflow on the GKE cluster using Helm.**
2. **Set up Cloud SQL for Airflow metadata.**
3. **Scale Airflow workers dynamically.**

Let me know if you’re ready for the next step! 🚀😊

O

kubectl get nodes Showing "No resources found"

### **Why is kubectl get nodes Showing "No resources found"?**

If you run:

bash

CopyEdit

kubectl get nodes

and see **"No resources found"**, it means that your kubectl is **not connected to a Kubernetes cluster**, or the cluster is **not running**.

## **🔍 Step 1: Check if kubectl is Connected to a Cluster**

Run:

bash

CopyEdit

kubectl config current-context

* If it **returns an empty result**, it means kubectl is **not connected to any Kubernetes cluster**.
* If it **returns a context name**, it means kubectl is configured but may be pointing to the wrong cluster.

I get this: gke\_vector-analytics-448522\_northamerica-northeast1\_airflow-cluster

## **🔍 Step 2: List Available Clusters**

Run:

bash

CopyEdit

kubectl config get-contexts

If no contexts are listed, it means **no Kubernetes clusters are configured**.

I get this:

malik\_hassan@cloudshell:~ (vector-analytics-448522)$ kubectl config get-contexts

CURRENT NAME CLUSTER AUTHINFO NAMESPACE

\* gke\_vector-analytics-448522\_northamerica-northeast1\_airflow-cluster gke\_vector-analytics-448522\_northamerica-northeast1\_airflow-cluster gke\_vector-analytics-448522\_northamerica-northeast1\_airflow-cluster

## **🔧 Step 3: Connect to Your GKE Cluster**

If you are using **Google Kubernetes Engine (GKE)**, you need to authenticate and set up kubectl to connect to your cluster.

### **3.1 Find Your Cluster Name**

Run:

bash

CopyEdit

gcloud container clusters list

You should see output like this:

plaintext

CopyEdit

NAME LOCATION MASTER\_VERSION

airflow-cluster us-central1-a 1.24.10-gke.2500

### **3.2 Reconnect to the Cluster**

Run:

bash

CopyEdit

gcloud container clusters get-credentials airflow-cluster --zone=us-central1-a

gcloud container clusters get-credentials airflow-cluster --zone=northamerica-northeast1

* Replace **airflow-cluster** with your actual cluster name.
* Replace **us-central1-a** with the zone of your cluster.

After this, verify again with:

bash

CopyEdit

kubectl get nodes

## **🔍 Step 4: Check If the Cluster is Running**

If your cluster is not running, restart it from the **GCP Console**:

1. Go to **GKE Console**
2. Click on your cluster (airflow-cluster).
3. If the status is **"Stopped"**, click **"Start"**.

## **🔍 Step 5: Check Your IAM Permissions**

If you see errors like **"Permission Denied"**, your account might not have permissions to view nodes.

Run:

bash

CopyEdit

gcloud projects get-iam-policy <PROJECT\_ID>

* Ensure your user has **Kubernetes Engine Admin** (roles/container.admin) permissions.

If needed, ask a GCP admin to grant you access:

bash

CopyEdit

gcloud projects add-iam-policy-binding <PROJECT\_ID> \

--member="user:your-email@example.com" \

--role="roles/container.admin"

## **✅ Final Check**

Run:

bash

CopyEdit

kubectl get nodes

If it now shows nodes like this:

plaintext

CopyEdit

NAME STATUS ROLES AGE VERSION

gke-airflow-cluster-default-pool-12345 Ready <none> 5m v1.23.8-gke.1500

Then your Kubernetes cluster is now correctly set up! 🚀

### **🛠️ If the Issue Persists**

* If kubectl is still not showing nodes, **delete and recreate the cluster**.

If you need further debugging, run:  
bash  
CopyEdit  
kubectl cluster-info

kubectl describe nodes

Let me know if you need more help! 😊

Error: NO node

### **Error: NO node Due to no explicit nodes because of Auto Nodes**

### **Why Are You Getting This Error?**

The error:

plaintext

CopyEdit

ERROR: (gcloud.container.node-pools.list) ResponseError: code=400, message=Autopilot node pools cannot be accessed or modified.

indicates that your **GKE cluster is an Autopilot cluster**, and **Autopilot mode does not allow direct node management**.

### **🔍 Understanding GKE Autopilot Mode**

GKE **Autopilot** manages the **nodes and node pools automatically**, which means: ✅ You **don’t need to manually manage nodes**.  
✅ **Google Cloud automatically scales and provisions nodes** as needed.  
❌ You **cannot manually list, modify, or create node pools**.

## **🔧 Fix: Verify Cluster Type**

Check if your cluster is **Autopilot or Standard**:

bash

CopyEdit

gcloud container clusters list --region=northamerica-northeast1

Look at the **MODE** column in the output.

* If it shows **MODE: Autopilot**, you **cannot manually list or modify nodes**.
* If it shows **MODE: Standard**, you should be able to list node pools.

## **🔍 Step 1: Verify Cluster Status**

Run:

bash

CopyEdit

gcloud container clusters describe airflow-cluster --region=northamerica-northeast1

Check:

* **Status**: Ensure it is RUNNING.
* **Mode**: If AUTOPILOT, you cannot modify node pools.

## **🔧 Step 2: Get Nodes in an Autopilot Cluster**

Since you **cannot manage node pools directly**, try listing nodes with kubectl:

bash

CopyEdit

kubectl get nodes

If it still shows **"No resources found"**, check workloads:

bash

CopyEdit

kubectl get pods --all-namespaces

This will show if any Airflow services are running.

## **🔧 Step 3: If No Nodes Are Found**

If **no nodes or pods are running**, the cluster **might not be provisioning resources automatically**.

### **Option 1: Deploy a Test Pod**

Try deploying a small workload to force Autopilot to provision nodes:

bash

CopyEdit

kubectl run test-pod --image=nginx --restart=Never

Then check:

bash

CopyEdit

kubectl get nodes

kubectl get pods

### **Option 2: Restart Cluster (Last Resort)**

bash

CopyEdit

gcloud container clusters update airflow-cluster --region=northamerica-northeast1 --enable-autoscaling

If the issue persists, consider **deleting and recreating the cluster as a Standard cluster**.

## **🔄 Alternative: Create a Standard Cluster Instead**

If you **need full control over nodes**, create a **Standard GKE cluster** instead of Autopilot:

### **Delete the Existing Autopilot Cluster**

bash

CopyEdit

gcloud container clusters delete airflow-cluster --region=northamerica-northeast1

### **Create a New Standard Cluster**

bash

CopyEdit

gcloud container clusters create airflow-cluster \

--num-nodes=3 \

--machine-type=e2-standard-4 \

--region=northamerica-northeast1

Reconnect:

bash

CopyEdit

gcloud container clusters get-credentials airflow-cluster --region=northamerica-northeast1

kubectl get nodes

### **✅ Summary**

| **Issue** | **Fix** |
| --- | --- |
| **GKE Autopilot does not allow node modifications** | Use kubectl get nodes instead of gcloud commands |
| **No nodes found in kubectl get nodes** | Deploy a test pod to force Autopilot to provision nodes |
| **Want full control over nodes** | Delete and recreate as a **Standard Cluster** |

Let me know what you want to do next! 🚀😊

Current leaving point

TIDBIT venv

TIDBIT: Always activate Virtual Environment  
source airflow-venv/bin/activate

Starting Airflow:

airflow db migrate

airflow webserver -p 8080 &

airflow scheduler &

airflow celery worker