This tutorial demonstrates a way to automate cloud infrastructure by using Cloud Composer. The example shows how to schedule automated backups of Compute Engine (/compute/) virtual machine (VM) instances.

Cloud Composer (/composer/) is a fully managed workflow orchestration service on Google Cloud. Cloud Composer lets you author workflows with a Python API, schedule them to run automatically or start them manually, and monitor the execution of their tasks in real time through a graphical UI.

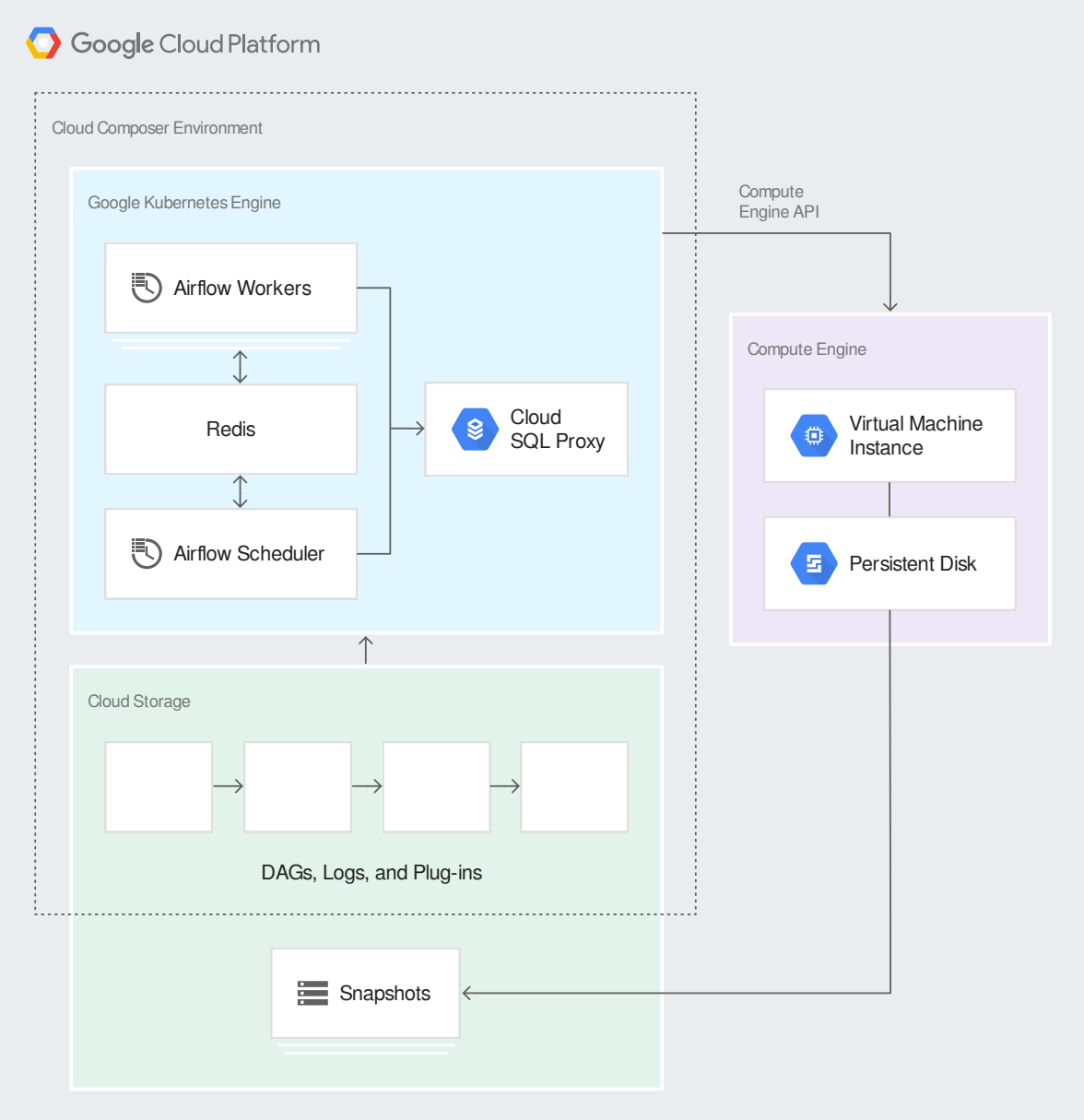
Cloud Composer is based on Apache Airflow (https://airflow.apache.org/). Google runs this open source orchestration platform on top of a Google Kubernetes Engine (/kubernetes-engine/) (GKE) cluster. This cluster manages the Airflow workers, and opens up a host of integration opportunities with other Google Cloud products.

This tutorial is intended for operators, IT administrators, and developers who are interested in automating infrastructure and taking a deep technical dive into the core features of Cloud Composer. The tutorial is not meant as an enterprise-level disaster recovery (DR) guide nor as a best practices guide for backups. For more information on how to create a DR plan for your enterprise, see the disaster recovery planning guide (/solutions/dr-scenarios-planning-guide).

Cloud Composer workflows are defined by creating a Directed Acyclic Graph (https://wikipedia.org/wiki/Directed_acyclic_graph) (DAG). From an Airflow perspective (https://airflow.apache.org/concepts.html#dags), a DAG is a collection of tasks organized to reflect their directional interdependencies. In this tutorial, you learn how to define an Airflow workflow that runs regularly to back up a Compute Engine virtual machine instance using Persistent Disk snapshots.

The Compute Engine VM used in this example consists of an instance with an associated boot persistent disk. Following the snapshot guidelines, described later, the Cloud Composer backup workflow calls the Compute Engine API to stop the instance, take a snapshot of the persistent disk, and restart the instance. In between these tasks, the workflow waits for each operation to complete before proceeding.

The following diagram summarizes the architecture:



Before you begin the tutorial, the next section shows you how to create a Cloud Composer environment. The advantage of this environment is that it uses multiple Google Cloud products, but you don't have to configure each one individually.

- **Cloud Storage**: The Airflow DAG, plugin, and logs are stored in a Cloud Storage bucket.
- **Google Kubernetes Engine**: The Airflow platform is based on a micro-service architecture, and is suitable to run in GKE.

    - Airflow workers load plugin and workflow definitions from Cloud Storage and run each task, using the Compute Engine API.
    - The Airflow scheduler makes sure that backups are executed in the configured cadence, and with the proper task order.
    - Redis is used as a message broker between Airflow components.
    - Cloud SQL Proxy is used to communicate with the metadata repository.
- **Cloud SQL and App Engine Flex**: Cloud Composer also uses a Cloud SQL instance for metadata and an App Engine Flex app that serves the Airflow UI. These resources are not pictured in the diagram because they live in a separate Google-managed project.

For more details, see the Overview of Cloud Composer (/composer/docs/concepts/overview).

The use case presented in this tutorial is simple: take a snapshot of a single virtual machine with a fixed schedule. However, a real-world scenario can include hundreds of VMs belonging to different parts of the organization, or different tiers of a system, each requiring different backup schedules. Scaling applies not only to our example with Compute Engine VMs, but to any infrastructure component for which a scheduled process needs to be run

Cloud Composer excels at these complex scenarios because it's a full-fledged workflow engine based on Apache Airflow hosted in the cloud, and not just an alternative to Cloud Scheduler (/scheduler/) or `cron`.

Airflow DAGs, which are flexible representations of a workflow, adapt to real-world needs while still running from a single codebase. To build DAGs suitable for your use case, you can use a combination of the following two approaches:

- Create one DAG instance for groups of infrastructure components where the same schedule can be used to start the process.
- Create independent DAG instances for groups of infrastructure components that require their own schedules.

A DAG can process components in parallel. A task must either start an asynchronous operation for each component, or you must create a branch to process each component. You can build DAGs dynamically from code to add or remove branches and tasks as needed.

Also, you can model dependencies between application tiers within the same DAG. For example: you might want to stop all the web server instances before you stop any app server instances.

These optimizations are outside of the scope of the current tutorial.

Persistent Disk (/persistent-disk/) is durable block storage that can be attached to a virtual machine instance and used either as the primary boot disk for the instance or as a secondary non-boot disk for critical data. PDs are highly available (https://youtu.be/AKM01LFRmAg?t=560)—for every write, three replicas are written, but Google Cloud customers are charged for only one of them.

A snapshot (/compute/docs/disks/create-snapshots) is an exact copy of a persistent disk at a given point in time. Snapshots are incremental and compressed, and are stored transparently in Cloud Storage.

It's possible to take snapshots of any persistent disk while apps are running. No snapshot will ever contain a partially written block. However, if a write operation spanning several blocks is in flight when the backend receives the snapshot creation request, that snapshot might contain only some of the updated blocks. You can deal with these inconsistencies the same way you would address unclean shutdowns.

We recommend that you follow these guidelines to ensure that snapshots are consistent:

- Minimize or avoid disk writes during the snapshot creation process. Scheduling backups during off-peak hours is a good start.
- For secondary non-boot disks, pause apps and processes that write data and freeze or unmount the file system.
- For boot disks, it's not safe or feasible to freeze the root volume. Stopping the virtual machine instance before taking a snapshot might be a suitable approach.

  To avoid service downtime caused by freezing or stopping a virtual machine, we recommend using a highly available architecture. For more information, see Disaster recovery scenarios for applications (/solutions/dr-scenarios-for-applications).
- Use a consistent naming convention for the snapshots. For example, use a timestamp with an appropriate granularity, concatenated with the name of the instance, disk, and zone.

For more information on creating consistent snapshots, see snapshot best practices (/compute/docs/disks/snapshot-best-practices).

- Create custom Airflow operators and a sensor for Compute Engine.
- Create a Cloud Composer workflow using the Airflow operators and a sensor.
- Schedule the workflow to back up a Compute Engine instance at regular intervals.

- Compute Engine (/compute/pricing/)
- GKE (/kubernetes-engine/pricing/)
- Cloud Storage (/storage/pricing/)
- Cloud Composer environment (/composer/pricing/)

You can use the pricing calculator (https://cloud.google.com/products/calculator/#id=53faae5c-7c29-45be-b1d0-a6edf0790242) to generate a cost estimate based on your projected usage.

1. Sign in (https://accounts.google.com/Login) to your Google Account.

   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).
2. In the Cloud Console, on the project selector page, select or create a Cloud project.

★ **Note**: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

Go to the project selector page (https://console.cloud.google.com/projectselector2/home/dashboard)

3. Make sure that billing is enabled for your Google Cloud project. Learn how to confirm billing is enabled for your project (/billing/docs/how-to/modify-project).
4. Create a Cloud Composer environment. To minimize cost, choose a disk size of 20 GB.

   GO TO THE CREATE ENVIRONMENT PAGE (/composer/docs/how-to/managing/creating)

   It usually takes about 15 minutes to provision the Cloud Composer environment, but it can take up to one hour.
5. The full code for this tutorial is available on GitHub. To examine the files as you follow along, open the repository in Cloud Shell:

   GO TO Cloud Shell (https://console.cloud.google.com/cloudshell)
6. In the Cloud Shell console home directory, run the following command:

When you finish this tutorial, you can avoid continued billing by deleting the resources you created. For more information, see Cleaning up (#clean-up).

The first step is to create the sample Compute Engine virtual machine instance to back up. This instance runs WordPress (https://wordpress.org/), an open source content management system.

Follow these steps to create the WordPress instance on Compute Engine:

1. In Google Cloud Marketplace, go to the WordPress Certified by Bitnami (https://console.cloud.google.com/marketplace/details/bitnami-launchpad/wordpress) launch page.
2. Click **Launch on Compute Engine**.
3. A pop-up window appears with a list of your projects. Select the project you previously created for this tutorial.

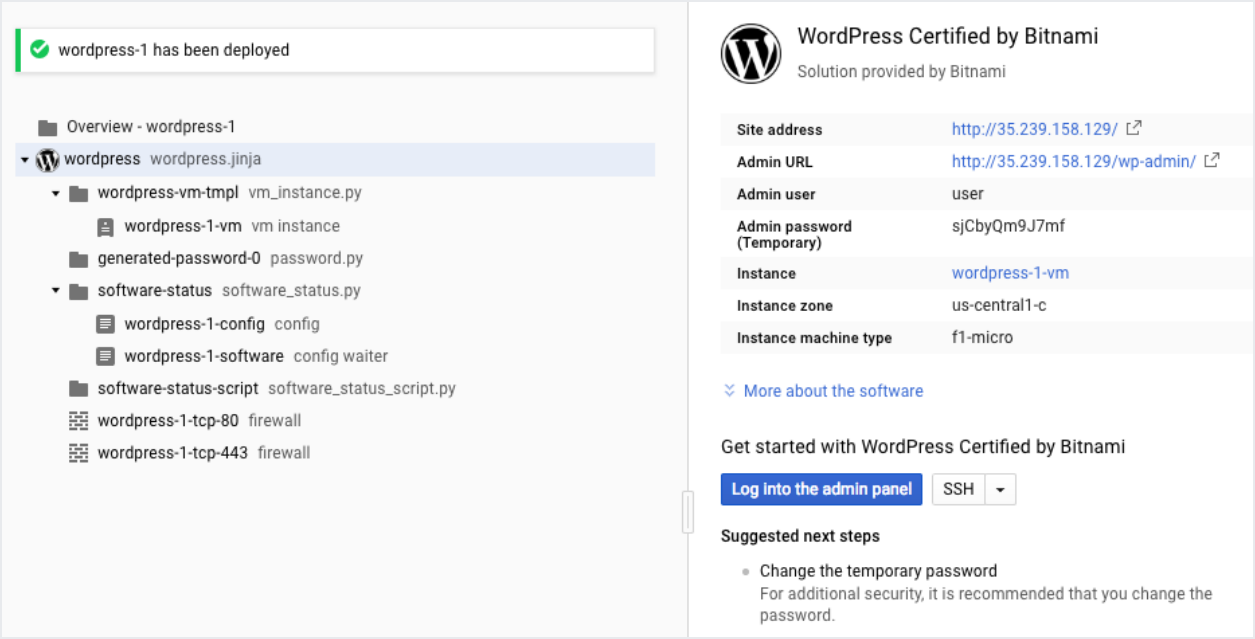
Google Cloud configures the required APIs in your project, and after a short wait it shows a screen with the different configuration options for your WordPress Compute Engine instance.

4. Optionally, change the boot disk type to SSD to increase the instance boot speed.
5. Click **Deploy**.

   You are taken to the Deployment Manager screen, where you can see the status of the deployment.

   The WordPress Deployment Manager script creates the WordPress Compute Engine instance and two firewall rules to allow TCP traffic to reach the instance through ports 80 and 443. This process might take several minutes, with each item being deployed and showing a progress-wheel icon.

   When the process is completed, your WordPress instance is ready and serving the default content on the website URL. The Deployment Manager screen shows the website URL (**Site address**), the administration console URL (**Admin URL**) with its user and password, documentation links, and suggested next steps.



6. Click the site address to verify that your WordPress instance is up and running. You should see a default WordPress blog page.

The sample Compute Engine instance is now ready. The next step is to configure an automatic incremental backup process of that instance's persistent disk.

To back up the persistent disk of the test instance, you can create an Airflow workflow that stops the instance, takes a snapshot of its persistent disk, and restarts the instance. Each of these tasks is defined as code with a custom Airflow operator (https://airflow.apache.org/concepts.html#operators). Operators' code is then grouped in an Airflow plugin (https://airflow.apache.org/plugins.html).

In this section, you learn how to build custom Airflow operators that call the Compute Engine Python Client library (/compute/docs/api/libraries#google_apis_python_client_library) to control the instance lifecycle. You have other options for doing this, for example:

- Use the Airflow `BashOperator` to execute `gcloud compute` commands (/sdk/gcloud/reference/compute/).
- Use the Airflow `HTTPOperator` to execute HTTP calls directly to the Compute Engine REST API (/compute/docs/reference/rest/v1/).
- Use the Airflow `PythonOperator` to call arbitrary Python functions without defining custom operators.

This tutorial doesn't explore those alternatives.

The custom operators that you create in this tutorial use the Python Client Library to call the Compute Engine API. Requests to the API must be authenticated and authorized. The recommended way is to use a strategy called Application Default Credentials (/docs/authentication/production#providing_credentials_to_your_application) (ADC).

The ADC strategy is applied whenever a call is made from a client library:

1. The library verifies if a service account is specified in the environment variable `GOOGLE_APPLICATION_CREDENTIALS`.
2. If the service account is not specified, the library uses the default service account that Compute Engine or GKE provides.

If these two methods fail, an error occurs.

Airflow operators in this tutorial fall under the second method. When you create the Cloud Composer environment, a GKE cluster is provisioned. The nodes of this cluster run Airflow worker pods. In turn, these workers execute the workflow with the custom operators you define. Because you didn't specify a service account when you created the environment, the default service account (/iam/docs/service-accounts#user-managed_service_accounts) for the GKE cluster nodes is what the ADC strategy uses.

GKE cluster nodes are Compute Engine instances. So it's straightforward to obtain the credentials associated with the Compute Engine default service account in the operator code.

no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

m/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

This code uses the default application credentials to create a Python client that will send requests to the Compute Engine API. In the following sections, you reference this code when creating each Airflow operator.

As an alternative to using the default Compute Engine service account, it's possible to create a service account and configure it as a connection in the Airflow administration console. This method is described in the Managing Airflow connections page (/composer/docs/how-to/managing/connections) and allows for more granular access control to Google Cloud resources. This tutorial doesn't explore this alternative.

This section analyzes the creation of the first custom Airflow operator, `StopInstanceOperator`. This operator calls the Compute Engine API to stop the Compute Engine instance that's running WordPress:

1. In Cloud Shell, use a text editor such as nano or vim to open the `gce_commands_plugin.py` file:

2. Examine the imports at the top of the file:

no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

```
om/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)
```

The notable imports are:

- `BaseOperator` (https://airflow.apache.org/_api/index.html#baseoperator): base class that all Airflow custom operators are required to inherit.
- `AirflowPlugin` (https://airflow.apache.org/plugins.html): base class to create a group of operators, forming a plugin.
- `apply_defaults`: function decorator that fills arguments with default values if they are not specified in the operator constructor.
- `GoogleCredentials`: class used to retrieve the app default credentials.
- `googleapiclient.discovery` (https://github.com/googleapis/google-api-python-client): client library entry point that allows the discovery of the underlying Google APIs. In this case, the client library builds a resource to interact with the Compute Engine API (/compute/docs/api/libraries#google_apis_python_client_library).

3. Next, look at the `StopInstanceOperator` class below the imports:

no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

om/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

The `StopInstanceOperator` class has three methods:

- `__init__`: the class constructor. Receives the project name, the zone where the instance is running, and the name of the instance you want to stop. Also, it initializes the `self.compute` variable by calling `get_compute_api_client`.
- `get_compute_api_client`: helper method that returns an instance of the Compute Engine API. It uses the ADC provided by `GoogleCredentials` to authenticate with the API and authorize subsequent calls.

- `execute`: main operator method overridden from `BaseOperator`. Airflow calls this method to run the operator. The method prints an info message to the logs and then calls the Compute Engine API to stop the Compute Engine instance specified by the three parameters received in the constructor. The `sleep()` function at the end waits until the instance has been stopped. In a production environment, you must use a more deterministic method such as operator cross-communication. That technique is described later in this tutorial.

The `stop()` method from the Compute Engine API shuts down the virtual machine instance cleanly (/compute/docs/instances/stopping-or-deleting-an-instance#stop_an_instance). The operating system executes the `init.d` shutdown scripts, including the one for WordPress at `/etc/init.d/bitnami`. This script also handles the WordPress startup when the virtual machine is started again. You can examine the service definition with the shutdown and startup configuration at `/etc/systemd/system/bitnami.service.`

This section creates the second custom operator, `SnapshotDiskOperator`. This operator takes a snapshot of the instance's persistent disk.

In the `gce_commands_plugin.py` file that you opened in the previous section, look at the `SnapshotDiskOperator` class:

no_sensor/plugins/gce_commands_plugin.py
(https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

m/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

The `SnapshotDiskOperator` class has the following methods:

- `__init__`: the class constructor. Similar to the constructor in the `StopInstanceOperator`class, but in addition to the project, zone, and instance name, this constructor receives the name of the disk to create the snapshot from. This is because an instance can have more than one persistent disk (/compute/docs/disks/#pdnumberlimits) attached to it.
- `generate_snapshot_name`: This sample method creates a simple unique name for each snapshot using the name of the instance, the date, and the time with a one-second granularity. Adjust the name to your needs, for example: by adding the disk name when multiple disks are attached to an instance, or by increasing the time granularity to support ad hoc snapshot creation requests.
- `execute`: the main operator method overridden from `BaseOperator`. When the Airflow worker executes it, it generates a snapshot name using the `generate_snapshot_name` method. Then it prints an info message and calls the Compute Engine API to create the snapshot with the parameters received in the constructor.

In this section, you create the third and final custom operator, `StartInstanceOperator`. This operator restarts a Compute Engine instance.

In the `gce_commands_plugin.py` file you previously opened, look at the `SnapshotDiskOperator` class toward the bottom of the file:

no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

om/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

The `StartInstanceOperator`class has the following methods:

- `__init__`: the class constructor. Similar to the constructor in the `StopInstanceOperator`class.

- `execute`: the main operator method overridden from `BaseOperator`. The difference from the previous operators is the invocation of the appropriate Compute Engine API to start the instance indicated in the constructor input parameters.

Earlier, you defined an Airflow plugin containing three operators. These operators define the tasks that form part of an Airflow workflow. The workflow presented here is simple and linear, but Airflow workflows can be complex Directed Acyclic Graphs.

This section creates the plugin class that exposes the three operators, then creates the DAG using these operators, deploys the DAG to Cloud Composer, and runs the DAG.

So far, the `gce_commands_plugin.py`file includes the start, snapshot, and stop operators. To use these operators in a workflow, you must include them in a plugin class.

1. Note the `GoogleComputeEnginePlugin` class at the bottom of the `gce_commands_plugin.py`file:

   no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

   m/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

   This class, which inherits from `AirflowPlugin`, gives the plugin the internal name `gce_commands_plugin` and adds the three operators to it.

2. Close the `gce_commands_plugin.py` file.

The DAG defines the workflow that Airflow executes. For the DAG to know which disk to back up, you need to define a few variables: which Compute Engine instance the disk is attached to, the zone the instance is running on, and the project where all the resources are available.

You could hard-code these variables in the DAG source code itself, but it's a best practice to define them as Airflow variables (https://airflow.apache.org/concepts.html#variables). This way, any configuration changes can be managed centrally and independently from code deployments.

Define the DAG configuration:

1. In Cloud Shell, set the location of your Cloud Composer environment:

   The location is the Compute Engine region where the Cloud Composer environment is located, for example: `us-central1` or `europe-west1`. It was set at the time of environment creation and is available in the Cloud Composer console page. (https://console.cloud.google.com/composer)

2. Set the Cloud Composer environment name:

   The `--format` parameter (/sdk/gcloud/reference/topic/formats) is used to select only the `name` column from the resulting table. You can assume that only one environment has been created.

3. Create the `PROJECT` variable in Airflow using the name of the current Google Cloud project:

   Where:

   - `gcloud composer environments run` is used to run Airflow CLI commands.

- The `variables` Airflow command sets the `PROJECT` Airflow variable to the value returned by `gcloud config`

4. Create the `INSTANCE` variable in Airflow with the name of the WordPress instance:

   This command uses the `--filter` (/sdk/gcloud/reference/topic/filters) parameter to select only the instance whose `name` matches a regular expression containing the string `wordpress`. This approach assumes that there is only one such instance, and that your instance and disk have "wordpress" as part of their name, which is true if you accepted the defaults.

5. Create the `ZONE` variable in Airflow using the zone of the WordPress instance:

6. Create the `DISK` variable in Airflow with the name of the persistent disk attached to the WordPress instance:

7. Verify that the Airflow variables have been created correctly:

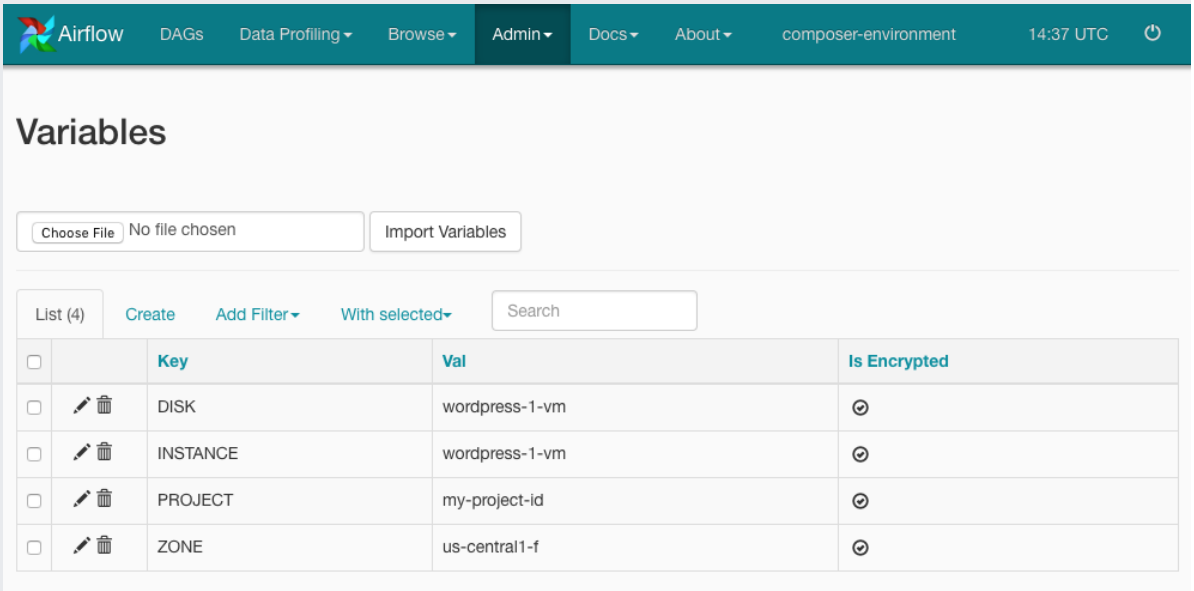
   a. In the Cloud Console, go to the Cloud Composer page.

   Go to the Cloud Composer page (https://console.cloud.google.com/composer)

   b. In the **Airflow web server** column, click the **Airflow** link. A new tab showing the Airflow web server main page opens.

c. Click **Admin** and then **Variables**.

The list shows the DAG configuration variables.



The DAG definition lives in a dedicated Python file. Your next step is to create the DAG, chaining the three operators from the plugin.

1. In Cloud Shell, use a text editor such as nano or vim to open the `backup_vm_instance.py` file:

2. Examine the imports at the top of the file:

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

Summarizing these imports:

- `DAG` is the Directed Acyclic Graph class (https://airflow.apache.org/concepts.html#dags) defined by Airflow.
- `DummyOperator` is used to create the beginning and ending no-op operators to improve the workflow visualization. In more complex DAGs, `DummyOperator` can be used to join branches (https://airflow.incubator.apache.org/concepts.html#branching) and to create SubDAGs (https://airflow.incubator.apache.org/concepts.html#subdags).
- The DAG uses the three operators that you defined in the previous sections.

3. Define the values of the parameters to be passed to operator constructors:

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

Where:

- `INTERVAL` defines how often the backup workflow runs. The preceding code specifies a daily recurrence using an Airflow cron preset. If you want to use a different interval, see the DAG Runs reference page (https://airflow.apache.org/scheduler.html#dag-runs). You could also trigger the workflow manually, independent of this schedule.
- `START_DATE` defines the point in time when the backups are scheduled to start. There is no need to change this value.

- The rest of the values are retrieved from the Airflow variables that you configured in the previous section.

4. Use the following code to create the DAG with some of the previously defined parameters. This code also gives the DAG a name and a description, both of which are shown in the Cloud Composer UI.

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

5. Populate the DAG with tasks, which are operator instances:

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

This code instantiates all the tasks needed for the workflow, passing the defined parameters to the corresponding operator constructors.

- The `task_id` values are the unique IDs that will be shown in the Cloud Composer UI. You use these IDs later to pass data between tasks.
- `retries` sets the number of times to retry a task before failing. For `DummyOperator` tasks, these values are ignored.
- `dag=dag` indicates that a task is attached to the previously created DAG. This parameter is only required in the first task of the workflow.

6. Define the sequence of tasks that comprise the workflow DAG:

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

7. Close the `gce_commands_plugin.py` file.

The workflow represented by the operator DAG is now ready to be run by Cloud Composer. Cloud Composer reads the DAG and plugin definitions from an associated Cloud Storage bucket (/composer/docs/concepts/cloud-storage). This bucket and the corresponding `dags` and `plugins` directories were automatically created when you created the Cloud Composer environment.

Using Cloud Shell, you can copy the DAG and plugin into the associated Cloud Storage bucket:

1. In Cloud Shell, get the bucket name:

There should be a single bucket with a name of the form: `gs://[REGION]-{ENVIRONMENT_NAME]-{ID}-bucket/.`

2. Execute the following script to copy the DAG and plugin files into the corresponding bucket directories:

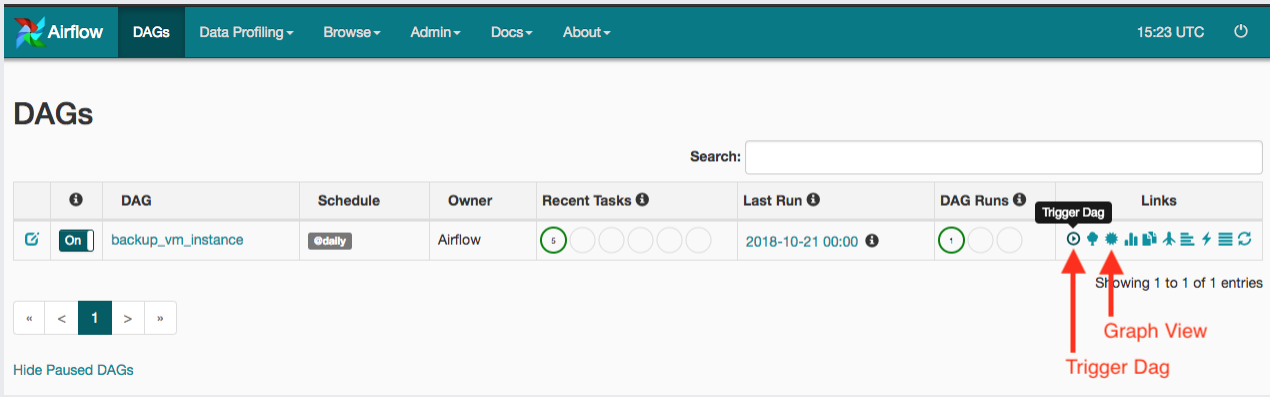
   The bucket name already includes a trailing slash, hence the double quotes around the `$BUCKET` variable.

3. In the Cloud Console, go to the Cloud Composer page.

   Go to the Cloud Composer page (https://console.cloud.google.com/composer)

4. In the **Airflow web server** column, click the **Airflow** link. A new tab showing the Airflow web server main page opens. Wait two to three minutes and reload the page. It might take a few cycles of waiting and then reloading for the page to be ready.

   A list showing the newly created DAG is shown, similar to the following:



   If there are syntax errors in the code, a message appears on top of the DAG table. If there are runtime errors, they are marked under **DAG Runs**. Correct any errors before continuing. The easiest way to do this is to recopy the files from the GitHub repo (https://github.com/GoogleCloudPlatform/composer-infra-python.git) into the bucket.

5. To see a more detailed stack trace, run the following command in Cloud Shell:

6. Airflow starts running the workflow immediately, shown under the column **Dag Runs**.

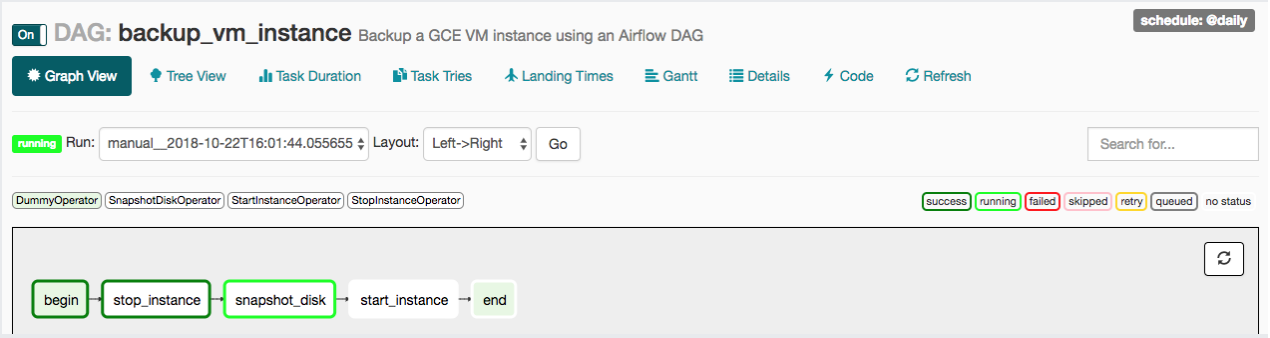
   The workflow is already underway, but if you need to run it again, you can trigger it manually with the following steps:

   a. In the **Links** column, click the first icon, **Trigger Dag**, marked with an arrow in the previous screenshot.

   b. In the pop-up confirming **Are you sure?**, click **OK**.

      In a few seconds, the workflow starts and a new run appears as a light green circle under **DAG Runs**.

7. In the **Links** column, click the Graph View icon, marked with an arrow in the previous screenshot.



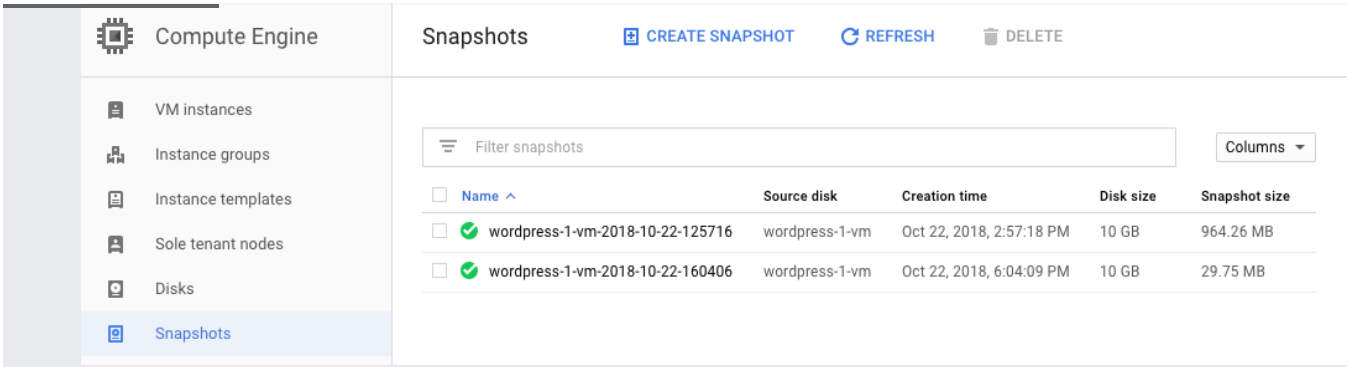
   The Graph View shows the workflow, the successfully executed tasks with a dark green border, the task being executed with a light green border and the pending tasks with no border. You can click the task to view logs, see its details, and perform other operations.

8. To follow the execution along, periodically click the refresh button at the top-right corner.

   Congratulations! You completed your first Cloud Composer workflow run. When the workflow finishes, it creates a snapshot of the Compute Engine instance persistent disk.

9. In Cloud Shell, verify that the snapshot has been created:

   Alternatively, you can use the Cloud Console menu to go to the Compute Engine Snapshots (https://console.cloud.google.com/compute/snapshots) page.



One snapshot should be visible at this point. Subsequent workflow runs, triggered either manually or automatically following the specified schedule, will create further snapshots.

Snapshots are incremental (/compute/docs/disks/create-snapshots). The size of the first snapshot is the largest because it contains all the blocks from the Persistent Disk in compressed form. Successive snapshots only contain the blocks that were changed from the previous snapshot, and any references to the unchanged blocks. So subsequent snapshots are smaller than the first one, take less time to produce, and cost less.

If a snapshot is deleted, its data is moved into the next corresponding snapshot to keep the consistency of consecutive deltas being stored in the snapshot chain. Only when all snapshots are removed is all the backed-up data from the persistent disk removed.

When running the workflow, you might have noticed that it takes some time to complete each step. This wait is because the operators include a `sleep()` instruction at the end to give time to the Compute Engine API to finish its work before starting the next task.

This approach is not optimal, however, and can cause unexpected issues. For example, during snapshot creation the wait time might be too long for incremental snapshots, which means you're wasting time waiting for a task that has already finished. Or the wait time might be too short. This can cause the whole workflow to fail or to produce unreliable results because the instance is not fully stopped or the snapshot process is not done when the machine is started.

You need to be able to tell the next task that the previous task is done. One solution is to use Airflow Sensors, which pause the workflow until some criteria is met. In this case, the criterion is the previous Compute Engine operation finishing successfully.

When tasks need to communicate with each other, Airflow provides a mechanism known as XCom, or "cross-communication." XCom lets tasks exchange messages consisting of a key, a value, and a timestamp.

The simplest way to pass a message using XCom is for an operator to return a value from its `execute()` method. The value can be any object that Python can serialize using the pickle module (https://docs.python.org/3/library/pickle.html).

The three operators described in previous sections call the Compute Engine API. All these API calls return an Operation resource (/compute/docs/reference/rest/v1/zoneOperations) object. These objects are meant to be used to manage asynchronous requests such as the ones on the Airflow operators. Each object has a field `name` that you can use to poll for the latest state of the Compute Engine operation.

Modify the operators to return the `name` of the Operation resource object:

1. In Cloud Shell, use a text editor such as nano or vim to open the `gce_commands_plugin.py` file, this time from the `sensor/plugins` directory:

2. In the execute method of the `StopInstanceOperator`, notice how the following code:

   no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

   om/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

   has been replaced with this code:

sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

Where:

- The first line captures the return value from the API call into the `operation` variable.
- The second line returns the operation `name` field from the `execute()` method. This instruction serializes the name using pickle (https://docs.python.org/2/library/pickle.html#module-pickle) and pushes it into the XCom intra-task shared space. The value will later be pulled in last-in, first-out order.

If a task needs to push multiple values, it's possible to give XCom an explicit `key` by calling `xcom_push()` (https://airflow.apache.org/concepts.html#xcoms) directly instead of returning the value.

3. Similarly, in the execute method of the `SnapshotDiskOperator`, note how the following code:

no_sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

m/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

has been replaced with this code:

sensor/plugins/gce_commands_plugin.py
(https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

There are two unrelated names in this code. The first one refers to the snapshot name, and the second is the operation name.

4. Finally, in the execute method of the `StartInstanceOperator`, note how the following code:

no_sensor/plugins/gce_commands_plugin.py
(https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

m/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/plugins/gce_commands_plugin.py)

has been replaced with this code:

sensor/plugins/gce_commands_plugin.py
(https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

5. At this point, there should not be any calls to the `sleep()` method throughout the `gce_commands_plugin.py` file. Make sure this is true by searching the file for `sleep`. Otherwise, double-check the previous steps in this section.

   Since no calls to `sleep()` are made from the code, the following line was removed from the imports section at the top of the file:

6. Close the `gce_commands_plugin.py`file.

In the previous section, you modified each operator to return a Compute Engine operation name. In this section, using the operation name, you create an Airflow Sensor to poll the Compute Engine API for the completion of each operation.

1. In Cloud Shell, use a text editor such as nano or vim to open the `gce_commands_plugin.py` file, making sure you use the `sensor/plugins` directory:

   Note the following line of code at the top of the import section, just below the `from airflow.models import BaseOperator` line:

   All sensors are derived from the `BaseSensorOperator` class, and must override its `poke()` method.

2. Examine the new `OperationStatusSensor` class:

   sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

).com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

The class `OperationStatusSensor` has the following methods:

- `__init__`: the class constructor. This constructor takes similar parameters to the ones for the Operators, with one exception: `prior_task_id`. This parameter is the ID of the previous task.
- `poke`: the main sensor method overridden from `BaseSensorOperator`. Airflow calls this method every 60 seconds until the method returns `True`. Only in that case are downstream tasks allowed to run.

You can configure the interval for these retries by passing the `poke_interval` parameter to the constructor. You can also define a `timeout`. For more information, see the BaseSensorOperator API reference (https://airflow.apache.org/_api/index.html#basesensoroperator).

In the implementation of the preceding `poke` method, the first line is a call to `xcom_pull()` (https://airflow.apache.org/concepts.html#xcoms). This method obtains the most recent XCom value for the task identified by `prior_task_id`. The value is the name of a Compute Engine Operation and is stored in the `operation_name` variable.

The code then executes the `zoneOperations.get()` method, passing `operation_name` as a parameter to obtain the latest status for the operation. If the status is `DONE`, then the `poke()` method returns `True`, otherwise `False`. In the former case, downstream tasks will be started; in the latter case, the workflow execution remains paused and the `poke()` method is called again after `poke_interval` seconds.

3. At the bottom of the file, note how the `GoogleComputeEnginePlugin` class has been updated to add `OperationStatusSensor` to the list of operators exported by the plugin:

sensor/plugins/gce_commands_plugin.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

o.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/plugins/gce_commands_plugin.py)

4. Close the `gce_commands_plugin.py`file.

After you create the sensor in the plugin, you can add it to the workflow. In this section, you update the workflow to its final state, which includes all three operators plus sensor tasks in between. You then run and verify the updated workflow.

1. In Cloud Shell, use a text editor such as nano or vim to open the `backup_vm_instance.py` file, this time from the `sensor/dags` directory:

2. In the imports section, notice that the newly created sensor is imported below the line `from airflow operators import StartInstanceOperator`:

3. Examine the lines following the `## Wait tasks` comment

sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/dags/backup_vm_instance.py)

ithub.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/dags/backup_vm_instance.py)

The code reuses `OperationStatusSensor` to define three intermediate "wait tasks". Each of these tasks waits for the previous operation to complete. The following parameters are passed to the sensor constructor:

- The `PROJECT`, `ZONE`, and `INSTANCE` of the WordPress instance, already defined in the file.

- `prior_task_id`: The ID of the task that the sensor is waiting for. For example, the `wait_for_stop` task waits for the task with ID `stop_instance` to be completed.

★ **Note:** The task IDs passed to the sensor constructor must match the IDs passed to the operator constructors for the corresponding Compute Engine tasks.

- `poke_interval`: The number of seconds that Airflow should wait in between retry calls to the sensor's `poke()` method. In other words, the frequency to verify whether `prior_task_id` is already done.
- `task_id`: The ID of the newly created wait task.

4. At the bottom of the file, note how the following code:

no_sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

b.com/GoogleCloudPlatform/composer-infra-python/blob/master/no_sensor/dags/backup_vm_instance.py)

has been replaced with this code:

sensor/dags/backup_vm_instance.py (https://github.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/dags/backup_vm_instance.py)

ithub.com/GoogleCloudPlatform/composer-infra-python/blob/master/sensor/dags/backup_vm_instance.py)

These lines define the full backup workflow.

5. Close the `backup_vm_instance.py`file.

Now you need to copy the DAG and plugin from the associated Cloud Storage bucket:

1. In Cloud Shell, get the bucket name:

   You should see a single bucket with a name of the form: `gs://[REGION]-[ENVIRONMENT_NAME]-[ID]-bucket/`.

2. Execute the following script to copy the DAG and plugin files into the corresponding bucket directories:

   The bucket name already includes a trailing slash, hence the double quotes around the `$BUCKET` variable

3. Upload the updated workflow to Airflow:

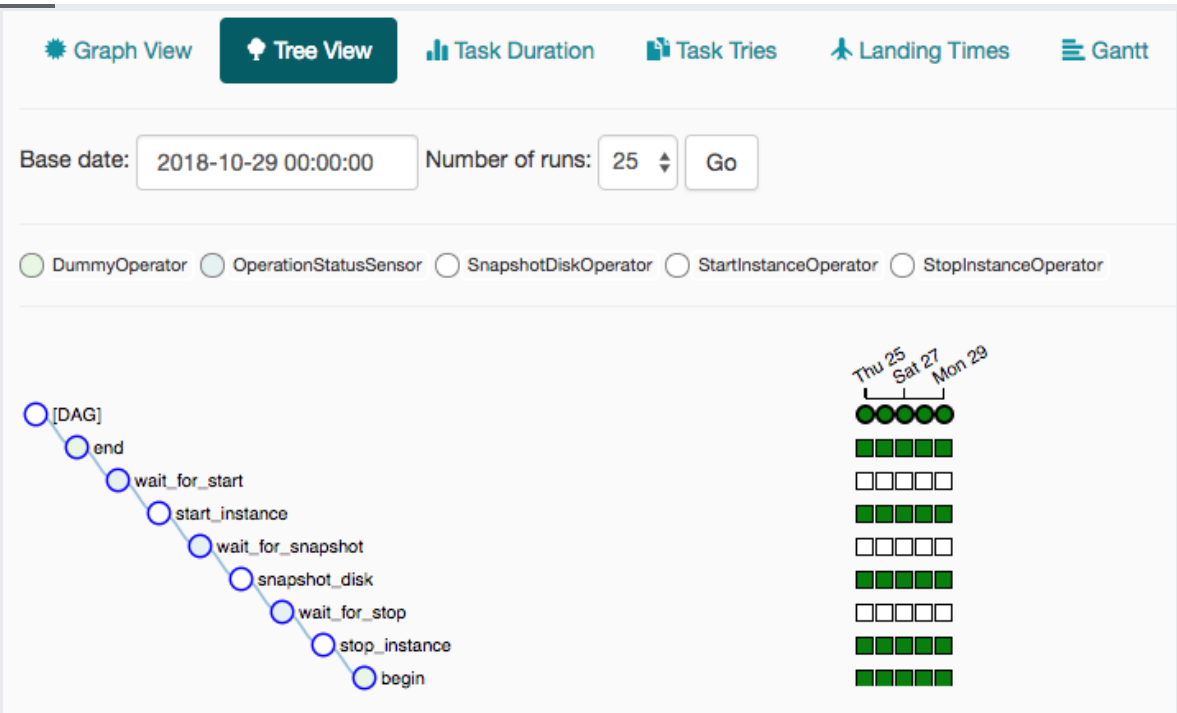
   a. In the Cloud Console, go to the Cloud Composer page.

   Go to the Cloud Composer page (https://console.cloud.google.com/composer)

   b. In the **Airflow** column, click the **Airflow web server** link to show the Airflow main page.

   c. Wait for two or three minutes until Airflow automatically updates the plugin and workflow. You might observe the DAG table becoming empty momentarily. Reload the page a few times until the **Links** section appears consistently.

   d. Make sure no errors are shown, and in the **Links** section, click **Tree View**.



On the left, the workflow is represented as a bottom-up tree. On the right, a graph of the task runs for different dates. A green square means a successful run for that specific task and date. A white square means a task that has never been run. Because you updated the DAG with new sensor tasks, all of those tasks are shown in white, while the Compute Engine tasks are shown in green.

e. Run the updated backup workflow:

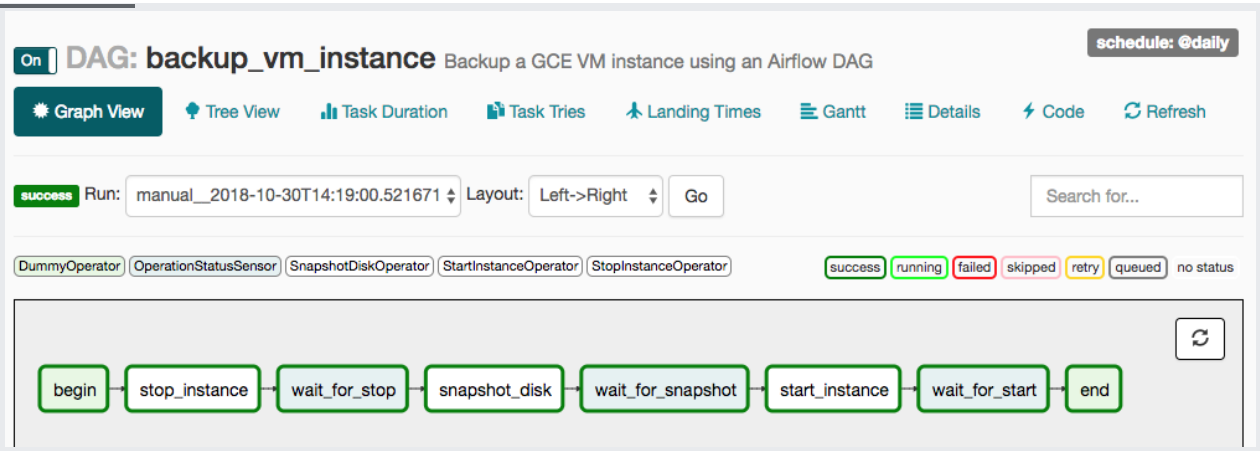
   i. In the top menu, click **DAGs** to go back to the main page.

   ii. In the **Links** column, click **Trigger DAG**.

   iii. In the pop-up confirming **Are you sure?**, click **OK**. A new workflowA run starts, appearing as a light green circle in the **DAG Runs** column.

f. Under **Links**, click the **Graph View** icon to observe the workflow execution in real time.

g. Click the refresh button on the right side to follow the task execution. Note how the workflow stops on each of the sensor tasks to wait for the previous task to finish. The wait time is adjusted to the needs of each task instead of relying on a hard-coded sleep value.



4. Optionally, during the workflow, go back to the Cloud Console, select the **Compute Engine** menu, and click **VM instances** to see how the virtual machine gets stopped and restarted. You can also click **Snapshots** to see the new snapshot being created.

You have now run a backup workflow that creates a snapshot from a Compute Engine instance. This snapshot follows best practices and optimizes the flow with sensors.

Having a snapshot available is only part of the backup story. The other part is being able to restore your instance from the snapshot.

To create an instance using a snapshot:

1. In Cloud Shell, get a list of the available snapshots:

   The output is similar to this:

2. Select a snapshot and create a standalone boot persistent disk from it. Replace the bracketed placeholders with your own values.

   Where:

   - `DISK_NAME` is a the name of the new standalone boot persistent disk.
   - `SNAPSHOT_NAME` is the selected snapshot from the first column of the previous output.
   - `ZONE` is the compute zone where the new disk will be created.

3. Create a new instance, using the boot disk. Replace `[INSTANCE_NAME]` with the name of the instance you want to create.

   With the two tags specified in the command, the instance is automatically allowed to receive incoming traffic on ports 443 and 80 because of the pre-existing firewall rules that were created for the initial WordPress instance.

   Take note of the new instance's External IP returned by the previous command.

4. Verify that WordPress is running on the newly created instance. On a new browser tab, navigate to the external IP address. The WordPress default landing page is shown.

5. Alternatively, create an instance using a snapshot from the console:

   a. In the Cloud Console, go to the Snapshots page:

      GO TO THE SNAPSHOTS PAGE (https://console.cloud.google.com/compute/snapshots)

   b. Click the most recent snapshot.

   c. Click **Create Instance**.

   d. In the **New VM Instance** form, click **Management, security, disks, networking, sole tenancy** and then **Networking**.

e. Add `wordpress-1-tcp-443` and `wordpress-1-tcp-80` to the Network tags (/vpc/docs/add-remove-network-tags) field, pressing enter after each tag. See above for an explanation of these tags.

f. Click **Create**.

A new instance based on the latest snapshot is created, and is ready to serve content.

6. Open the Compute Engine instances page (https://console.cloud.google.com/compute/instances) and take note of the new instance's external IP.
7. Verify that WordPress is running on the newly created instance. Navigate to the external IP on a new browser tab.

For more details, see Creating an instance from a snapshot (/compute/docs/instances/create-start-instance#createsnapshot).

Caution: Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such as an `appspot.com` URL, delete selected resources inside the project instead of deleting the whole project.

1. In the Cloud Console, go to the **Manage resources** page.

   Go to the Manage resources page (https://console.cloud.google.com/iam-admin/projects)

2. In the project list, select the project you want to delete and click **Delete** .
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

- Read about best practices for enterprise organizations (/docs/enterprise/best-practices-for-enterprise-organizations).
- Read about designing and implementing a disaster recovery plan (/solutions/dr-scenarios-planning-guide).
- Read more about Cloud Composer concepts (/composer/docs/concepts).
- Read more about Apache Airflow (https://airflow.apache.org/).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).