Bujar Bakiu · Oct 14, 2022 · 5 min read

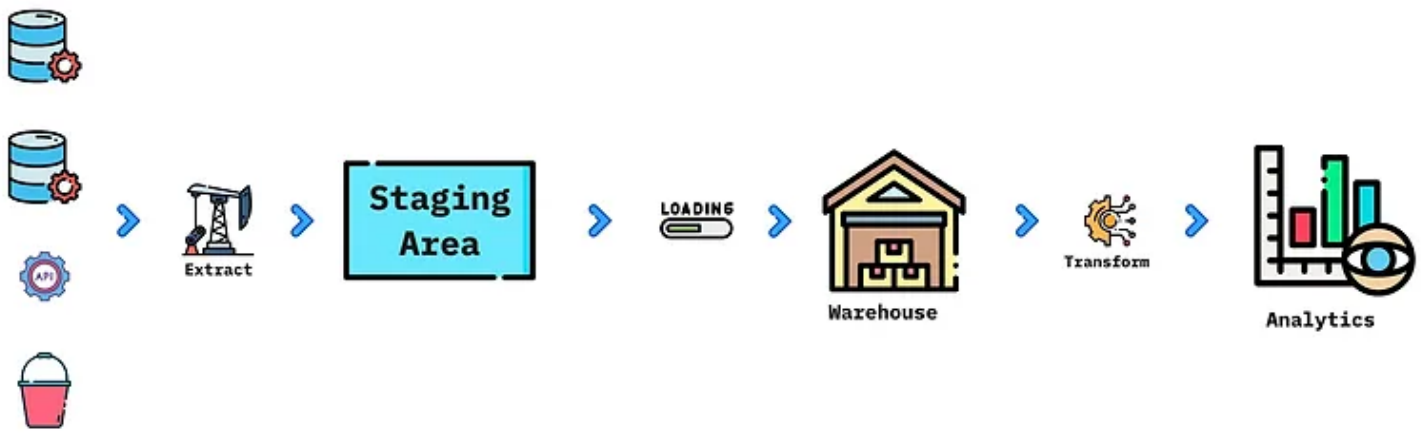
Dockerizing dbt Transformations for Managed Airflow: Docker, dbt, and GCP Cloud Composer

Airflow is one of the most popular pipeline orchestration tools out there. It has been around for more than 8 years, and it is used extensively in the data engineering world.
Popular cloud providers offer Airflow as a managed service e.g: GCP offers Cloud Composer and AWS offers Amazon Managed Workflows for Apache Airflow (MWAA). It is a very fast way to start an ETL (Extract, Transform and Load) pipeline and use it in a production environment.

Meanwhile, dbt (Data Build Tool) is exclusively focused on the Transformation step (the T in ETL). Its popularity has grown significantly and is now defacto the default tool for building powerful and flexible SQL transformations.

Often ETL process is implemented as an ELT (Extract, Load, and Transform) process. First, the data is extracted (or collected) from various sources e.g: databases, APIs, or files of different formats, and stored in a staging area. This

could be a data lake in the form of an S3 bucket. As a second step, the data is loaded in a data warehouse (DWH). Typical data warehouses are Redshift from AWS, BigQuery from GCP, or Snowflake. With the rapid adoption of cloud technologies, storage has become cheaper. Therefore loading the data in the warehouse early is no longer as costly. The last step is Transformations, where the data is aggregated, and adapted to the requirements of the Analytics teams or other stakeholders.



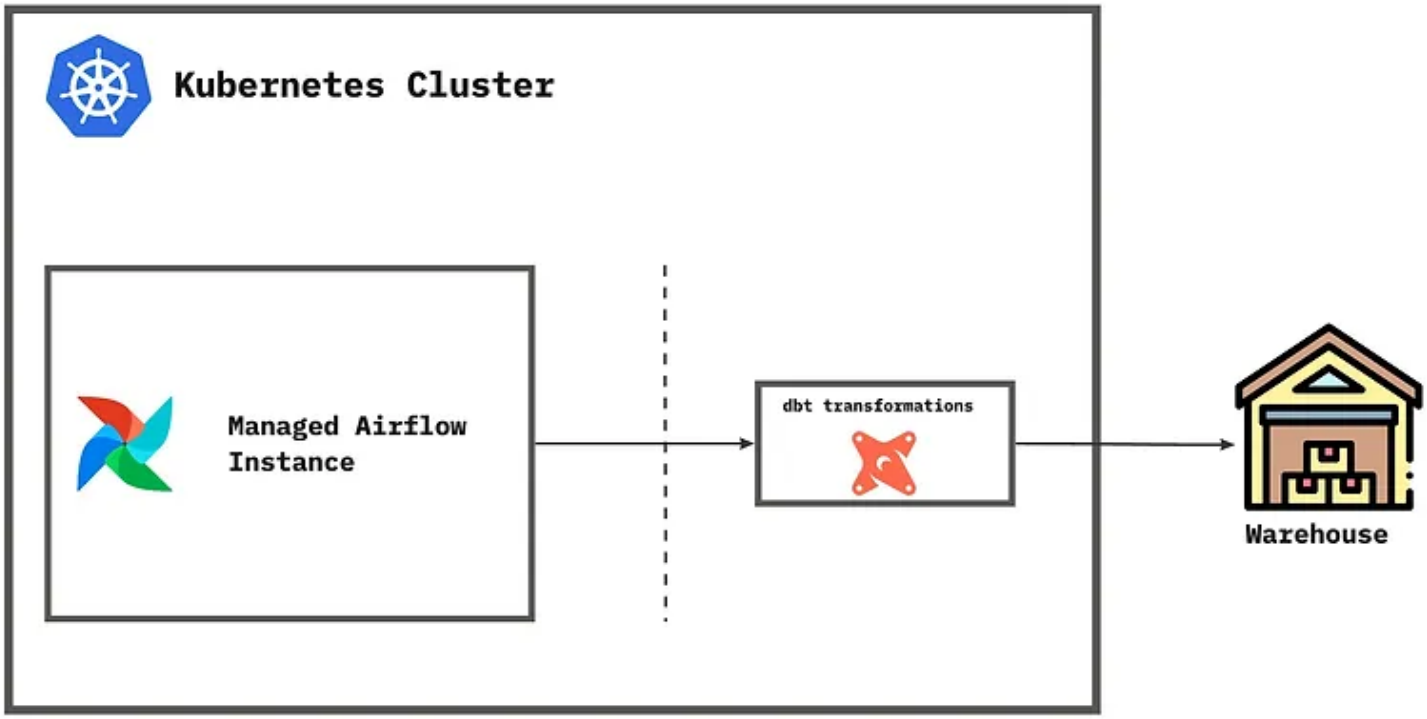
ELT process

The goal of this post is to show how *dbt* transformations can be integrated into a *managed Airflow instance*. The main challenge we have faced in our projects with managed Airflow instances and dbt is the process of *resolving dependency conflicts*. Airflow dependencies, dbt dependencies, and the default packages pre-installed in the managed Cloud Composer instance do not always work well together. Finding the right setup for them to work is time-consuming and can be obsolete when one dependency changes.
Another limitation of managed Airflow instances is the versions of Python (and Airflow), that are available. They are managed by the cloud provider and are limited in what they support. It takes a while from the moment one version is released until it is available to be used. Similar issues are present in AWS MWAA as well.

Source Code: *GitHub repo: https://github.com/data-max-hq/dbt-docker*

Solution shortly

The idea that we came up with is to run the dbt transformations in an isolated environment e.g: a Kubernetes pod, and run them as independent workloads.



Isolating dbt transformations in a separate pod in Kubernetes

This seems to be a popular idea from other teams that faced similar challenges.
In addition to the obvious environment isolation advantage, this solution provides the added benefit of being agnostic to the underlying infrastructure, be it a managed Kubernetes cluster by a cloud provider or a cluster in on-premise infrastructure.

But first, a bit of dbt

dbt is a tool focused on the Transformations in ELT. Before dbt, SQL scripts were stored in random files that were not very flexible. dbt brings best practices to SQL transformations. It allows for powerful SQL data modeling, that is flexible and version controlled. In addition, it brings testing and quality check for the data.

Major benefits of integrating dbt in the ELT pipeline include:

a. Reusability - it is no different than calling a function with different parameters.
b. History - with source code version control, it allows for keeping track of the changes over time.
c. Easily document the sources, sinks, and lineage of the data.
d. Include testing and quality check for the data.
e. Keeps track of the volume of data inserted, updated, or deleted.

Why containerize dbt

In order to run the transformations in an isolated environment, e.g: a pod in a Kubernetes cluster, we need to containerize them. This way the transformations are modularised and independent from the other components in the system's architecture.

How to build and test transformations

Starting a dbt project is pretty easy:

```
$ dbt init <your_project_name>
```

This command will create the directory `<your_project_name>` and also create the file `~/.dbt/profiles.yml` somewhere in your local system. To generalize this a bit we came up with the idea of having the `profiles` in the project directory. To achieve this:

```
$ cd <your_project_name>
$ mkdir profiles
$ cd profiles
$ touch profiles.yml
$ cd ..
```

After creating the profiles.yml file, fill it with the correct data based on dbt's documentation.

> What are dbt profiles?

> In short, dbt profiles define how dbt connects to the data warehouse. Using profiles, dbt knows how to connect to a Postgres, BigQuery, Redshift, or Snowflake warehouse.

> A profiles.yml file looks something like this:

```yaml
profile-name:
  target: dev
  outputs:
    dev:
      type: redshift
      host: dev-hostname.region.redshift.amazonaws.com
      user: [dev-username]
      password: [dev-password]
      port: 5439
      dbname: [dev-database name]
      schema: [dev-database schema]
      threads: 4
    prod:
      type: redshift
      host: prod-hostname.region.redshift.amazonaws.com
      user: [prod-username]
      password: [prod-password]
      port: 5439
      dbname: [prod-database name]
      schema: [prod-database schema]
      threads: 4
```

profiles.yml hosted with ❤ by GitHub view raw

> In this case, there are two targets, meaning two different warehouses. One is the dev warehouse and the other is the production one. Credentials for each are provided respectively. This is the case for a Redshift warehouse. For BigQuery or Snowflake the profiles file will look similar, but some other attributes might be required.

How to run a dbt project?

After creating the models, sources, seeds, and other entities testing it is rather easy.
One need only run the correct dbt command. Here are some examples:

```
$ dbt compile --profiles-dir ./profiles
$ dbt seed --profiles-dir ./profiles
$ dbt run --profiles-dir ./profiles
$ dbt test --profiles-dir ./profiles
```

How to wrap it in a Docker container

The docker image will look something similar to the picture below. The first layer will be a version of python, then some system dependencies are added. The third layer contains the dbt python packages required. Finally, layered on top of each other will be the dbt transformations grouped by project.

Make sure to add your project in the docker image by including the directory in this docker file. Lines 24-25 or 29-30 show how to include a dbt project in the docker image.

```
FROM python:3.9.13

# Update and install system packages
RUN apt-get update -y && \
  apt-get install --no-install-recommends -y -q \
  git libpq-dev && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# Set environment variables
ENV DBT_DIR /dbt

# Set working directory
WORKDIR $DBT_DIR

# Copy requirements
COPY requirements.txt .

# Install DBT
RUN pip install -U pip
RUN pip install -r requirements.txt

# Add dbt_project_1 to the docker image
COPY dbt_project_1 ./dbt_project_1
RUN ["dbt", "deps", "--project-dir", "./dbt_project_1"]

# Add dbt_project_2 to the docker image
# Repeat these line for every project in the repository
COPY dbt_project_2 ./dbt_project_2
RUN ["dbt", "deps", "--project-dir", "./dbt_project_2"]
```

Dockerfile hosted with ❤ by GitHub view raw

How to run dbt transformations from the docker image

```
$ docker build -t dbt-transformations:latest .
$ docker run dbt-transformations:latest dbt run --project-dir
./<your_project_name> --profiles-dir ./<your_project_name>/profiles
```

First, the docker image is built from the Dockerfile. Afterward, run the image followed by the dbt command. In the example above it will run the dbt transformations for <your_project_name>.

How to push dbt transformations to the cloud

To add the image to the cloud will need to push it to a container registry. It could be Docker Hub, AWS ECR, GCP Artifact Registry, Azure Container Registry, or another self-hosted option.

Using GitHub Actions is rather easy and usually, lots of prebuilt templates exist.
Here are examples of AWS ECR and GCP Artifact Registry.

```yaml
name: Build and Push to AWS ECR
on:
  push:
    branches:
      - main
  workflow_dispatch:

env:
  REPOSITORY_NAME: transformations-repository
  IMAGE_NAME: dbt-transformations
  AWS_REGION: eu-central-1

jobs:
  build-push:
    name: Build and push image to ECR
    runs-on: ubuntu-latest
    if: "!contains(github.event.head_commit.message, 'ci skip')"
    environment: production

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY
          aws-region: ${{ env.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build Docker Image
        run: docker build -t $IMAGE_NAME:latest .

      - name: Tag, and push image to Amazon ECR
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        run: |
          docker tag $IMAGE_NAME:latest $ECR_REGISTRY/$REPOSITORY_N
          docker tag $IMAGE_NAME:latest $ECR_REGISTRY/$REPOSITORY_N
          docker push $ECR_REGISTRY/$REPOSITORY_NAME/$IMAGE_NAME:la
          docker push $ECR_REGISTRY/$REPOSITORY_NAME/$IMAGE_NAME:$
```

```
docker push $ECR_REGISTRY/$REPOSITORY_NAME/$IMAGE_NAME:$
```

build-push-dbt-aws.yaml hosted with ❤ by GitHub view raw

```yaml
name: Build and Push to GCP Artifact registry
on:
  push:
    branches:
      - main
  workflow_dispatch:

env:
  PROJECT_ID: ${{ secrets.GCP_PROJECT_ID }}
  REPOSITORY_NAME: transformations-repository
  IMAGE_NAME: dbt-transformations
  GCP_REGION: europe-west1

jobs:
  build-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - uses: google-github-actions/auth@v0
        with:
          credentials_json: ${{ secrets.GCP_CREDENTIALS }}

      - name: Set up Cloud SDK
        uses: google-github-actions/setup-gcloud@v0

      - name: Configure Docker Client
        run: gcloud auth configure-docker $GCP_REGION-docker.pkg.de

      - name: Push Docker Image to GCP Artifact Registry
        run: |-
          gcloud builds submit --tag $GCP_REGION-docker.pkg.dev/$PR
          gcloud container images add-tag $GCP_REGION-docker.pkg.de
```

build-push-dbt-gcp.yaml hosted with ❤ by GitHub view raw

How to use the transformations in Airflow

KubernetesPodOperator will pull the image of the transformations from the container registry and run it in a Kubernetes Pod (DEH!). Transformations are run in the Pod using the docker image in the desired environment, connected to the DWH. Here is an example of what the code looks like:

```
migrate_data = KubernetesPodOperator(
        namespace='default',
        image='europe-west1-docker.pkg.dev/<project-id>/transformat
        cmds=["dbt", "run"],
        arguments=[
            "--project-dir", "./<project_dir>", "--profiles-dir", '
        ],
        name="dbt_transformations",
        task_id="dbt_transformations",
        get_logs=True
    )
```

kubernetes-pod-operator.py hosted with ❤ by GitHub view raw

Summary

This post gave a solution to running dbt transformations in a managed Airflow service from popular cloud providers. However, this solution can be used also in self-managed Airflow deployments.
Using KubernetesPodOperator to run dockerized transformations helps in resolving dependency conflicts, isolating the code, and modularizing the components of the infrastructure.

We would love to hear your feedback. Drop us a line at hi@data-max.io.