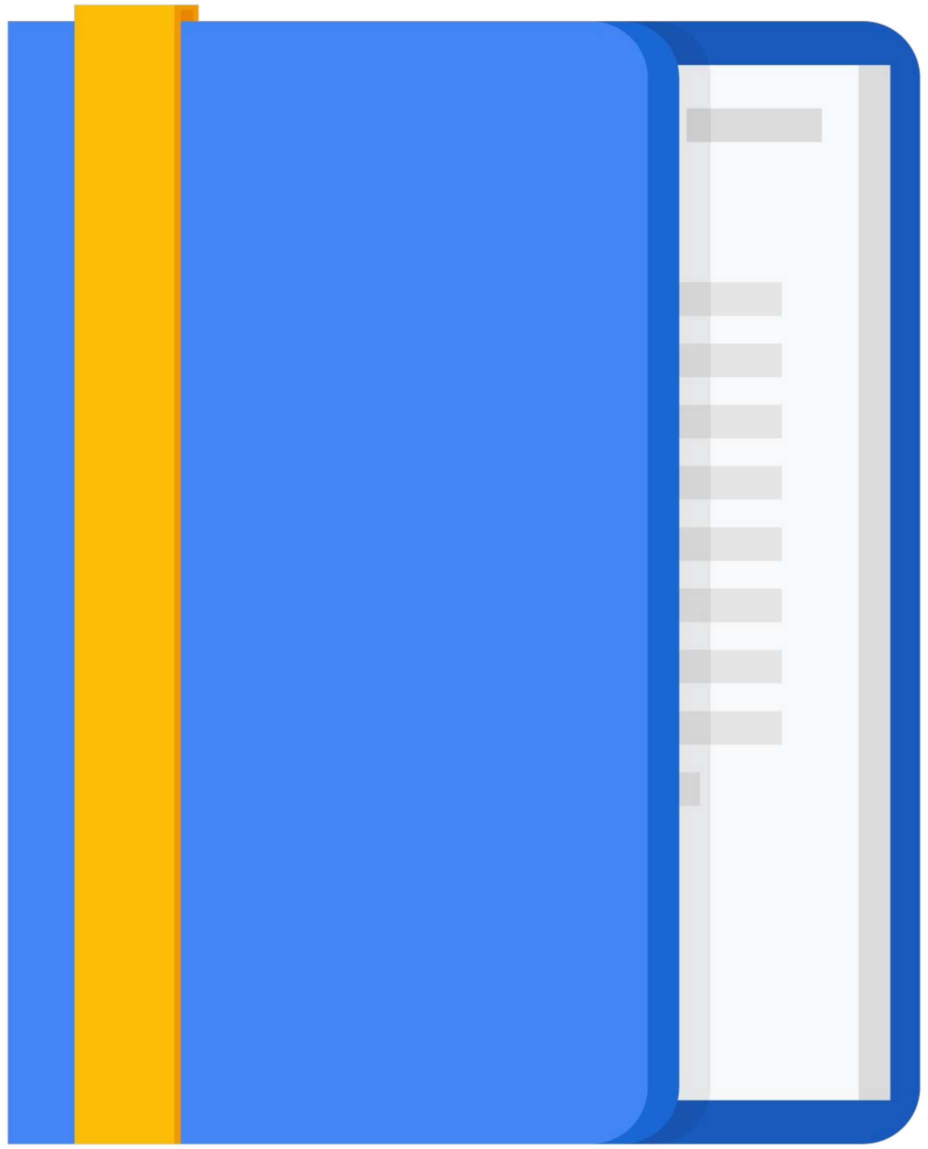
Continuous Training with Cloud Composer

Michael Abel
Data and Machine Learning Technical Trainer

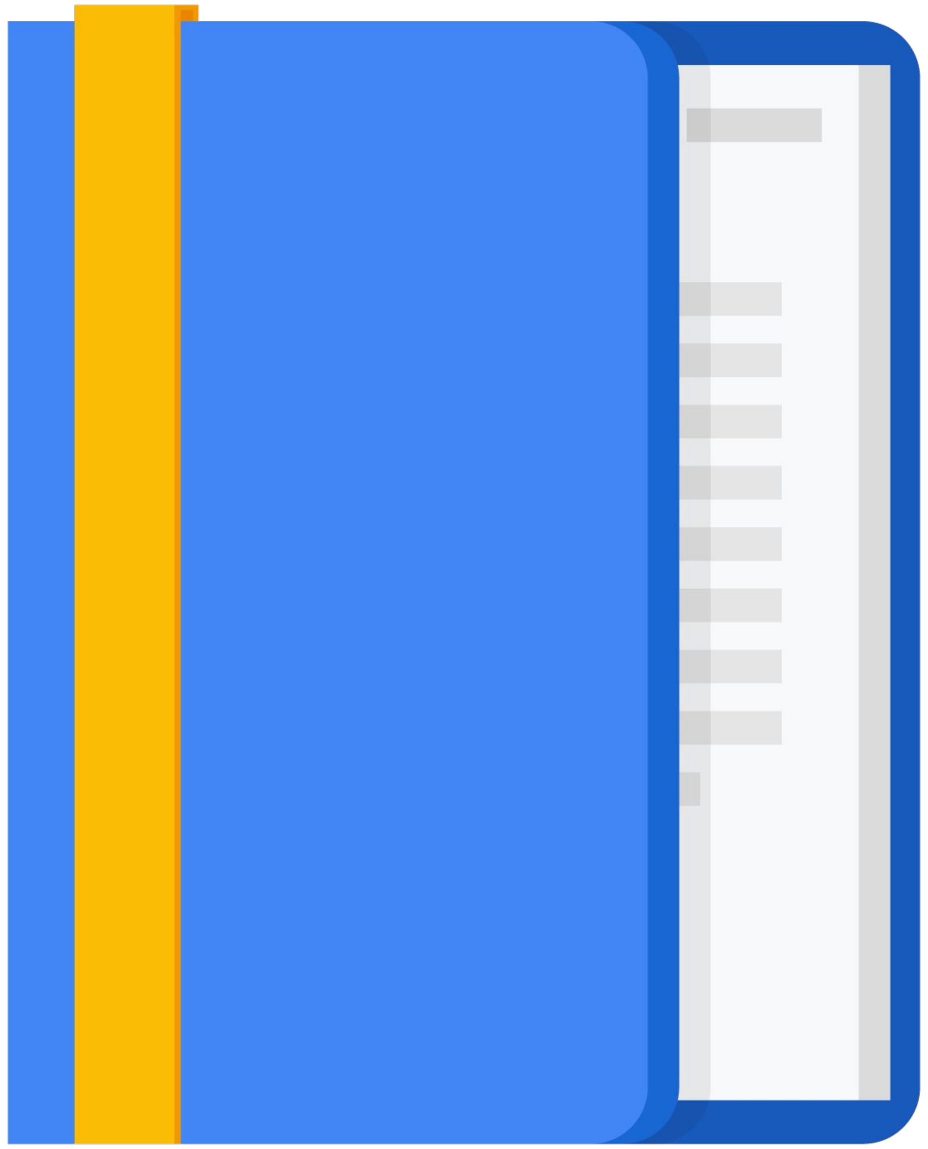
Agenda

- What is Cloud Composer?
- Core concepts of Apache Airflow
- Continuous training pipelines using Cloud Composer
- Apache Airflow, containers, and TFX



Agenda

- What is Cloud Composer?
- Core concepts of Apache Airflow
- Continuous training pipelines using Cloud Composer
- Apache Airflow, containers, and TFX



Apache Airflow

Apache Airflow is a popular open source tool for authoring, scheduling, and monitoring workflows.

Apache Airflow

Apache Airflow is a popular open source tool for authoring, scheduling, and monitoring workflows.

- Apache Airflow is a top-level project in the Apache Software Foundation.

Apache Airflow

Apache Airflow is a popular open source tool for authoring, scheduling, and monitoring workflows.

- Apache Airflow is a top-level project in the Apache Software Foundation.

- Workflows are authored as directed acyclic graphs (DAGs) and are configured in Python.

Why use Apache Airflow?

- Open source with an easy-to-use Python SDK

Why use Apache Airflow?

- Open source with an easy-to-use Python SDK
- Well-established interfaces (CLI, web server)

Why use Apache Airflow?

- Open source with an easy-to-use Python SDK
- Well-established interfaces (CLI, web server)
- Flexible scheduling and dependency management with retries

Why use Apache Airflow?

- Open source with an easy-to-use Python SDK
- Well-established interfaces (CLI, web server)
- Flexible scheduling and dependency management with retries
- Rich library of connectors

Why use Apache Airflow?

- Open source with an easy-to-use Python SDK
- Well-established interfaces (CLI, web server)
- Flexible scheduling and dependency management with retries
- Rich library of connectors
- Active community support

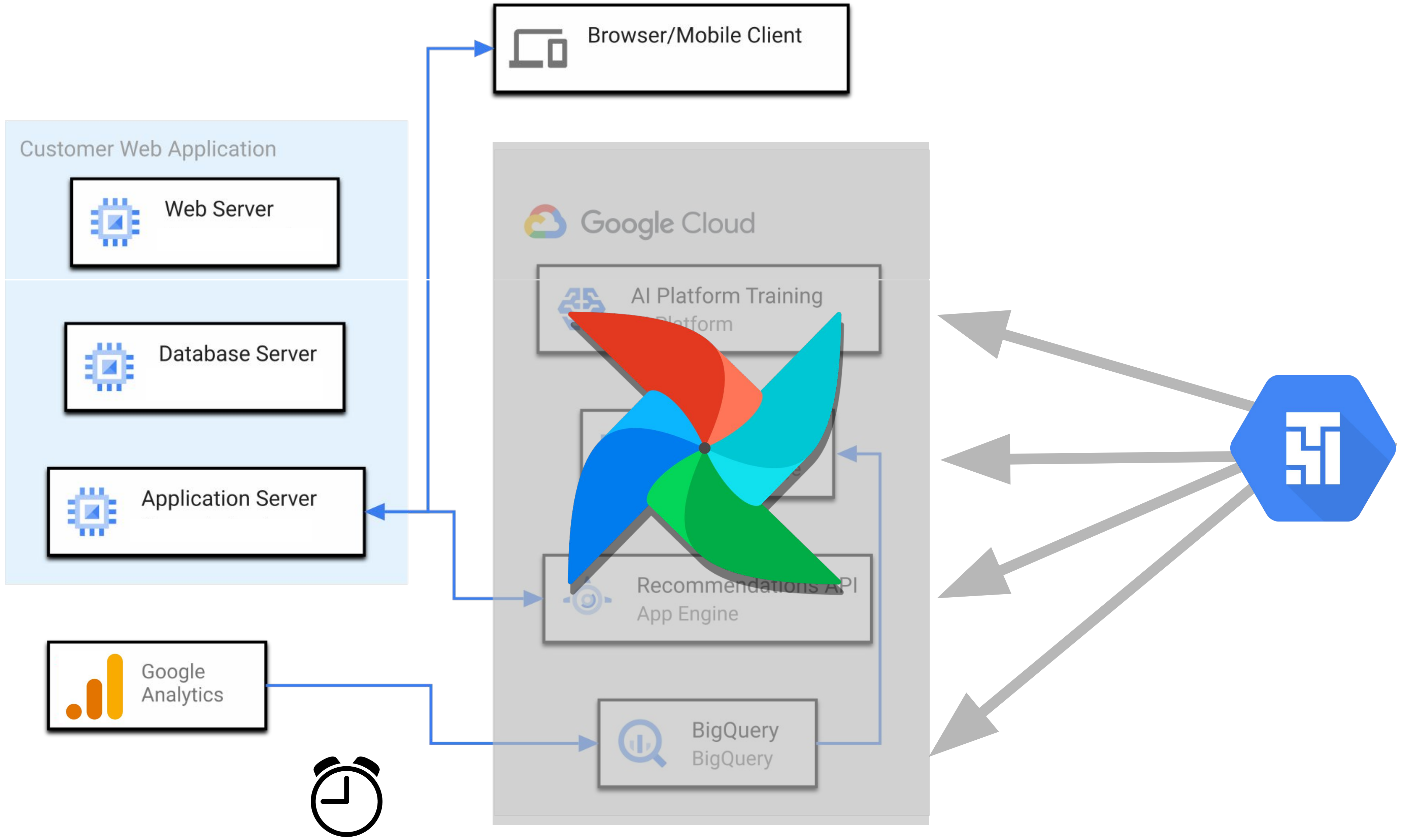
Why use Apache Airflow?

- Open source with an easy-to-use Python SDK
- Well-established interfaces (CLI, web server)
- Flexible scheduling and dependency management with retries
- Rich library of connectors
- Active community support

However, setup, management, and logging/debugging can be time-consuming and tedious...

Cloud Composer

Cloud Composer is a **managed** Apache Airflow service that helps you **create, schedule, monitor,** and **manage** workflows.



Why use Cloud Composer?

- Author end-to-end workflows on Google Cloud

Why use Cloud Composer?

- Author end-to-end workflows on Google Cloud
- Integrate with BigQuery, Cloud Storage, AI Platform, etc.

Why use Cloud Composer?

- Author end-to-end workflows on Google Cloud
- Integrate with BigQuery, Cloud Storage, AI Platform, etc.
- Secure your workflows across Google Cloud using tools such as Cloud IAM

Why use Cloud Composer?

- Author end-to-end workflows on Google Cloud
- Integrate with BigQuery, Cloud Storage, AI Platform, etc.
- Secure your workflows across Google Cloud using tools such as Cloud IAM
- Have your infrastructure fully managed by Google

Why use Cloud Composer?

- Author end-to-end workflows on Google Cloud
- Integrate with BigQuery, Cloud Storage, AI Platform, etc.
- Secure your workflows across Google Cloud using tools such as Cloud IAM
- Have your infrastructure fully managed by Google
- Explore Cloud Composer and Airflow logs through Cloud Operations Logging and Monitoring

Cloud Composer

Cloud
Composer

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

Cloud Composer

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

Pub/Sub
Ingest event streams from anywhere, at any scale, for simple, reliable, real-time stream analytics.

Cloud Composer

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

Pub/Sub
Ingest event streams from anywhere, at any scale, for simple, reliable, real-time stream analytics.

Cloud Composer

Dataflow
Simplified stream and batch data processing, with equal reliability and expressiveness.

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

Pub/Sub
Ingest event streams from anywhere, at any scale, for simple, reliable, real-time stream analytics.

Cloud Composer

Dataflow
Simplified stream and batch data processing, with equal reliability and expressiveness.

Dataproc
A faster, easier, more cost-effective way to run Apache Spark and Apache Hadoop.

Cloud Composer

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

Pub/Sub
Ingest event streams from anywhere, at any scale, for simple, reliable, real-time stream analytics.

Dataflow
Simplified stream and batch data processing, with equal reliability and expressiveness.

BigQuery
A fast, highly scalable, cost-effective, and fully managed data warehouse for analytics at any scale.

Dataproc
A faster, easier, more cost-effective way to run Apache Spark and Apache Hadoop.

Cloud Composer

Cloud Storage
Unified object storage for developers and enterprises.

AI Platform
Build superior machine learning models and deploy them into production.

Pub/Sub
Ingest event streams from anywhere, at any scale, for simple, reliable, real-time stream analytics.

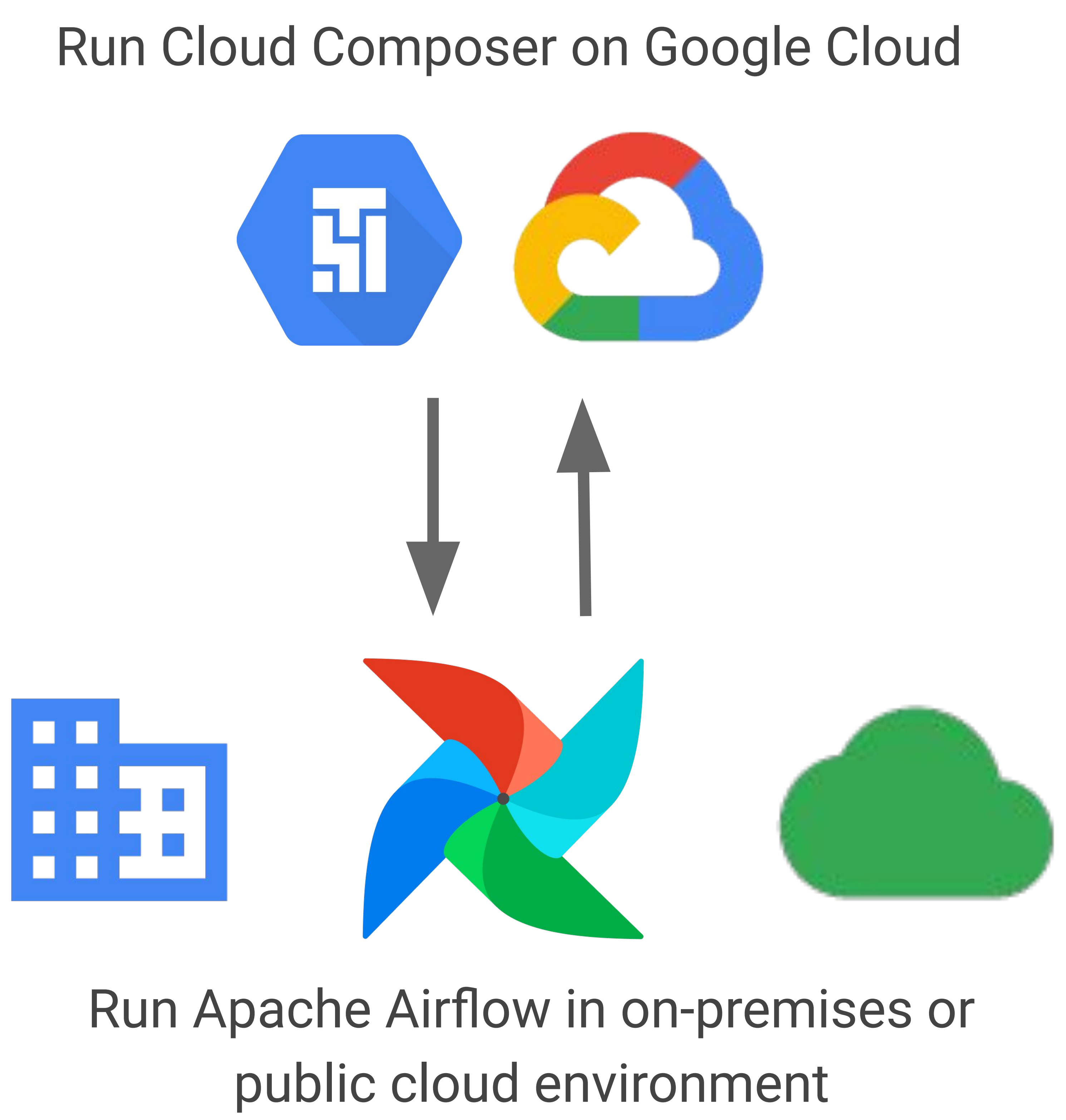
BigQuery
A fast, highly scalable, cost-effective, and fully managed data warehouse for analytics at any scale.

Cloud Composer

Dataflow
Simplified stream and batch data processing, with equal reliability and expressiveness.

Dataproc
A faster, easier, more cost-effective way to run Apache Spark and Apache Hadoop.

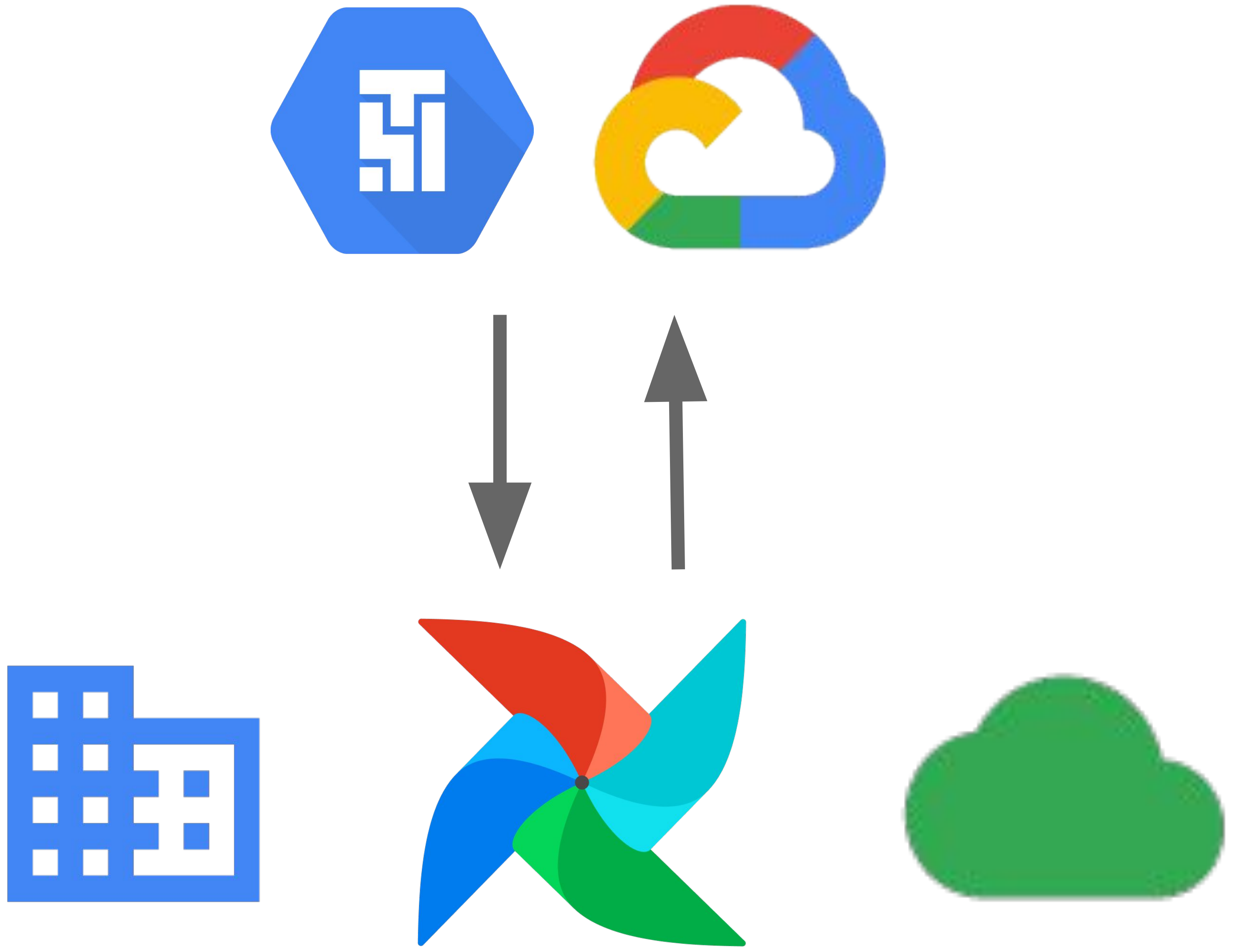
OSS = Portability



OSS = Portability

- Your workflows are as valuable as your data!

Run Cloud Composer on Google Cloud



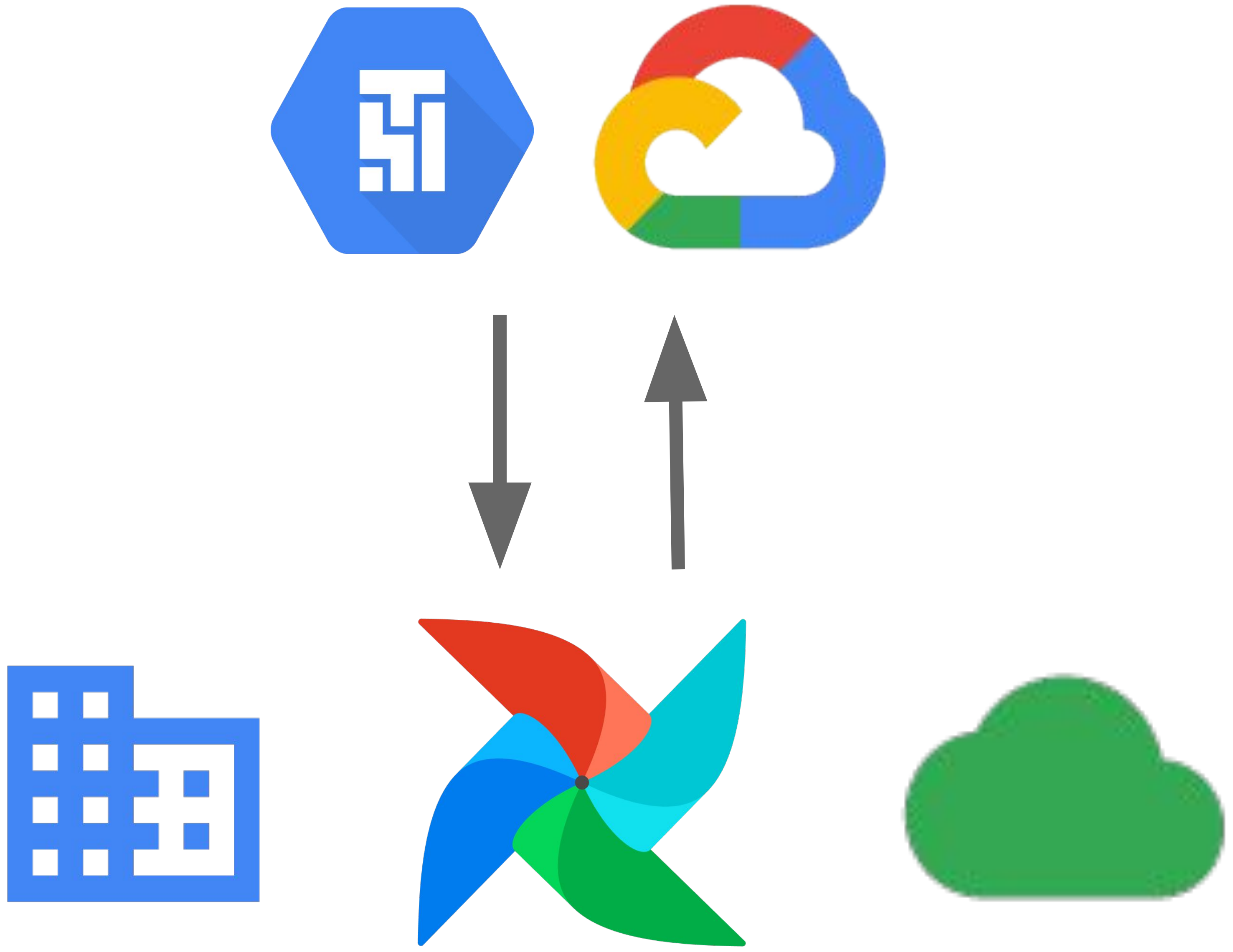
Run Apache Airflow in on-premises or public cloud environment

OSS = Portability

- Your workflows are as valuable as your data!

- Workflows represent hours of hard work by engineers.

Run Cloud Composer on Google Cloud

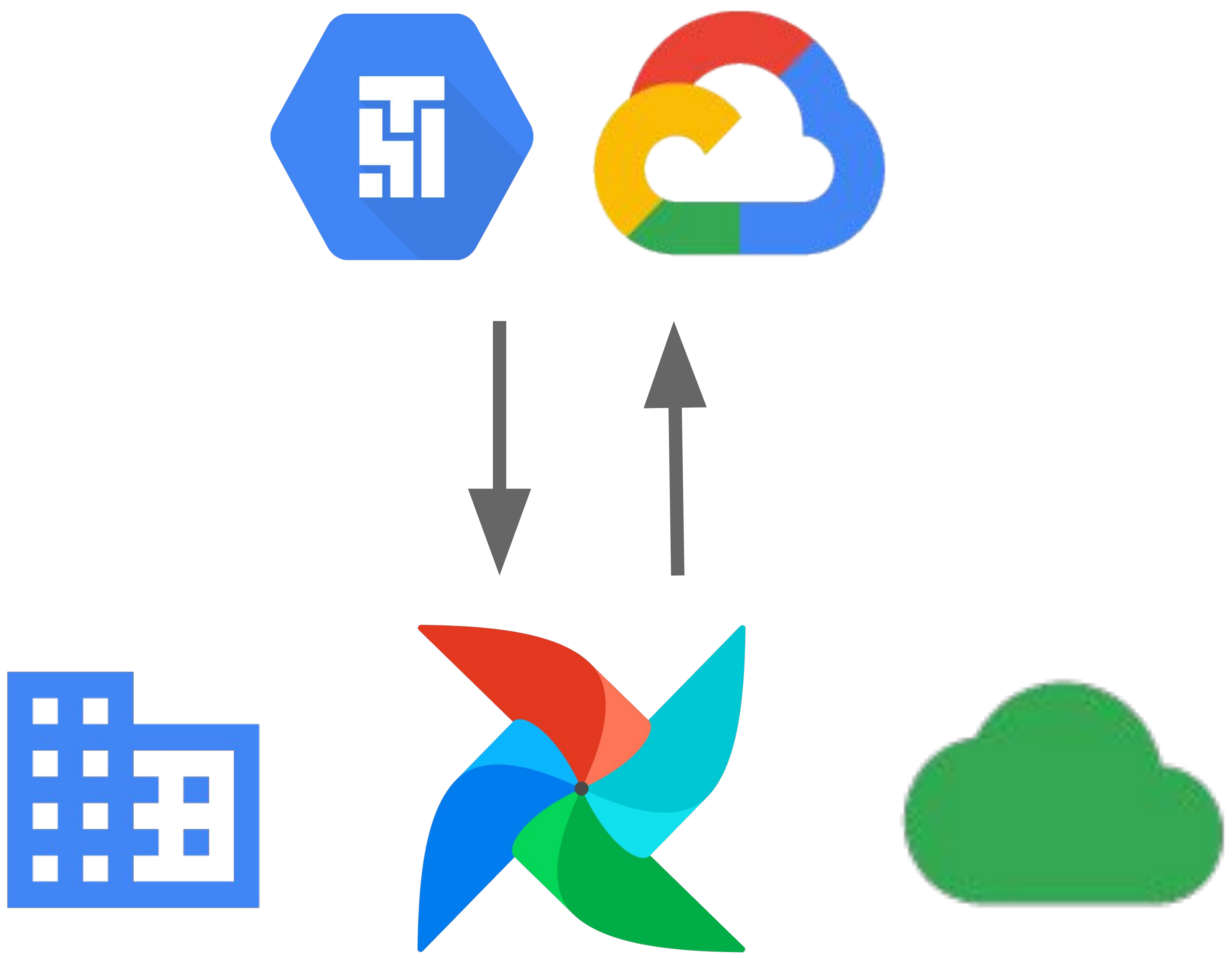


Run Apache Airflow in on-premises or public cloud environment

OSS = Portability

- Your workflows are as valuable as your data!
- Workflows represent hours of hard work by engineers.
- The ability to move your workflows from one platform to another helps ensure that effort is not lost when migrating.

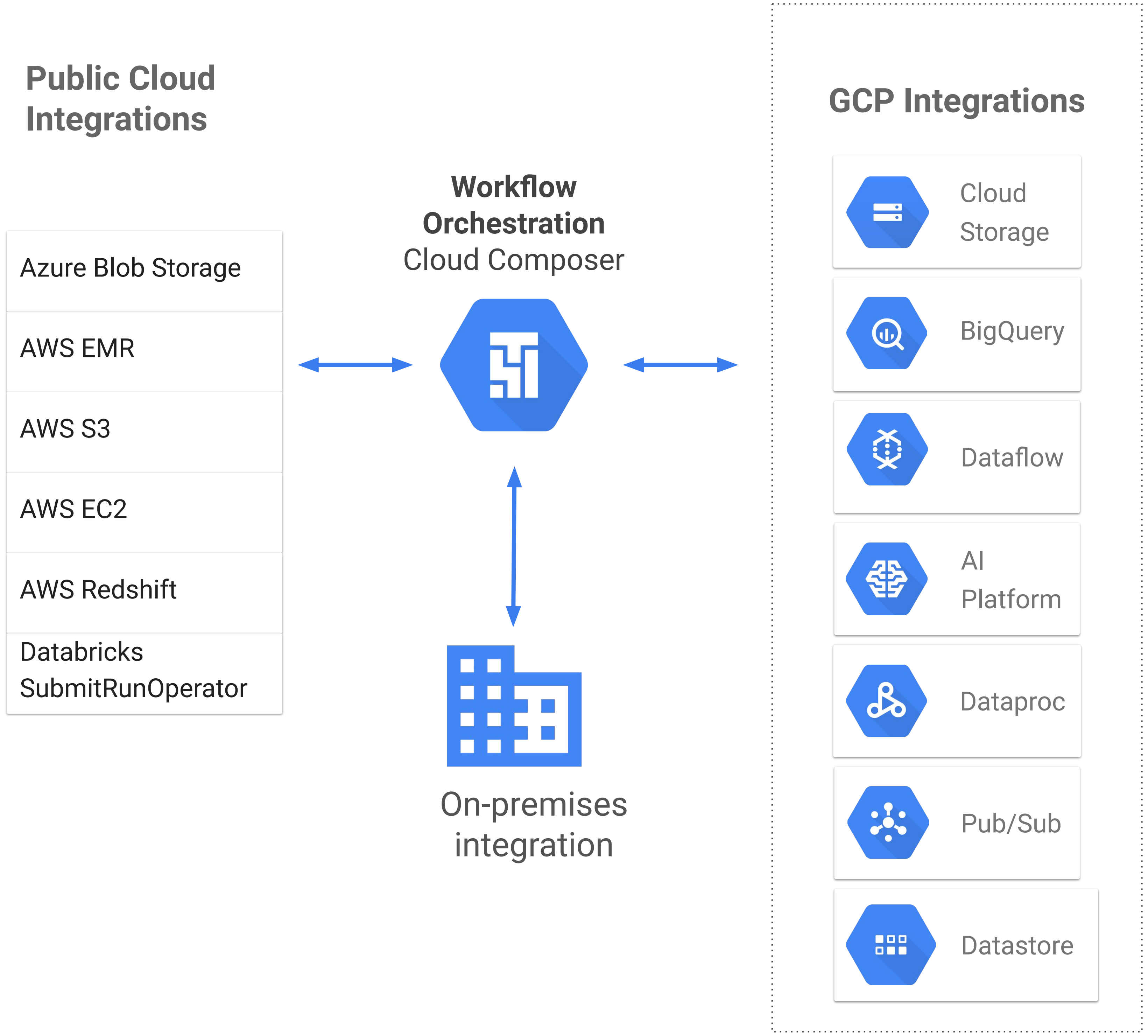
Run Cloud Composer on Google Cloud



Run Apache Airflow in on-premises or public cloud environment

Rich library of connectors

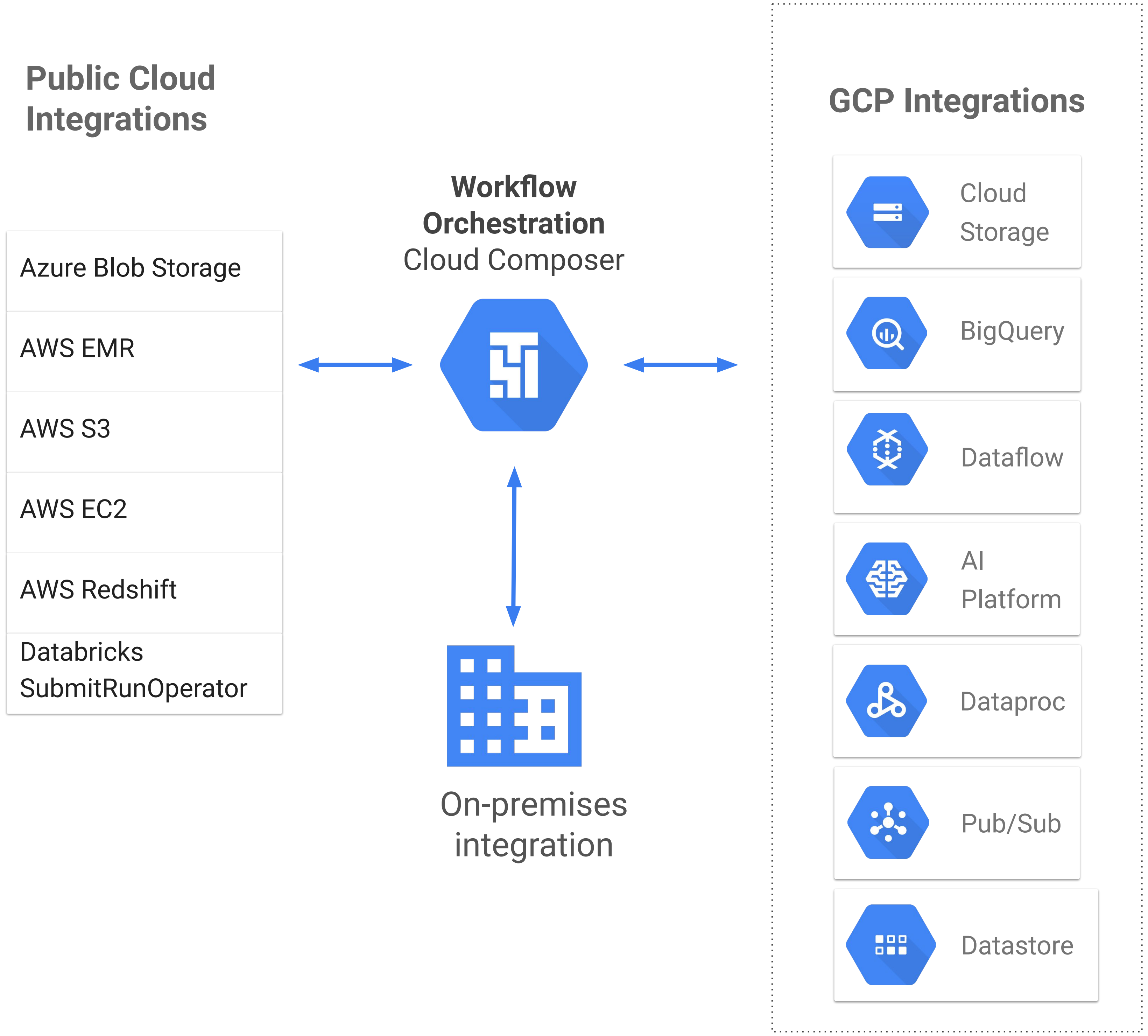
- Broad developer community ensures that connectors are built to a variety of services.



Rich library of connectors

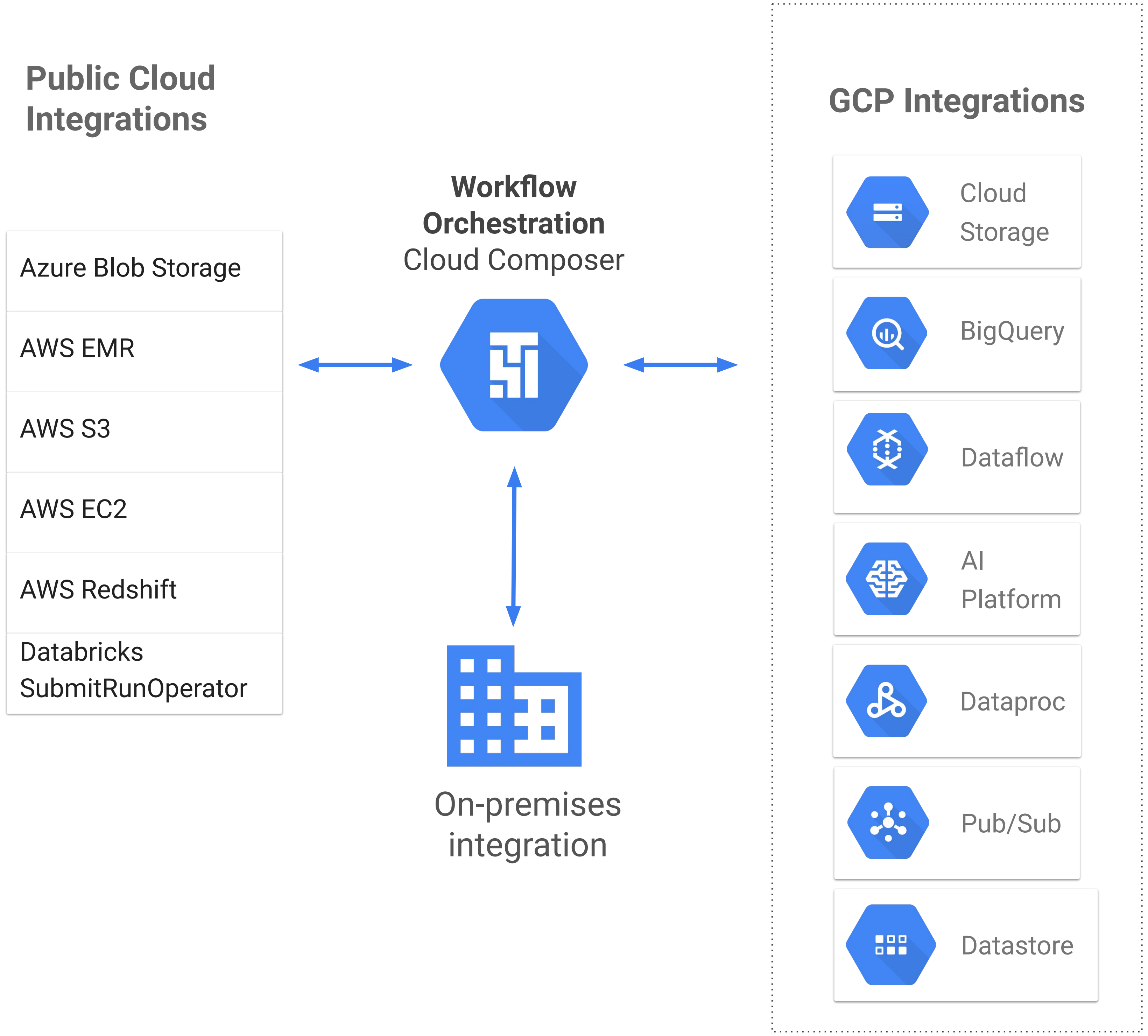
- Broad developer community ensures that connectors are built to a variety of services.

- Connect data across environments within a single workflow.



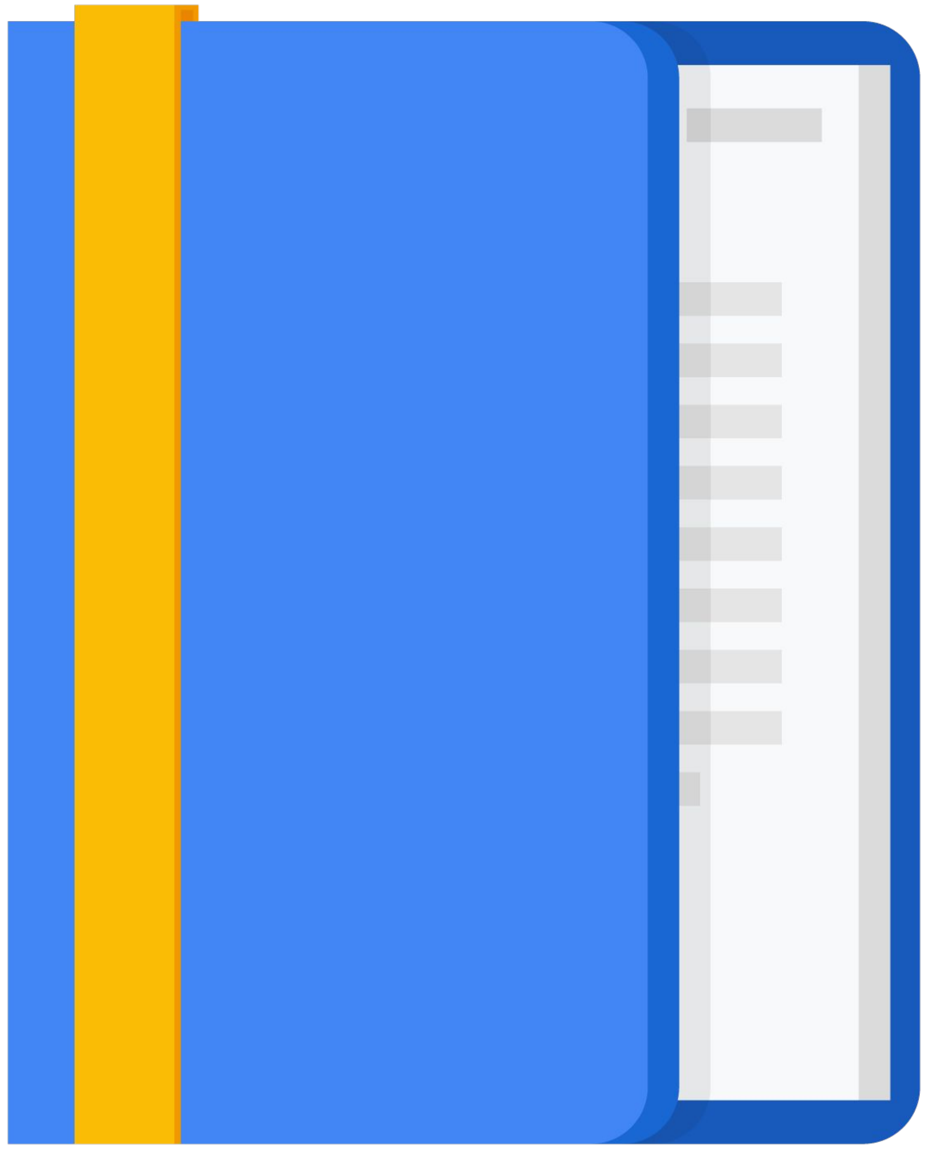
Rich library of connectors

- Broad developer community ensures that connectors are built to a variety of services.
- Connect data across environments within a single workflow.
- For example, you can use an AWS S3 bucket as a source for data, and Cloud Storage on Google Cloud as your sink.



Agenda

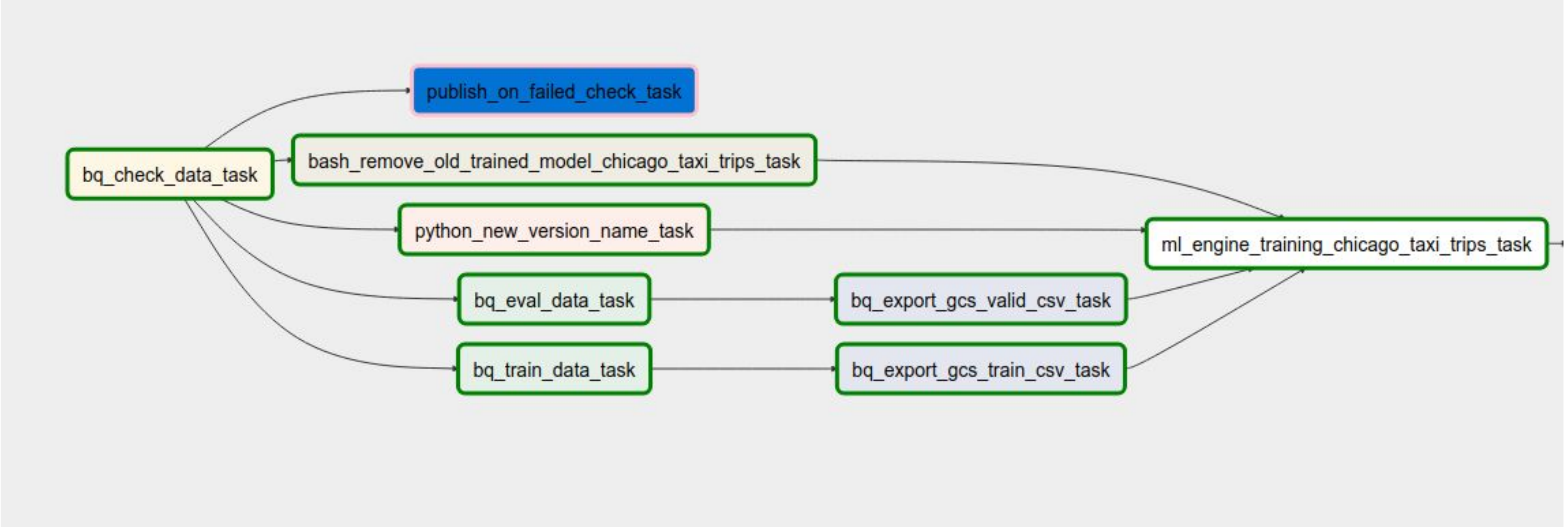
- What is Cloud Composer?
- Core concepts of Apache Airflow
- Continuous training pipelines using Cloud Composer
- Apache Airflow, containers, and TFX



Directed acyclic graphs (DAGs)

A DAG is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

In Airflow, a DAG is defined in a Python script, which represents the DAG structure (tasks and their dependencies) as code.



DAG runs

A **DAG run** is a physical instance of a DAG, containing **task instances** that run for a specific `execution_date`.

DAG runs

A **DAG run** is a physical instance of a DAG, containing **task instances** that run for a specific `execution_date`.

A DAG run is usually created by the Airflow scheduler, but it can also be created by an external trigger.

DAG runs

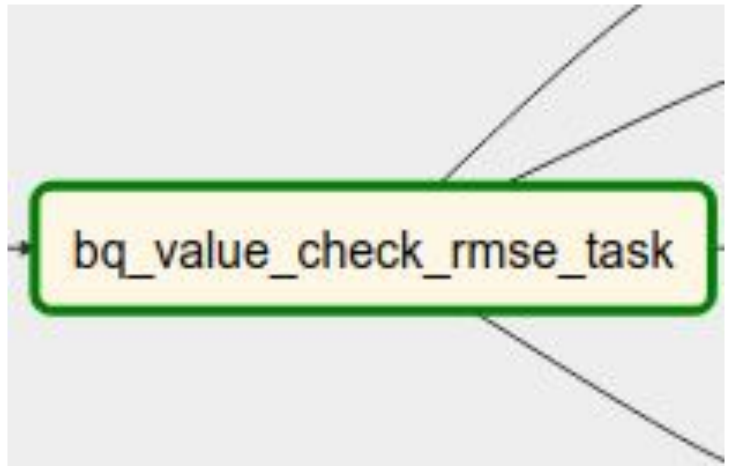
A **DAG run** is a physical instance of a DAG, containing **task instances** that run for a specific `execution_date`.

A DAG run is usually created by the Airflow scheduler, but it can also be created by an external trigger.

Multiple DAG runs for the same DAG can run concurrently, each with a different `execution_date`.

Tasks and operators

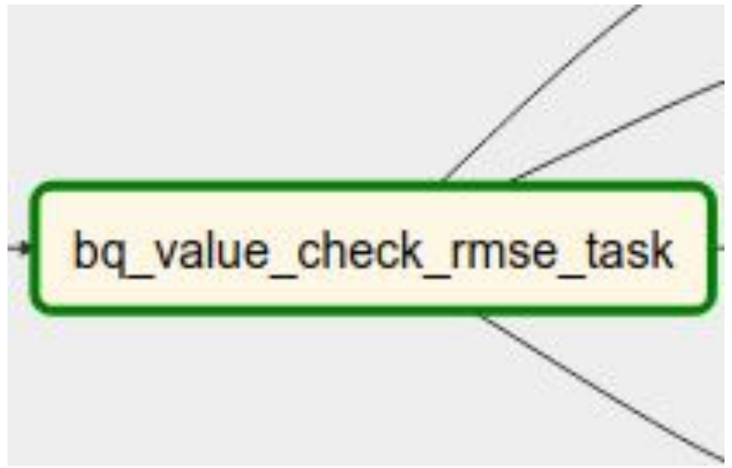
A task is represented by a node in your DAG and defines a unit of work in your workflow. Each task is an implementation of an operator.



Tasks and operators

A task is represented by a node in your DAG and defines a unit of work in your workflow. Each task is an implementation of an operator.

There are 3 main types of operators:

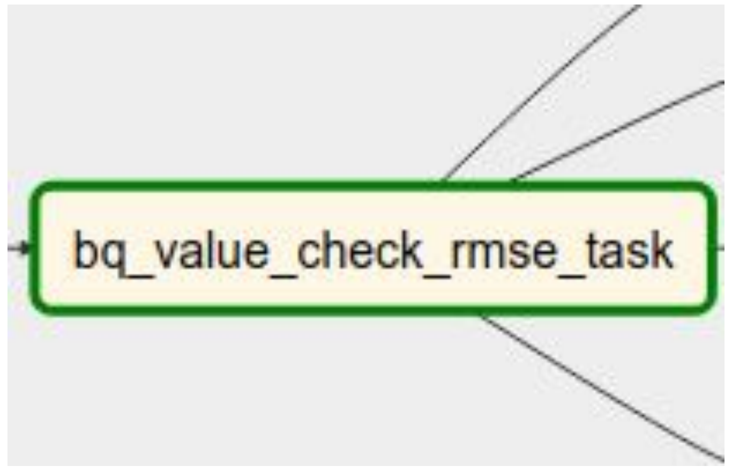


Tasks and operators

A task is represented by a node in your DAG and defines a unit of work in your workflow.
Each task is an implementation of an operator.

There are 3 main types of operators:

- Operators that perform an action or tell another system to perform an action

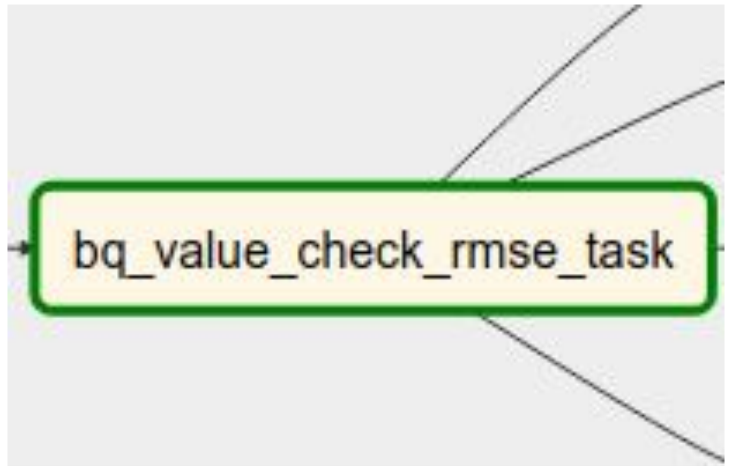


Tasks and operators

A task is represented by a node in your DAG and defines a unit of work in your workflow. Each task is an implementation of an operator.

There are 3 main types of operators:

- Operators that perform an action or tell another system to perform an action
- Transfer operators that move data from one system to another

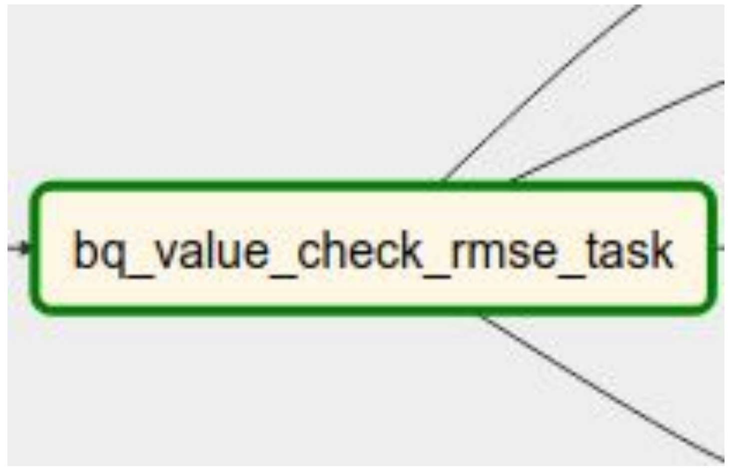


Tasks and operators

A task is represented by a node in your DAG and defines a unit of work in your workflow. Each task is an implementation of an operator.

There are 3 main types of operators:

- Operators that perform an action or tell another system to perform an action
- Transfer operators that move data from one system to another
- Sensors that will keep running until a specific criterion is met



Understanding Airflow DAGs

Airflow DAGs are Python scripts with 5 main sections

1. Imports
2. Arguments
3. Instantiation
4. Tasks
5. Dependencies

Imports

Import Python dependencies required for the workflow.

```
from airflow import DAG
from airflow.models import Variable

from airflow.contrib.operators.bigquery_operator import BigQueryOperator
from airflow.contrib.operators.mlengine_operator import MLEngineVersionOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.contrib.operators.pubsub_operator import PubSubPublishOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.trigger_rule import TriggerRule
```

Imports

Import Python dependencies required for the workflow.

```
from airflow import DAG
from airflow.models import Variable

from airflow.contrib.operators.bigquery_operator import BigQueryOperator
from airflow.contrib.operators.mlengine_operator import MLEngineVersionOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.contrib.operators.pubsub_operator import PubSubPublishOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.trigger_rule import TriggerRule
```

Imports

Import Python dependencies required for the workflow.

```
from airflow import DAG
from airflow.models import Variable

from airflow.contrib.operators.bigquery_operator import BigQueryOperator
from airflow.contrib.operators.mlengine_operator import MLEngineVersionOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.contrib.operators.pubsub_operator import PubSubPublishOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.trigger_rule import TriggerRule
```

Imports

Import Python dependencies required for the workflow.

```
from airflow import DAG
from airflow.models import Variable

from airflow.contrib.operators.bigquery_operator import BigQueryOperator
from airflow.contrib.operators.mlengine_operator import MLEngineVersionOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.contrib.operators.pubsub_operator import PubSubPublishOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.trigger_rule import TriggerRule
```

Arguments

Define default and DAG-specific arguments. You can define a dictionary of default parameters to be used when creating tasks.

```
DEFAULT_ARGS = {
    'owner': 'Google Cloud User',
    'depends_on_past': False,
    'start_date': datetime.datetime(2019, 12, 1),
    'email': ['example@email.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5)
}
```

Arguments

Define default and DAG-specific arguments. You can define a dictionary of default parameters to be used when creating tasks.

```
DEFAULT_ARGS = {
    'owner': 'Google Cloud User',
    'depends_on_past': False,
    'start_date': datetime.datetime(2019, 12, 1),
    'email': ['example@email.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5)
}
```

Instantiation

Name the DAG and set the `dag_id`, which serves as a unique identifier for your DAG.

You also configure the DAG schedule and other DAG settings via the default argument dictionary that you just defined.

```
with DAG(
    'chicago_taxi_dag',
     catchup=False,
     default_args=DEFAULT_ARGS,
     schedule_interval='@monthly') as dag:
```

Tasks

Airflow provides operators for many common tasks, including:

- **BashOperator:** Executes a bash command
- **PythonOperator:** Calls an arbitrary Python function

Airflow also provides operators for tasks on Google Cloud, including:

- BigQuery operators
- Cloud Storage operators
- Dataproc operators
- Dataflow operators
- Cloud Build operators
- AI Platform operators

```
bq_train_data_op = BigQueryOperator(
    task_id="bq_train_data_task",
    sql=sql_train,
    destination_dataset_table=...,
    write_disposition="WRITE_TRUNCATE",
    use_legacy_sql=False,
    dag=dag
    )
```

Dependencies

Operator relationships are set with the `set_upstream()` and `set_downstream()` methods. Note that this can be done with the Python bitshift operators `>>` and `<<`.

The following four statements are all functionally equivalent:

```
op1 >> op2        op1.set_downstream(op2)
op2 << op1        op2.set_upstream(op1)
```

The bitshift operators can also be used with lists; for example:

`op1 >> [op2, op3] >> op4` is equivalent to `op1 >> op2 >> op4`
`op1 >> op3 >> op4`

Creating and accessing environments

- Google Cloud Console
  - Create a Composer environment through the Console UI by navigating to Cloud Composer.

Creating and accessing environments

- Google Cloud Console
  - Create a Composer environment through the Console UI by navigating to Cloud Composer.

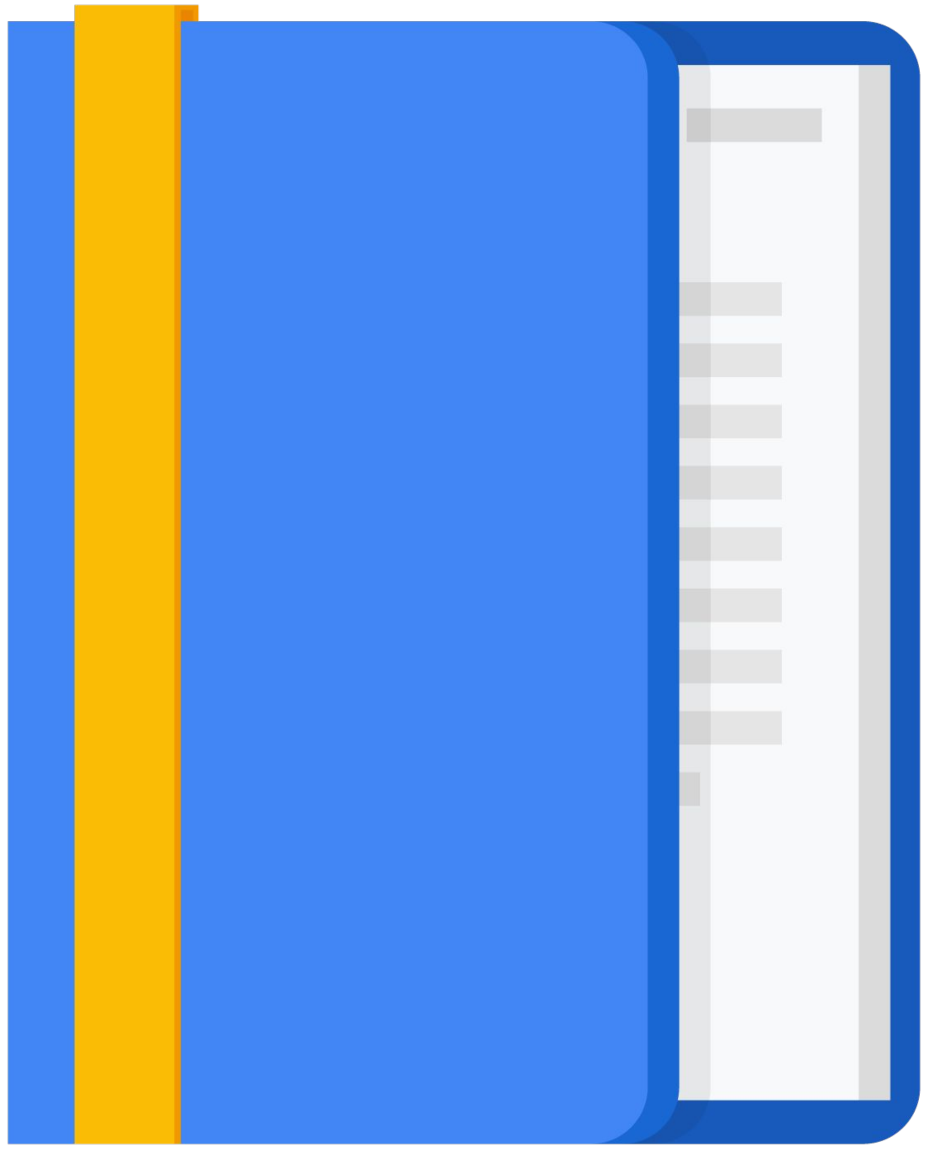
- Google Cloud SDK CLI
  - ```
    gcloud composer environments create $ENV_NAME \
        --location $REGION ...
    ```
    ```

Creating and accessing environments

- Google Cloud Console
  - Create a Composer environment through the Console UI by navigating to Cloud Composer.

- Google Cloud SDK CLI
  - ```
    gcloud composer environments create $ENV_NAME \
        --location $REGION ...
    ```

- Cloud Composer REST API
  - Construct an `environments.create` API request.
    ```

Cloud Composer buckets

Cloud Storage “folder”	Mapped Local Directory	Usage	Sync type
gs://{composer-bucket}/dags	/home/airflow/gcs/dags	DAGs and dependencies (e.g., SQL Queries)	Periodic 1-way rsync (workers/web-server)
gs://{composer-bucket}/plugins	/home/airflow/gcs/plugins	Airflow plugins (Custom Operators/Hooks, etc.)	Periodic 1-way rsync (workers/web-server)
gs://{composer-bucket}/data	/home/airflow/gcs/data	Workflow-related data	Cloud Storage FUSE (workers only)
gs://{composer-bucket}/logs	/home/airflow/gcs/logs	Airflow task logs (should only read)	Cloud Storage FUSE (workers only)

Agenda

- What is Cloud Composer?
- Core concepts of Apache Airflow
- Continuous training pipelines using Cloud Composer
- Apache Airflow, containers, and TFX



Continuous training pipelines

Continuous training pipelines

1

Data ingestion

Continuous training pipelines

1. Data ingestion
2. Data validation

Continuous training pipelines

1. Data ingestion
2. Data validation
3. Data preparation

Continuous training pipelines

1. Data ingestion
2. Data validation
3. Data preparation
4. Model training

Continuous training pipelines

1. Data ingestion
2. Data validation
3. Data preparation
4. Model training
5. Model evaluation

Continuous training pipelines

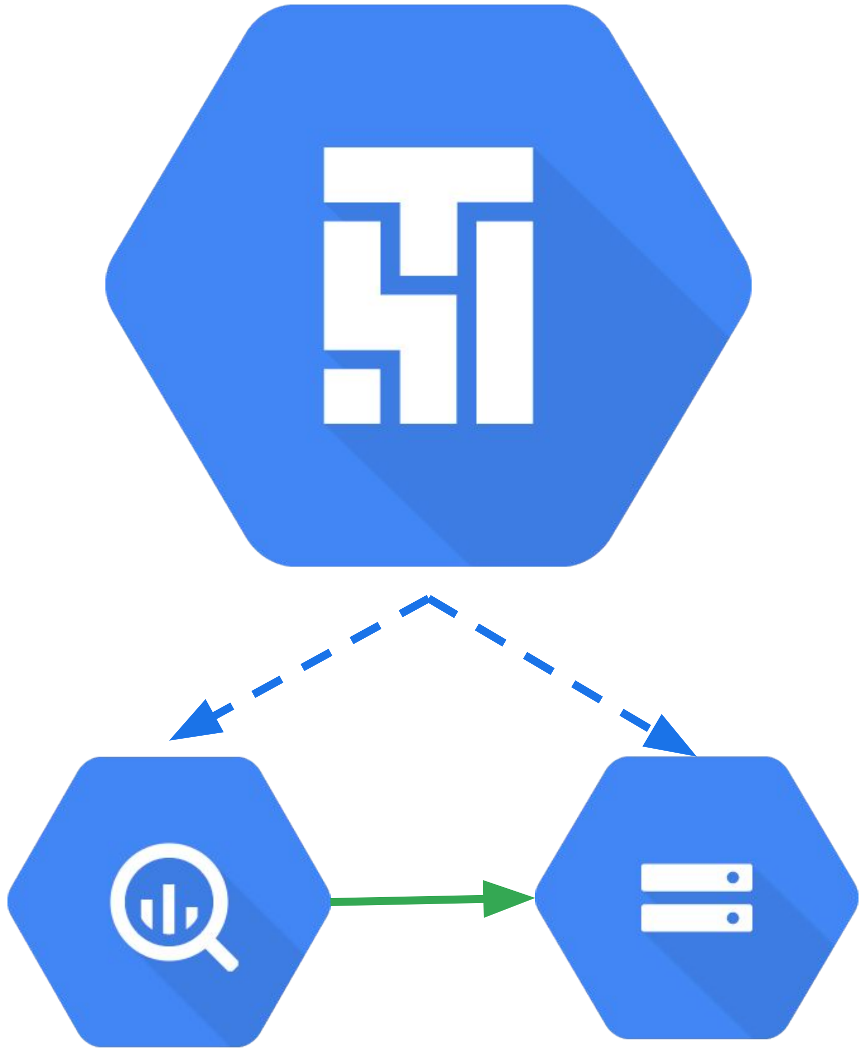
1. Data ingestion
2. Data validation
3. Data preparation
4. Model training
5. Model evaluation
6. Model deployment

1. Data ingestion

You can ingest data from various different sources using built-in operators in Apache Airflow.

- Example: BigQuery to Cloud Storage

  Use the `BigQueryToCloudStorageOperator` to export data from BigQuery to Cloud Storage.



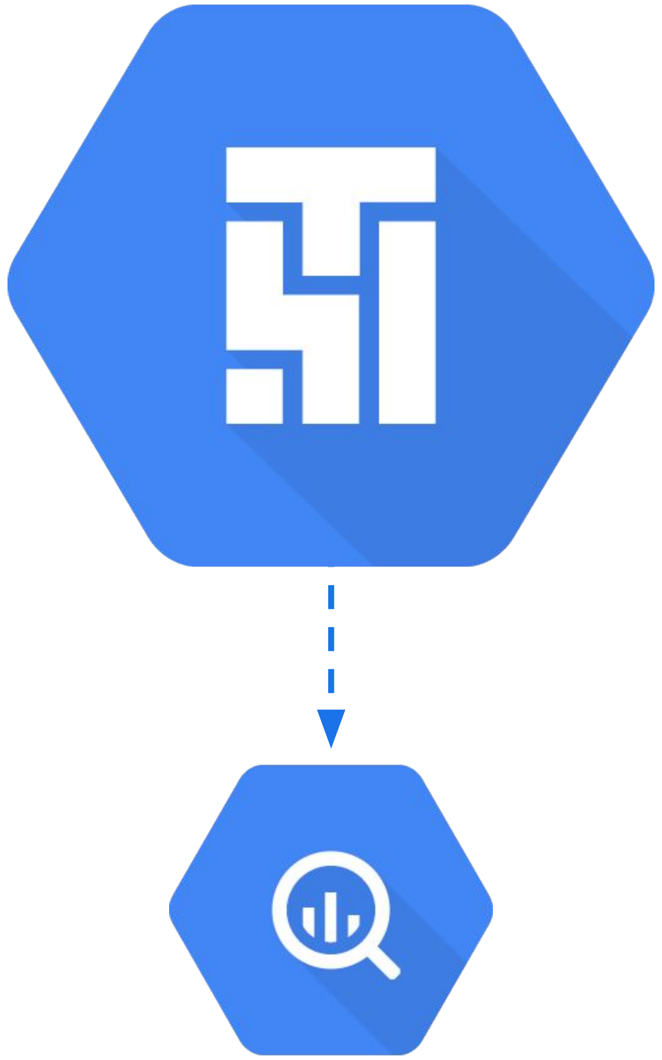
Similar operators exist for other Google Cloud storage solutions, storage solutions on other public clouds, and even on-premises data sources (e.g., via JDBC drivers).

2. Data validation

Be sure that your data is valid and meets your expectations for training; otherwise you may want to stop the pipeline and further explore your data.

When working with data in BigQuery, you can leverage the `BigQueryCheckOperator` with a validation query.

The task generated by the operator will fail if the query returns a certain result, such as `0` or `False`.



2. Data validation

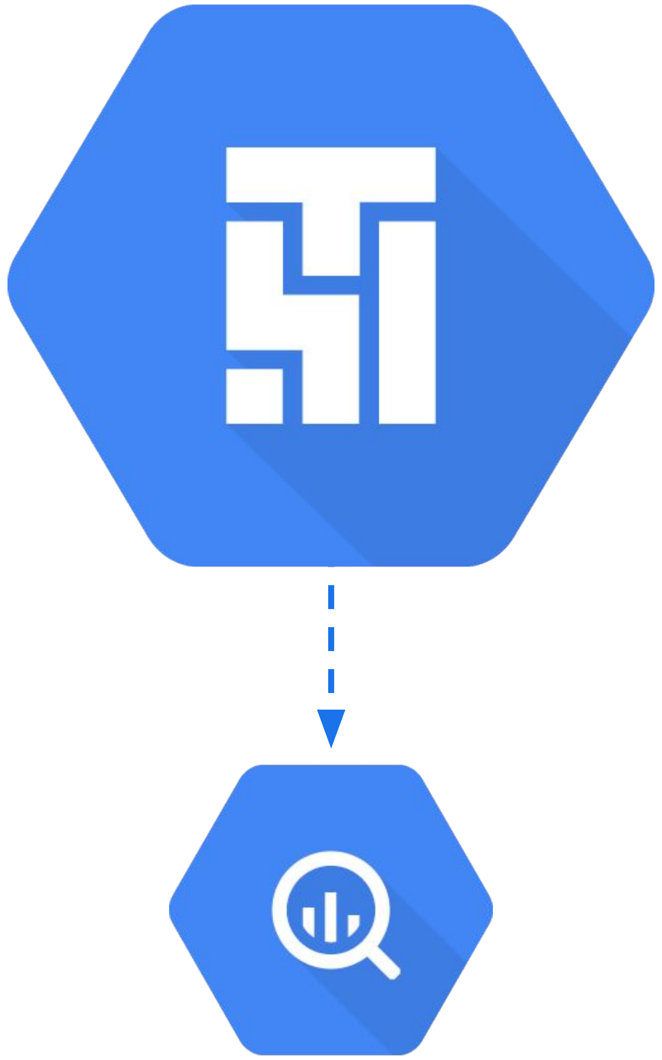
```
check_sql = """
            SELECT COUNT(*)
            FROM ...
            WHERE
                trip_start_timestamp >= TIMESTAMP('{{ macros.ds_add(ds, -30) }}')
            """

bq_check_data_op = BigQueryCheckOperator(
    task_id="bq_check_data_task",
    use_legacy_sql=False,
    sql=check_sql,
)
```

2. Data validation

```
check_sql = """
            SELECT COUNT(*)
            FROM ...
            WHERE
                trip_start_timestamp >= TIMESTAMP('{{ macros.ds_add(ds, -30) }}')
            """

bq_check_data_op = BigQueryCheckOperator(
    task_id="bq_check_data_task",
    use_legacy_sql=False,
    sql=check_sql,
)
```

2. Data validation

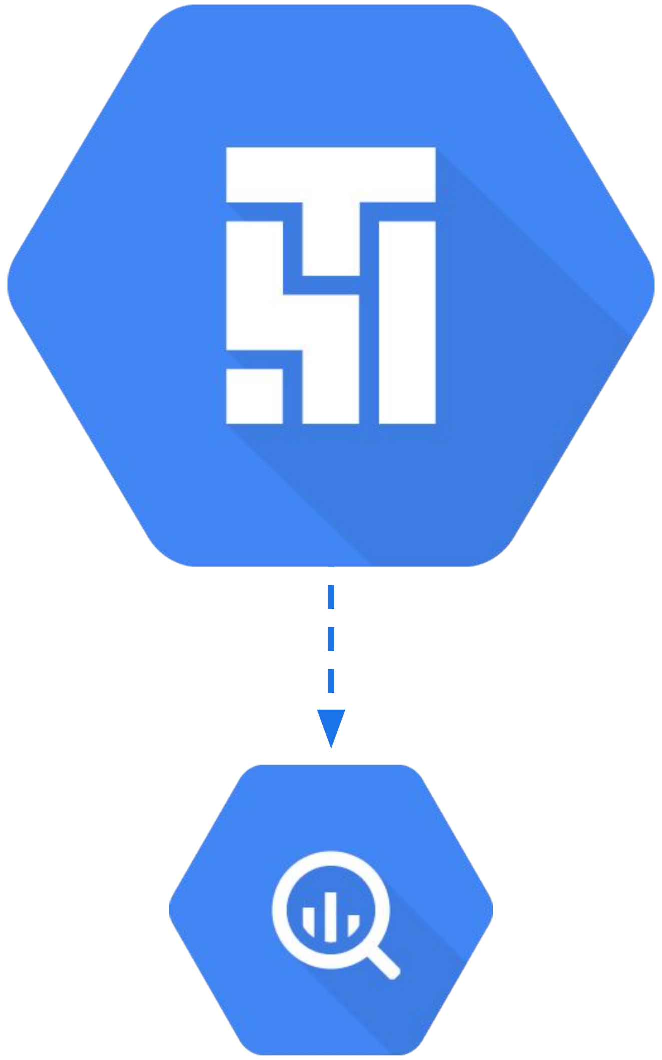
Other types of check operators can also be used.



2. Data validation

Other types of check operators can also be used.

- `BigQueryIntervalCheckOperator`: Checks that the values of metrics given as SQL expressions are within a certain tolerance of values from days_back before.

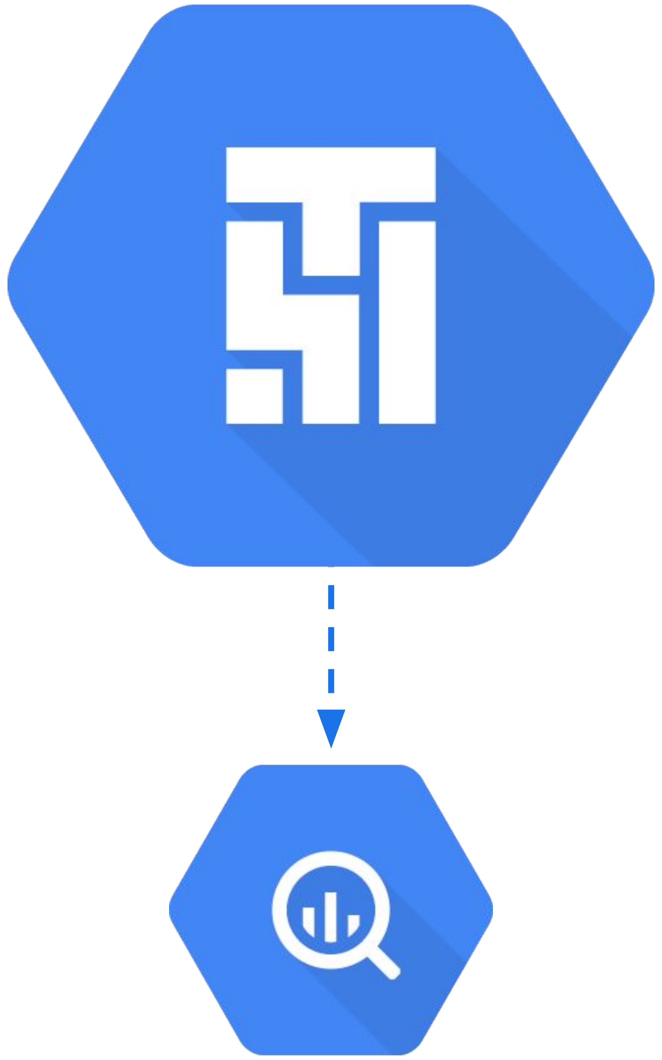


2. Data validation

Other types of check operators can also be used.

- `BigQueryIntervalCheckOperator`: Checks that the values of metrics given as SQL expressions are within a certain tolerance of values from days_back before.

- `BigQueryValueCheckOperator`: Checks that the result of a query is within a certain tolerance of an expected pass value.



Generic SQL versions of these operators are available if you are using Cloud SQL or a database external to Google Cloud.

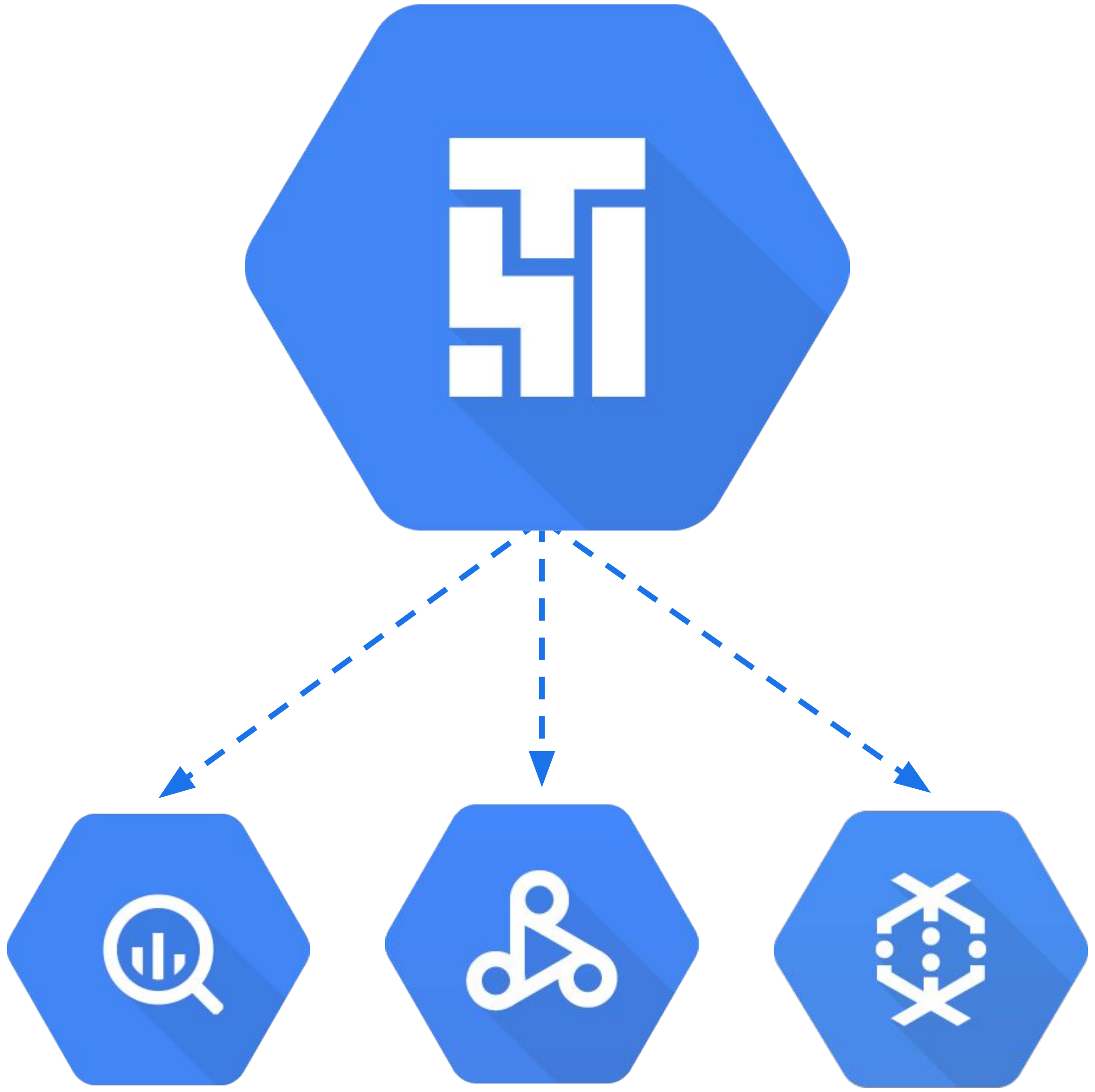
2. Data validation

If the check operator fails, you can use the `ALL_FAILED` trigger rule to trigger a specific downstream task.

```
publish_if_failed_check_op = PubSubPublishOperator(
    task_id="publish_on_failed_check_task",
    project=PROJECT_ID,
    topic=TOPIC,
    messages=[{'data': ERROR_MESSAGE}],
    trigger_rule=TriggerRule.ALL_FAILED
)
...

bq_check_data_op >> publish_if_failed_check_op
```

3. Data preparation

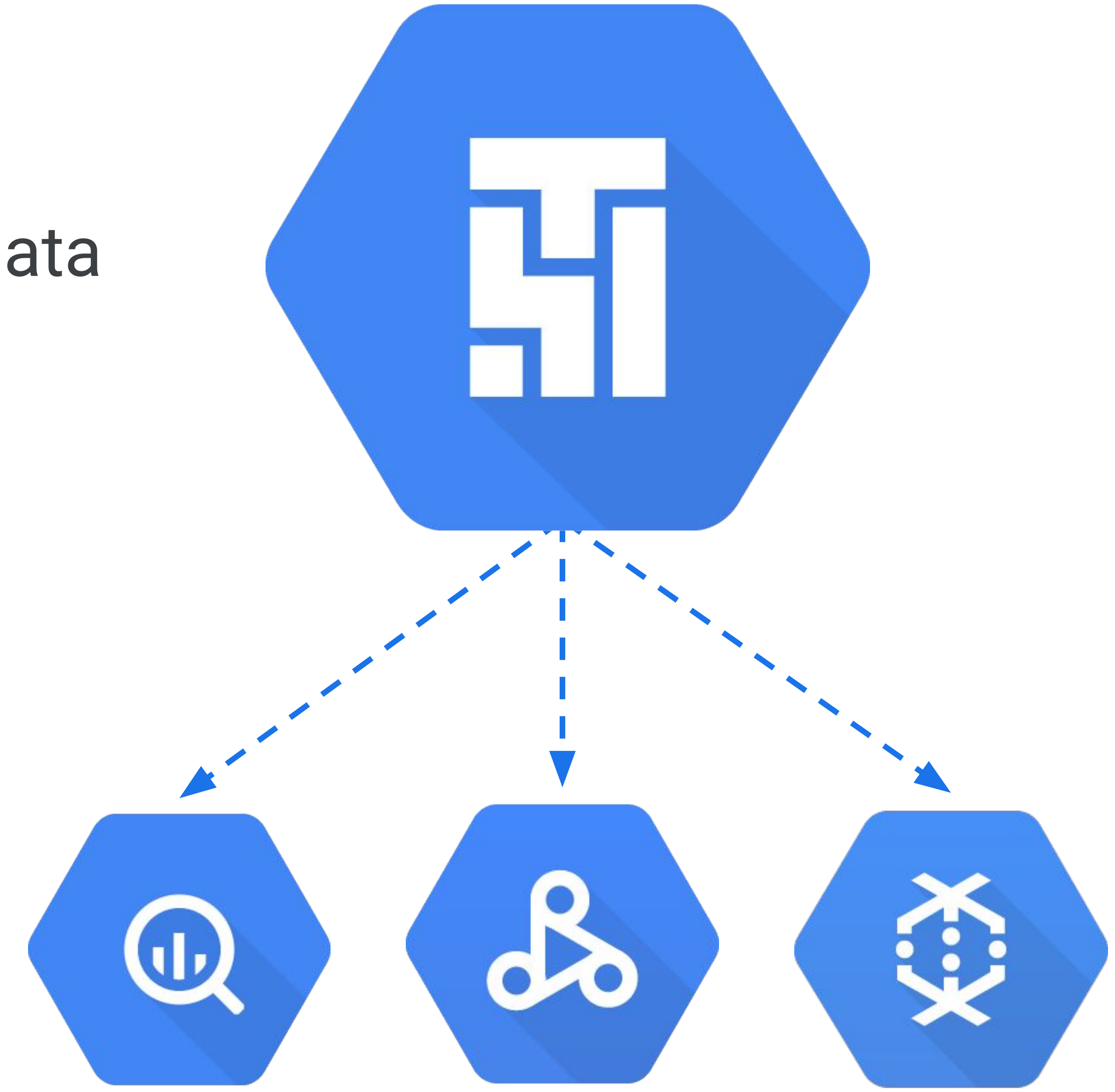
Next, prepare your data for machine learning; for example:



3. Data preparation

Next, prepare your data for machine learning; for example:

- Use BigQuery with a `BigQueryOperator` to process your data using SQL.

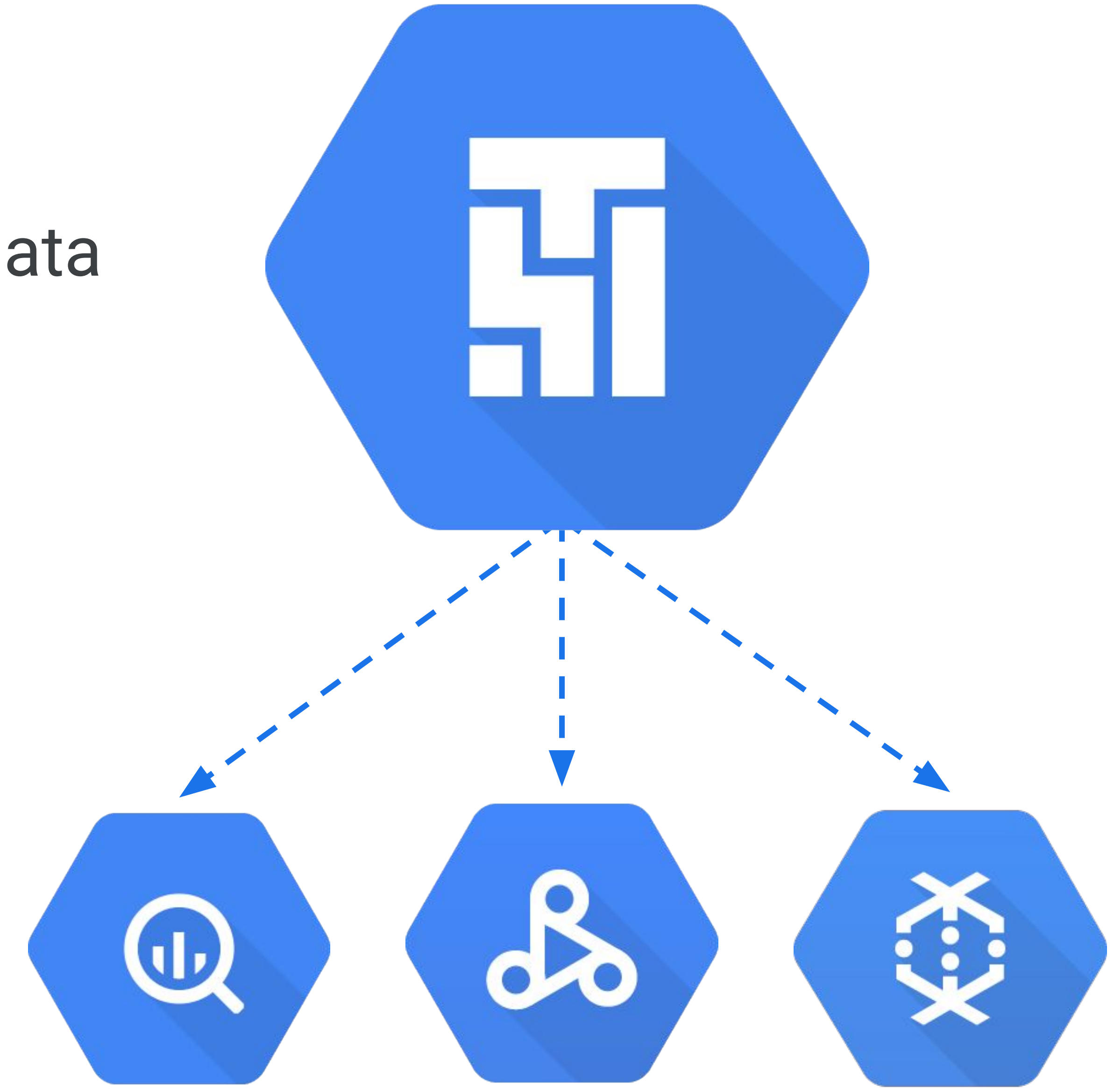


3. Data preparation

Next, prepare your data for machine learning; for example:

- Use BigQuery with a `BigQueryOperator` to process your data using SQL.

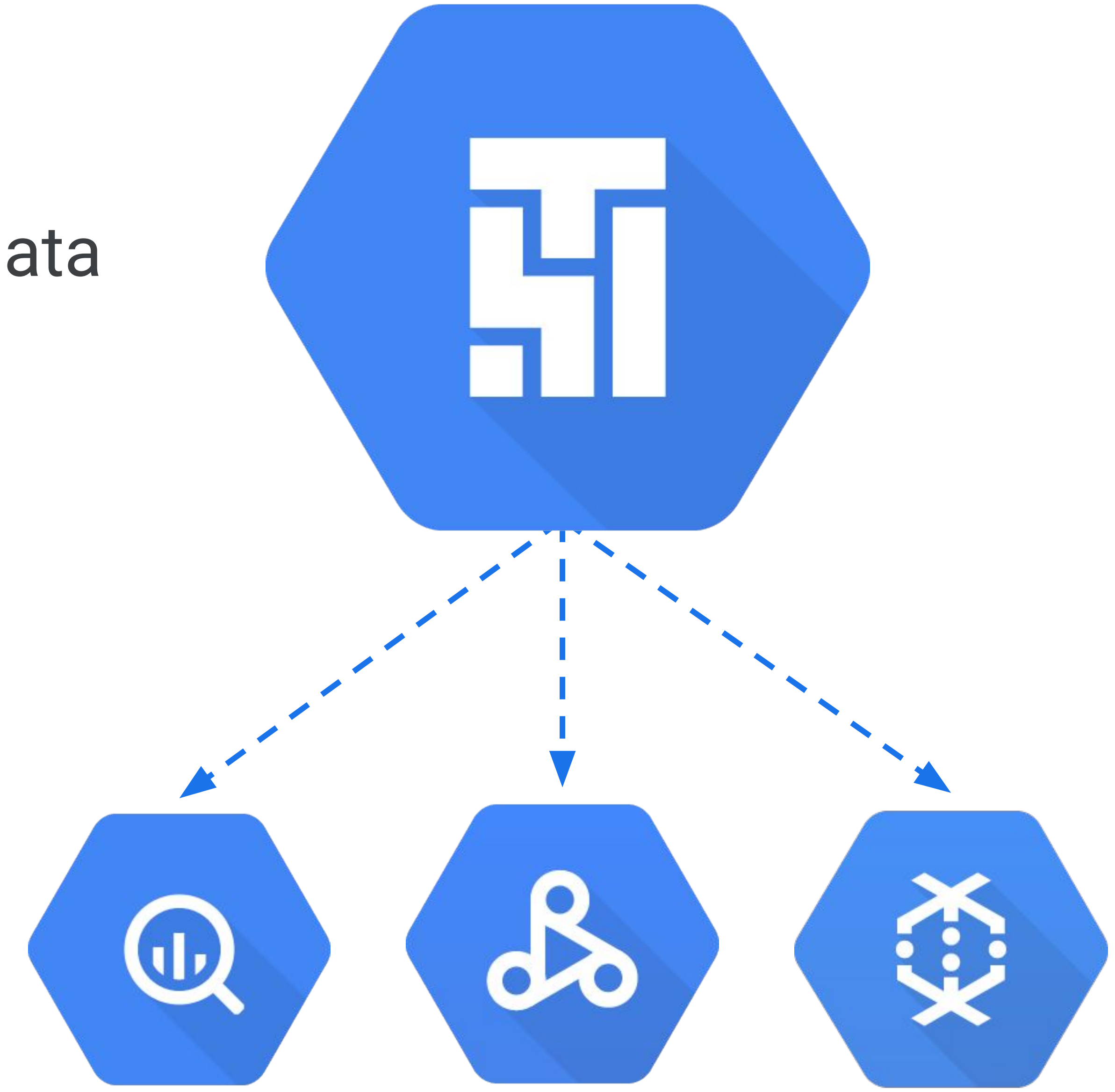
- Run Spark jobs at scale using:



3. Data preparation

Next, prepare your data for machine learning; for example:

- Use BigQuery with a `BigQueryOperator` to process your data using SQL.
- Run Spark jobs at scale using:
  - `DataprocCreateClusterOperator`

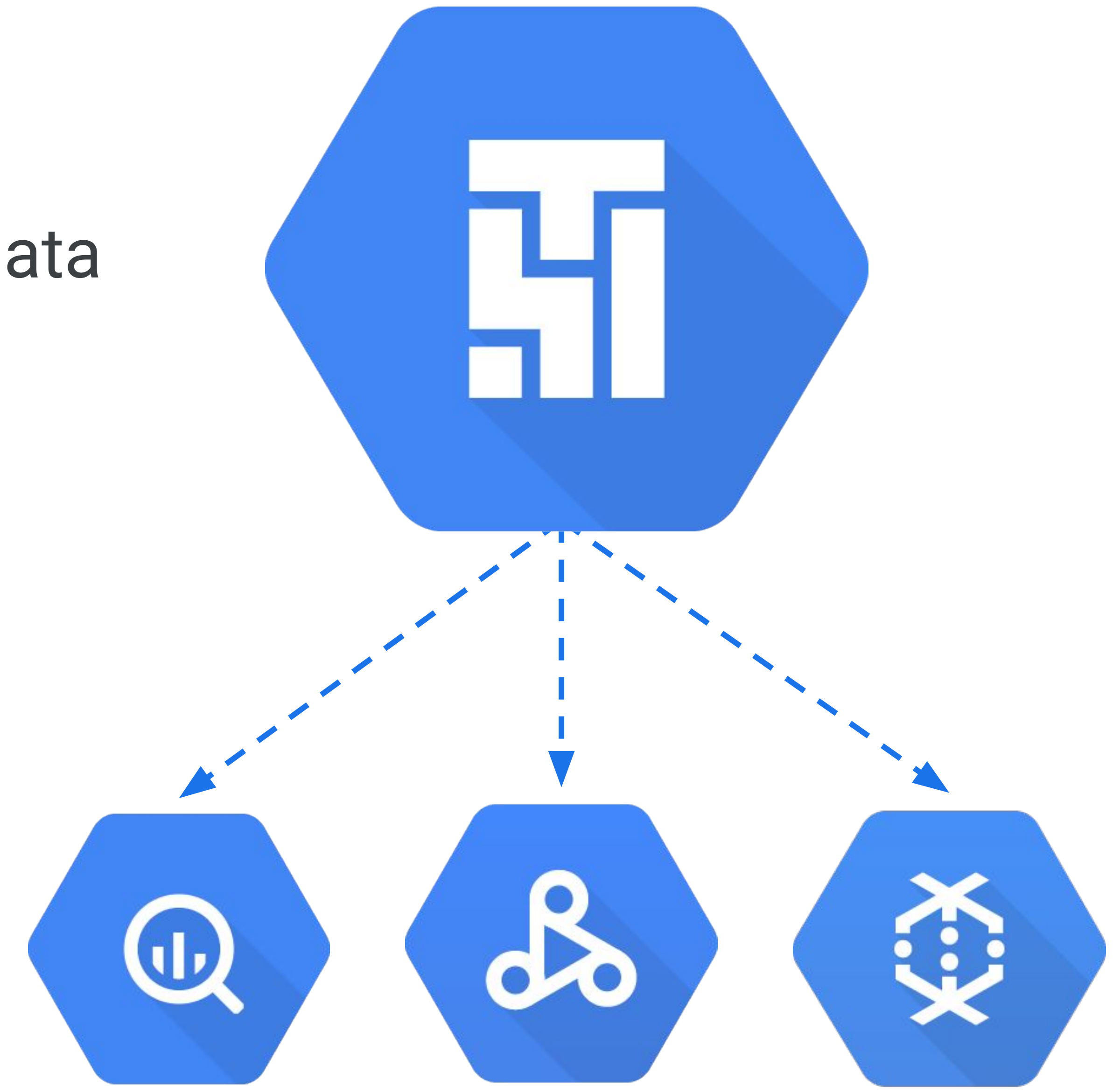


3. Data preparation

Next, prepare your data for machine learning; for example:

- Use BigQuery with a `BigQueryOperator` to process your data using SQL.

- Run Spark jobs at scale using:
  - `DataprocCreateClusterOperator`
  - `DataprocSubmitJobOperator`

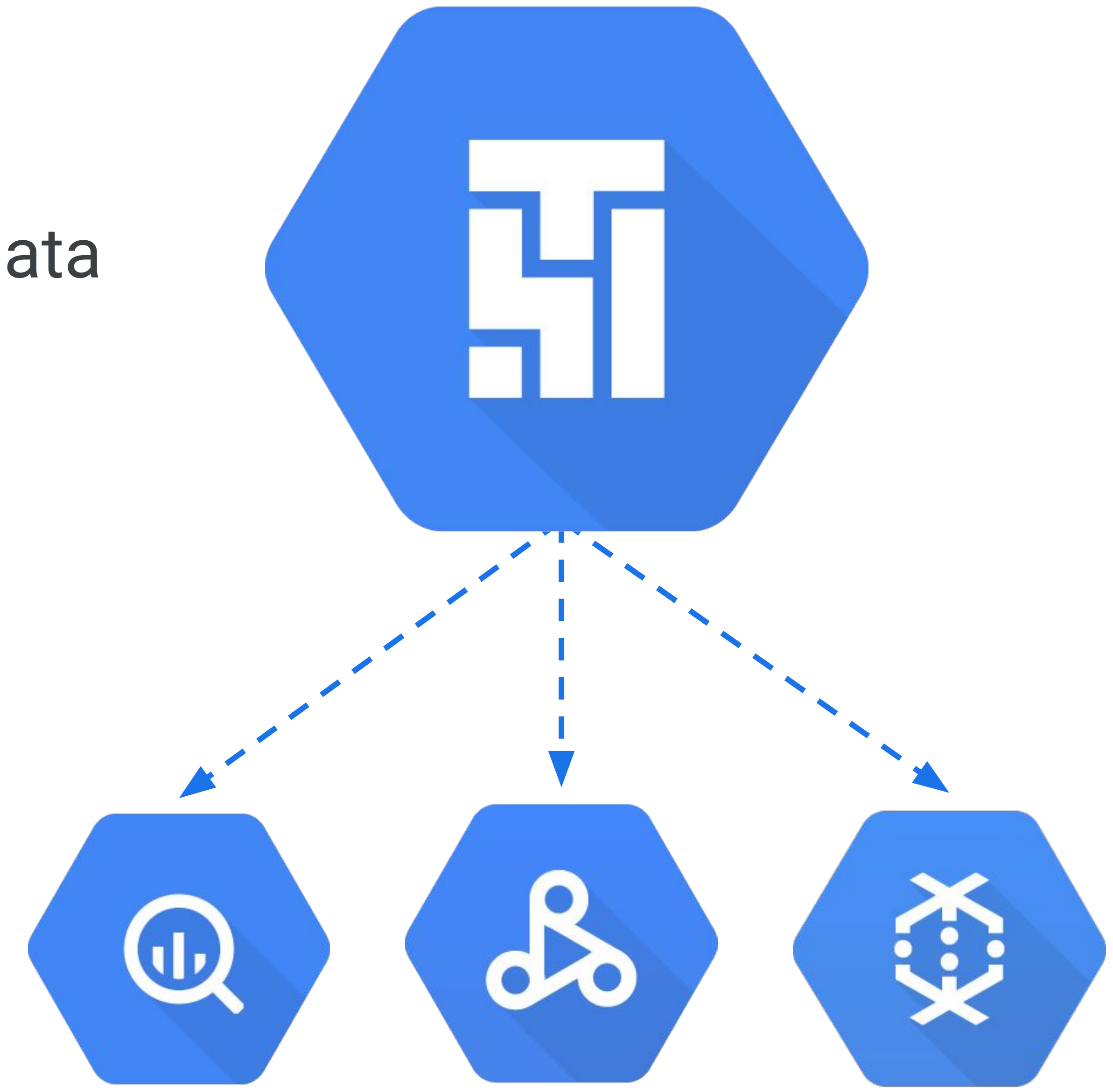


3. Data preparation

Next, prepare your data for machine learning; for example:

- Use BigQuery with a `BigQueryOperator` to process your data using SQL.

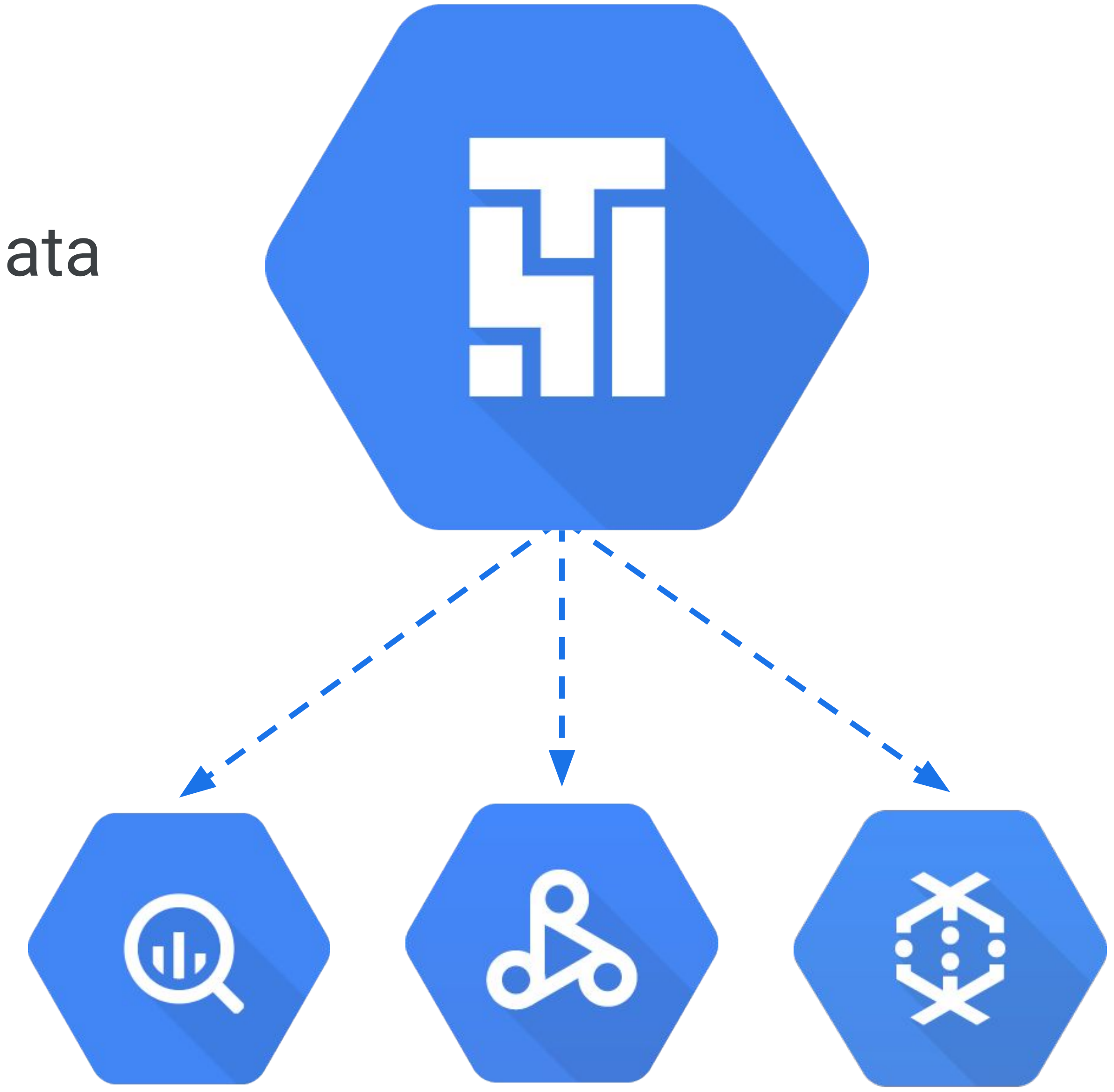
- Run Spark jobs at scale using:
    - `DataprocCreateClusterOperator`
    - `DataprocSubmitJobOperator`
    - `DataprocDeleteClusterOperator`



3. Data preparation

Next, prepare your data for machine learning; for example:

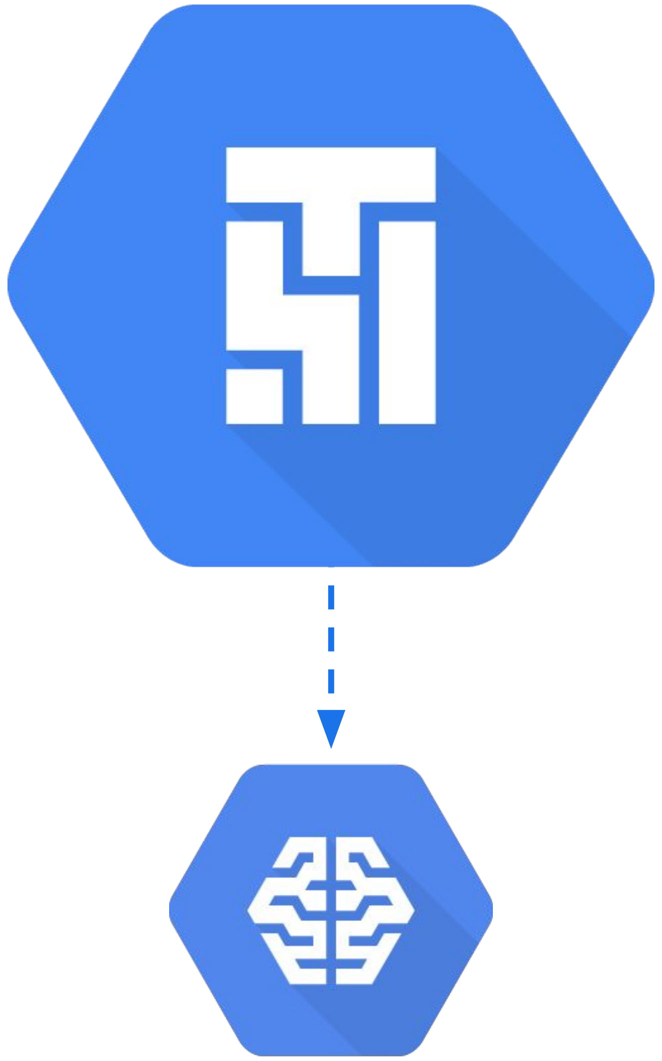
- Use BigQuery with a `BigQueryOperator` to process your data using SQL.
- Run Spark jobs at scale using:
  - `DataprocCreateClusterOperator`
  - `DataprocSubmitJobOperator`
  - `DataprocDeleteClusterOperator`
- Run Apache Beam pipelines using `DataFlowPythonOperator`/`DataFlowJavaOperator`.



4. Model training

Submit your training jobs to AI Platform using the `MLEngineTrainingOperator`.

- This works for TensorFlow, xgboost, and Sci-kit learn models.
- You need to package your training op as a Python package to submit to AI Platform.



4. Model training

Training arguments are expected to be passed in as a list. Each argument will have two consecutive elements of the list:

- "`--arg_name`", where `arg_name` is the name of the argument
- `arg_value`, the argument's value

```
training_args = [
    "--job-dir", job_dir,
    "--output_dir", output_dir,
    "--log_dir", log_dir,
    "--train_data_path", train_files + "chicago_taxi_trips/*.csv",
    "--eval_data_path", valid_files + "chicago_taxi_trips/*.csv" ]
```

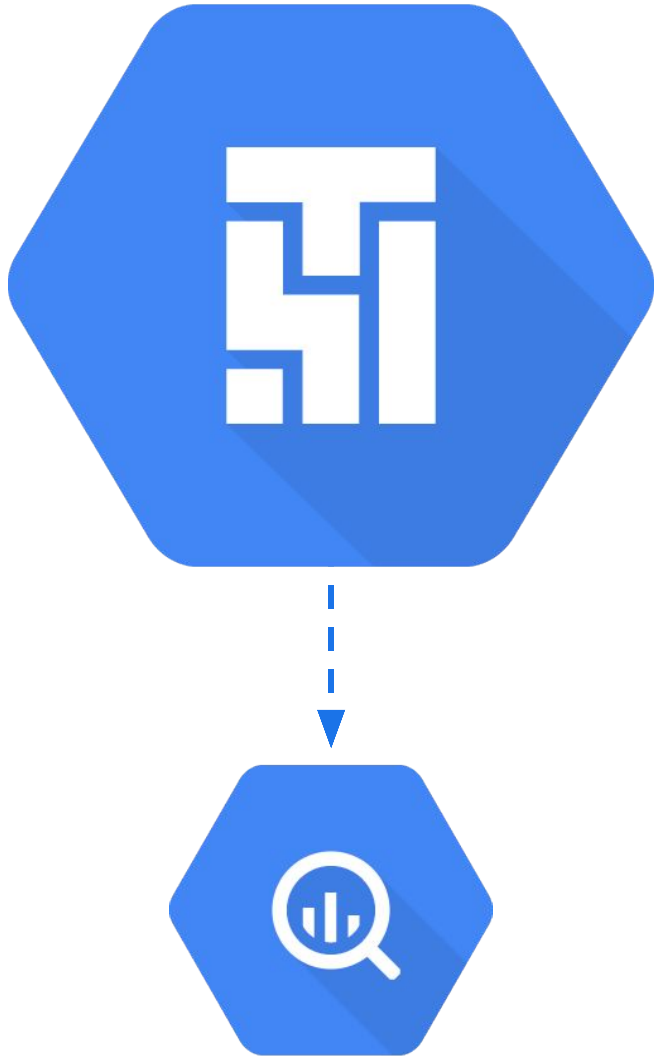
4. Model training

```
ml_engine_training_op = MLEngineTrainingOperator(
    task_id="ml_engine_training_task",
    project_id=PROJECT_ID,
    job_id=job_id,
    package_uris=[PACKAGE_URI],
    training_python_module="trainer.task",
    training_args=training_args,
    region=REGION,
    scale_tier="BASIC",
    runtime_version="2.1",
    python_version="3.7",
    dag=dag  )
```

5. Model evaluation

Capture the evaluation metrics of the newly trained model in a BigQuery table.

```
sql = """ INSERT chicago_taxi.model_metrics
          VALUES('model_version', {0}),
                ('rmse', {1}),
           """.format(Variable.get("NEW_VERSION_NAME"),rmse)

from google.cloud import bigquery
query_job = bigquery.Client.query(sql)
```

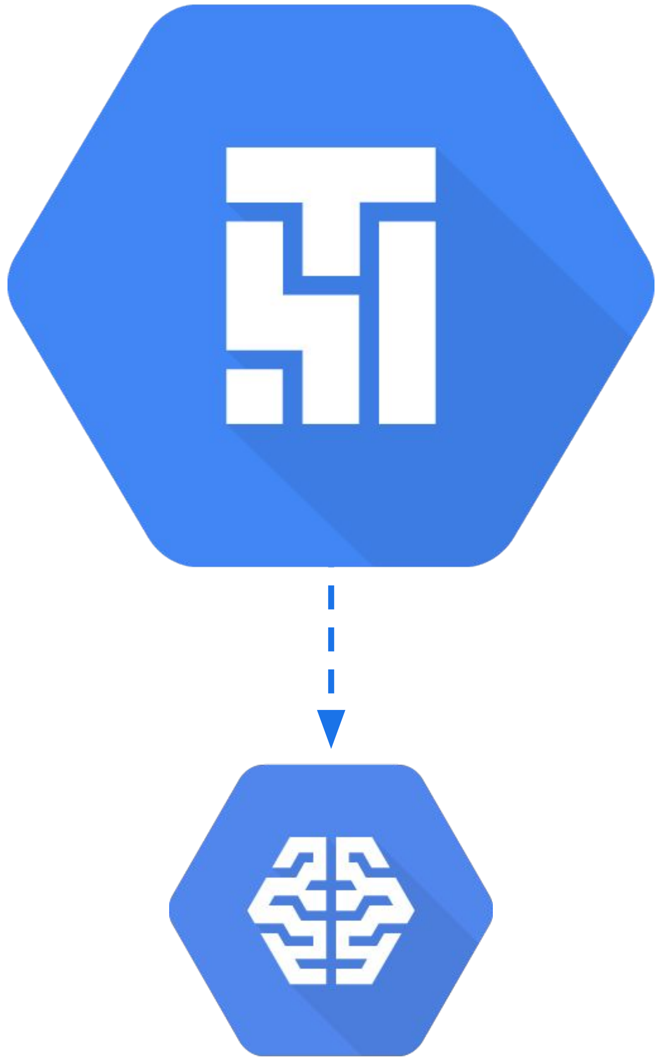


Use a `BigQueryValueCheckOperator` to ensure that the model meets your business needs before deployment.

6. Model deployment

Create a model on AI Platform using the `MLEngineModelOperator`.

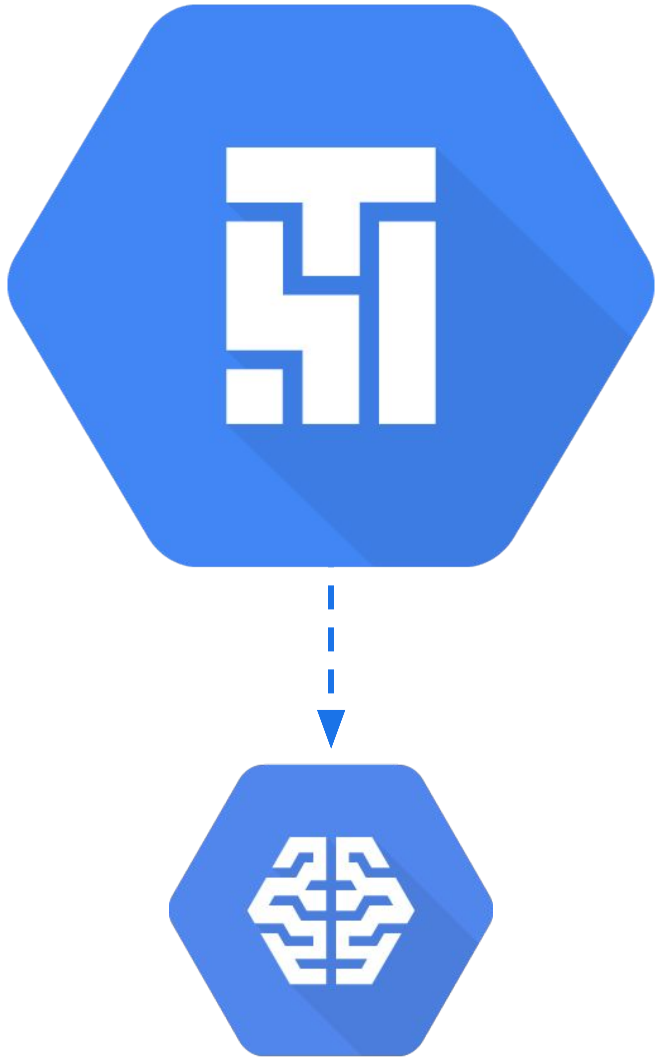
```
ml_engine_create_model_op = MLEngineModelOperator(
        task_id="ml_engine_create_model_{}_task"
                .format(model.replace(".", "_")),
        project_id=PROJECT_ID,
        model={"name": MODEL_NAME},
        operation="create",
        dag=dag
        )
```



6. Model deployment

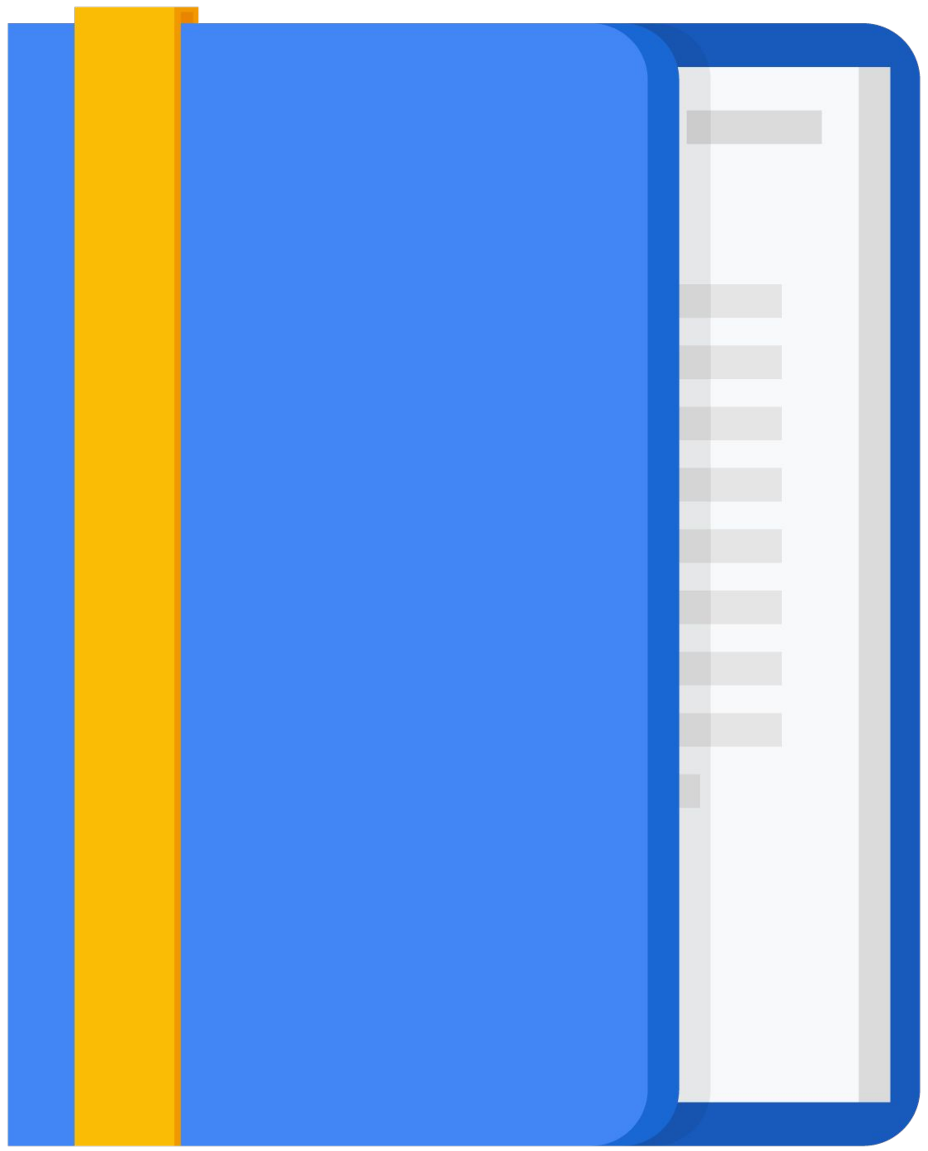
Deploy your trained model using the `MLEngineVersionOperator`.

```
ml_engine_create_version_op = MLEngineVersionOperator(
      task_id="ml_engine_create_version_task",
      project_id=PROJECT_ID,
      model_name=MODEL_NAME,
      version_name=Variable.get("VERSION_NAME"),
      version={
          "deploymentUri": MODEL_LOCATION,
          ...
          },
      operation="create",
      dag=dag
```



Agenda

- What is Cloud Composer?
- Core concepts of Apache Airflow
- Continuous training pipelines using Cloud Composer
- Apache Airflow, containers, and TFX



Running containerized tasks in Airflow

If you have tasks with non-PyPI dependencies, or if the tasks have already been containerized, you can run the containers as Airflow tasks.

Running containerized tasks in Airflow

If you have tasks with non-PyPI dependencies, or if the tasks have already been containerized, you can run the containers as Airflow tasks.

- `KubernetesPodOperator`: By default, launches a pod in the Cloud Composer GKE cluster.
  - Not generally recommended because it can lead to resource starvation.

Running containerized tasks in Airflow

If you have tasks with non-PyPI dependencies, or if the tasks have already been containerized, you can run the containers as Airflow tasks.

- `KubernetesPodOperator`: By default, launches a pod in the Cloud Composer GKE cluster.
    - Not generally recommended because it can lead to resource starvation.
- `GKEPodOperator`: Launch a pod in a specified GKE cluster.

Running containerized tasks in Airflow

If you have tasks with non-PyPI dependencies, or if the tasks have already been containerized, you can run the containers as Airflow tasks.

- `KubernetesPodOperator`: By default, launches a pod in the Cloud Composer GKE cluster.
  - Not generally recommended because it can lead to resource starvation.

- `GKEPodOperator`: Launch a pod in a specified GKE cluster.

Make your `KubernetesPodOperator/GKEPodOperator` tasks idempotent. The pod runs to completion despite Airflow worker shutdown or restart.

TFX pipelines in Airflow

If you have already written your ML workflows using TFX, you can use Airflow as the runner for your TFX DAG.

```
from tfx.orchestration.airflow.airflow_dag_runner import AirflowDagRunner
from tfx.orchestration.airflow.airflow_dag_runner import AirflowPipelineConfig
...
DAG = AirflowDagRunner(AirflowPipelineConfig(_airflow_config)).run(_create_pipeline(
       pipeline_name=_pipeline_name,
       pipeline_root=_pipeline_root,
       data_root=_data_root,
       module_file=_module_file,
       serving_model_dir=_serving_model_dir,
       metadata_path=_metadata_path,
       beam_pipeline_args=_beam_pipeline_args))
```

Lab

Continuous Training Pipelines with Cloud Composer