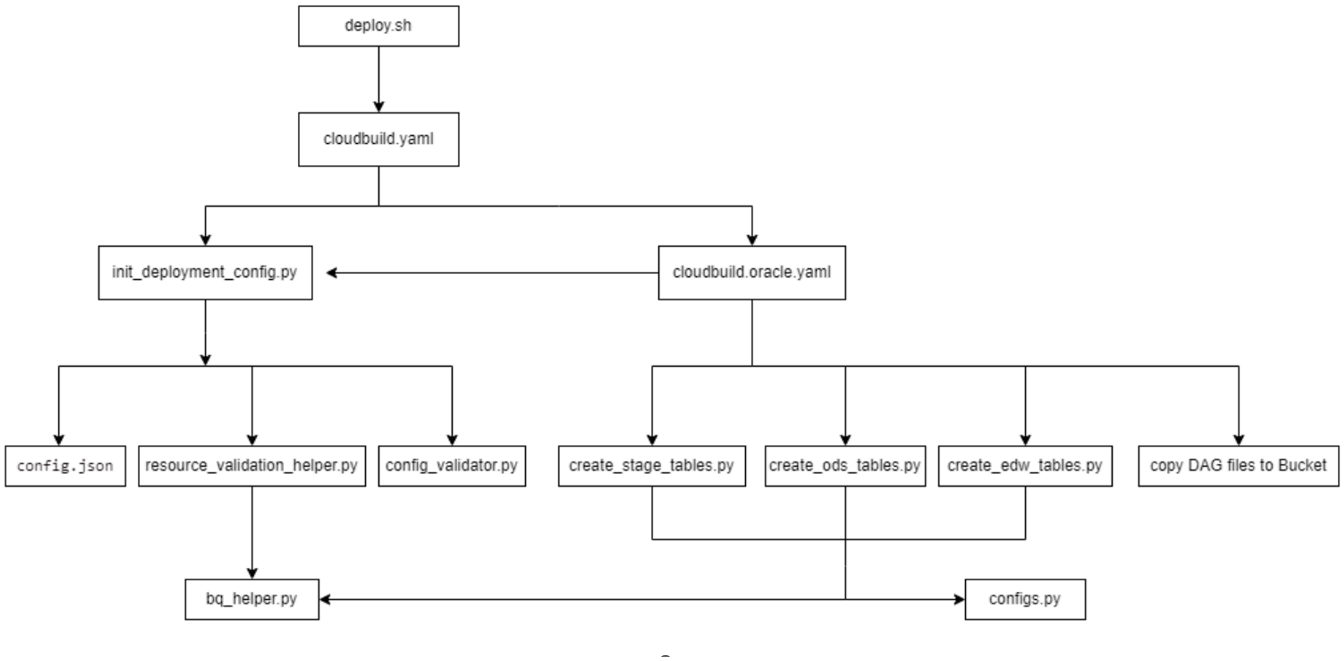
Below is flow of framework in script based and their corresponding features.



deploy.sh : Automates the deployment process for the Cortex Framework for Oracle.
It includes the following steps:

- Reads configuration details from **config.json**, such as the Cloud Build project and the target bucket for logs
- Determines the logs bucket to use, either defaulting to the target bucket or using a provided log bucket parameter.
- Submits a build job to Cloud Build using gcloud builds submit, passing the Cloud Build configuration file **cloudbuild.yaml** and necessary configurations like project ID and substitutions.

cloudbuild.yaml : The cloudbuild.yaml file serves as the configuration blueprint for automating the deployment process. Its primary objectives are to initialize the deployment configuration and execute the necessary steps for deploying Oracle components.
It includes the following steps:

- Executes a Python script (**init_deployment_config.py**) to validate and process the deployment configuration stored in **config.json**, ensuring deployment parameters are correctly set.
- Makes the processed configuration available for subsequent build steps.
- Submits a build job to deploy the Oracle component using the specified Cloud Build configuration file (**cloudbuild.oracle.yaml**).

cloudbuild.oracle.yaml : This configuration file outlines three sequential actions:initializing deployment settings, setting up BigQuery tables, and transferring DAG files to a designated storage location.
These actions collectively facilitate the deployment process of the Oracle-Ebs.
It includes the following steps:

- **Initialize Deployment Configurations :** Reads and processes a configuration file (config.json) using a Python script (init_deployment_config.py)
- **Create BigQuery Tables :** This step involves the creation of BigQuery tables in OdsStage, Ods and Edw layers. Python scripts (create_stage_tables.py, create_ods_tables.py, create_edw_tables.py) are executed to generate these tables.
- **Copy DAG Files :** In this step, DAG files are copied from a specified source directory to a designated bucket location.The gsutil command is employed to facilitate this file transfer.

init_deployment_config.py :

- Loads and validates the configuration file (**config.json**), Validates common configuration resources, including project permissions and resource availability in Google Cloud, such as BigQuery datasets and Cloud Storage buckets.
- Runs workload-specific validators using the **config_validator.py** file.
- Outputs validated and processed configuration or logs error messages if validation fails.
- Maintains a flag (validated) in the configuration file to indicate whether the configuration has been successfully validated to avoid redundant validation.

Supporting Python Scripts :

config_validator.py :
This script is used to ensure that the configuration dictionary contains all necessary values for an ORACLE setup, specifically related to datasets.
It checks for the presence of required values and performs validation on resource constraints using helper functions.
It includes the following steps:

- Retrieve ORACLE configuration from the provided dictionary and check the presence of required keys like 'ORACLE' and 'datasets'.
- Check for the presence of specific dataset keys ('Ods', 'OdsStage', and 'Edw')
- Construct a list of dataset constraints using the provided information.
- Validate the constructed dataset constraints using a **resource_validation_helper** function.
- Log success or failure messages based on validation results.
- Return the processed configuration dictionary if validation succeeds.

resource_validation_helper.py : This script validates the existence, accessibility, and writability of specified Cloud Storage Buckets and BigQuery Datasets based on provided constraints.

It includes the following steps:

- initialize Google Cloud clients for storage and BigQuery.
- For each bucket constraint:It fetches the bucket object and performs location and writability checks.
- For each dataset constraint:It checks dataset existence and location.
- Creates a temporary table to check writability and cleans up afterward.
- Returns True if all resources pass the validation criteria, otherwise it returns False.

bq_helper.py : This Python script is a library for handling BigQuery-related operations. It includes functions for checking table and dataset existence, creating tables, and deleting tables. It includes the following utility functions:

- table_exists: Checks if a given table exists in BigQuery.
- dataset_exists: Checks if a given dataset exists in BigQuery.
- dataset_exists_in_location: Checks if a given dataset exists in BigQuery in a specific location.
- create_table: Creates a BigQuery table based on a given schema.
- delete_table: Deletes a BigQuery table.

configs.py :
The script facilitates the handling of Cortex configuration files in JSON format. It offers a function to load these files and convert them into Python dictionaries.
load_config_file Function:

- It loads a JSON configuration file.
- Checks if the specified config file exists, raising a FileNotFoundError if it doesn't.
- Attempts to open and parse the config file using the json.load() method.
- Raises an exception with a clear error message if JSON parsing fails due to malformed or empty content.
- Logs the loaded configuration using Python's logging module.

Python scripts for creating tables in BigQuery within specified datasets:

create_stage_tables.py, create_ods_tables.py, create_edw_tables.py

The Python script generates BigQuery raw, Ods and Edw tables using configuration parameters from a **config.json** file and list of tables from a **table_settings.yaml** file, with each table's schema defined in separate CSV files (**table_schema** for Raw tables, **ods_edw_schema** for Ods and Edw tables), and creates the tables if they do not exist, based on the provided configurations.
It includes the following steps:

- Load configuration parameters from a JSON config file.
- Extract necessary parameters such as project ID, dataset name, and location.
- Read table configurations from a YAML settings file and iterate over each table.
- Check if the table already exists in BigQuery. If not, create it.

- Create tables based on schema mappings defined in CSV files.Logs messages throughout the process, indicating the progress and any issues encountered.

dataform_dag.py

This Airflow DAG automates the integration of files between Google Cloud Storage (GCS) and Dataform. It handles creating necessary Dataform repositories and workspaces, listing files from a GCS bucket, reading their content, and writing them to specified directories in Dataform.

Core Functions:

list_files_in_bucket(bucket_name):
Lists all files in the specified GCS bucket with .sqlx or .js extensions.

read_file_from_gcs(bucket_name, file_name):
Reads the content of a specified file from GCS.

write_to_dataform(file_name, data, directory, kwargs):
Writes the content of a file to a specific directory in Dataform.

read_and_write_to_dataform(context):
Orchestrates the reading of files from GCS and writing them to Dataform.

DAG Tasks:

make_repository: Creates a Dataform repository.
make_workspace: Creates a Dataform workspace.
first_initialization_step, last_initialization_step: Initializes the Dataform workspace.
install_npm_packages: Installs necessary NPM packages in Dataform.
make_oracle_directory: Creates the "OracleFinance" directory in Dataform.
make_source_directory: Creates the "sources" directory in Dataform.
list_files_task: Lists files from the GCS bucket.
read_write_task: Reads files from GCS and writes them to Dataform workspace.