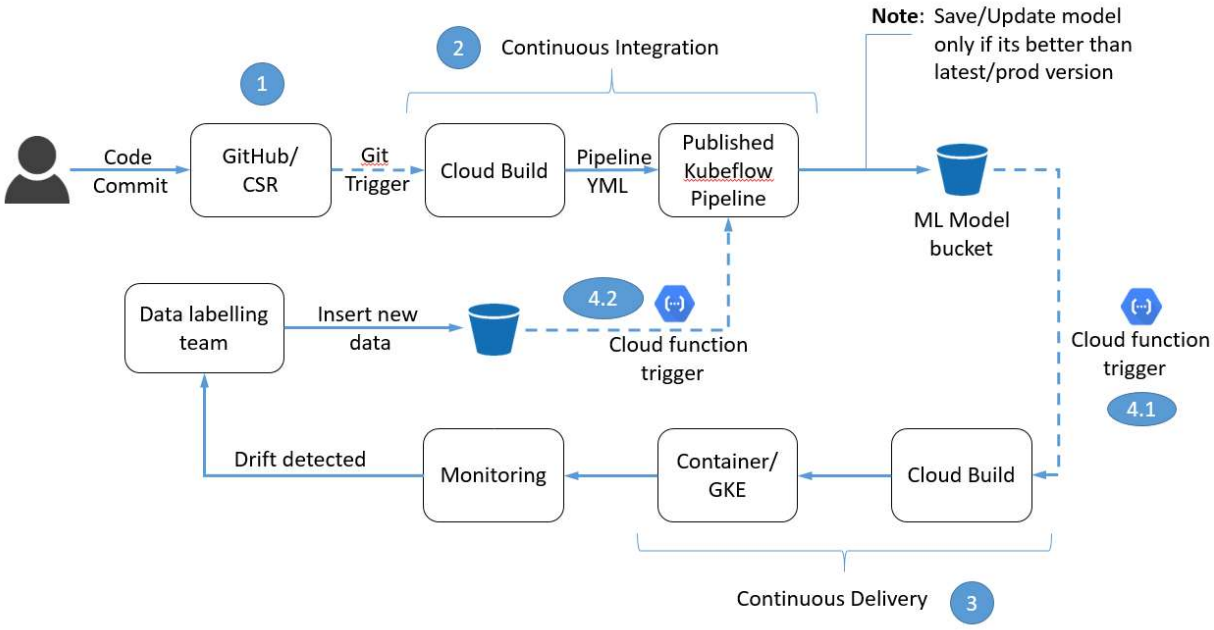
Architecture



Folder Structure

In [ ]:
```
+---data
|       dataloan.csv
|       FE_test_inputs.csv
|       validation_data.csv
|
+---img
|       screenshots.png
|
+---train_pipeline (CI)
|       pipeline_components.py
|       pipeline_publisher.py
|       pipeline_run.py
|       cloudbuild.yaml
|       Dockerfile
|       pipeline_base_image_builder.sh
|
+---deployment (CD)
|       score.py
|       requirements.txt
|       deployment.yaml
|       service.yaml
|       cloudbuild.yaml
|       Dockerfile
|       scoring_image_builder.sh
|
+---tests
|       test_1_FE.py
|       test_2_impute_class.py
|       test_3_score.py
|
+---utils
|       FE.py
|       impute_class.py
|
|---architecture.pptx
|---main.ipynb
|---requirements.txt
|---secrets.json
```

Index

Step 1 : Setup a GitHub/CSR/ Bitbucket repository containing project folders

- 1.1 Details of “train_pipeline” folder
- 1.2 Details of “deployment” folder
- 1.3 Details of “tests” folder
- 1.4 Details of “utils” folder
- 1.5 Push entire project folder to GitHub/CSR/ Bitbucket

Step 2 : Setup Cloud Build trigger for CI pipeline

- 2.1 Details of “train_pipeline/cloudbuild.yaml” file
- 2.2 Set up CI pipeline on GCP

Step 3 : Setup Cloud Build trigger for CD pipeline

- 3.1 Details of “deployment /cloudbuild.yaml” file
- 3.2 Set up CD pipeline on GCP

Step 4 : Setup cloud function triggers

- 4.1 CD pipeline trigger on model update
- 4.2 Model retrain trigger on data change

Step 5 : Setup monitoring service

Step 1 :

Setup a GitHub/CSR/Bitbucket repository containing project folders

1.1 Details of “train_pipeline” folder

```
+---train_pipeline
|       pipeline_components.py
|       pipeline_publisher.py
|       pipeline_run.py
|       cloudbuild.yaml
|       Dockerfile
|       pipeline_base_image_builder.sh
```

pipeline_components.py

```
Note : There are 2 methods to create component
```

```
1. Create component from python function and use base_image which can run all
steps
2. Create component by defining separate docker containers for each step
In below code we have used method 1
```

In [4]:
```
%%writefile pipeline/pipeline_components.py
# Import all dependencies
import subprocess, sys, os
from kfp.components import InputPath, InputTextFile, OutputPath, OutputTextFile,C
from typing import NamedTuple
import kfp.dsl as dsl

# Every step in training job requires base image which already contains dependenc
# Lets build base image with new tag everytime new code is checked in, as depende
# To have new image name everytime we use git commit id as image tag.

# Get git commit id
git_commit_id = subprocess.run(['git', 'log','-1', '--pretty=%h'], capture_output

# Create container name with new tag
os.environ["CONTAINER_NAME"] = 'vinodswnt306/new_public_mlops:' + git_commit_id.s
CONTAINER_NAME = os.environ["CONTAINER_NAME"]

# e.g.
# CONTAINER_NAME = 'vinodswnt306/new_public_mlops:aada71f'

# Step 1 of Kubeflow pipeline
# Read the data
@dsl.python_component(
    name='read_split',
    description='',
    base_image=CONTAINER_NAME  # you can define the base image here, or when you
)
def read_and_split(gcs_path: str,output_csv: OutputPath(str),mlpipeline_ui_metada
    """
    Read and Splits data into train, validation and test set

    Parameters:
    gcs_path (str) : Path of input data
    output_csv (str) : (internally assigned by kfp) Path where output csv will be
                       passed to next container step
    mlpipeline_ui_metadata_path : Path where metadata is stored

    """
    from sklearn.model_selection import train_test_split
    import os
    print(os.listdir())
    import json
    import pandas
    import gcsfs
    import pandas as pd

    file_list = gcs_path.split(',')

    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = 'secrets.json'
    fs = gcsfs.GCSFileSystem(project='leafy-ether-314809' , token='secrets.json')

    # Note : Here I have read only 1 file, we can create empty dataframe and
    # read all the csv files in it with concat on axis 0 (stacking one above the
    with fs.open(file_list[0]) as f:
        loan_data = pandas.read_csv(f)
```

```
    # keeping validation set aside
    X = loan_data.iloc[:,loan_data.columns!='Loan_Status']
    y = loan_data.iloc[:,loan_data.columns=='Loan_Status']
    X, Xval, y, yval = train_test_split(X,y,test_size=0.15, random_state=45)
    loan_data = pd.concat([X,y],axis=1).reset_index(drop=True)
    Xval,yval = Xval.reset_index(drop=True),yval.reset_index(drop=True)

    # Send output for next container step
    loan_data.to_csv(output_csv,index=False)

    # Log train data files
    file_list = [i+'#'+fs.stat(i)['generation'] for i in file_list]
    metadata = {
    'outputs' : [
    # Markdown that is hardcoded inline
    {
      'storage': 'inline',
      'source': '# Training files used\n'+ ','.join(file_list),
      'type': 'markdown',
    }]
      }
    with open(mlpipeline_ui_metadata_path, 'w') as f:
        json.dump(metadata, f)

    #Save splitted data to validation forlder if required


# Step 2 of Kubeflow pipeline
@dsl.python_component(
    name='preprocess',
    description='',
    base_image=CONTAINER_NAME # you can define the base image here, or when you b
)
def preprocess(text_path: InputPath(),output_csv: OutputPath(str),imputer_path: O
    """
    Data preprocessing step

    Parameters
    ----------
    text_path (str) : Path of input training data
    output_csv (str) : (internally assigned by kfp) Path where output csv will be
                       passed to next container step
    imputer_path (str) : (internally assigned by kfp) Path where imputer instance
                       passed to training container step

    """
    import os
    print(os.listdir())
    import sys
    sys.path.append('.')
    import pandas as pd
    import numpy as np
    from utils.impute_class import impute

    global imputer_cls
    loan_data = pd.read_csv(text_path)
```

```
    imputer = impute()
    loan_data = imputer.fit(loan_data)
    loan_data.to_csv(output_csv,index=False)

    imputer_cls = imputer
    import joblib
    import dill
    with open(imputer_path, "wb") as dill_file:
        dill.dump([imputer_cls],dill_file)


# Step 3 of Kubeflow pipeline
@dsl.python_component(
    name='FE',
    description='adds two numbers',
    base_image=CONTAINER_NAME  # you can define the base image here, or when you
)
def FE(text_path: InputPath(),output_csv: OutputPath(str),FE_path: OutputPath(str
    """
    Feature engineering step

    Parameters
    ----------
    text_path (str) : Path of input training data
    output_csv (str) : (internally assigned by kfp) Path where output csv will be
                        passed to next container step
    FE_path (str) : (internally assigned by kfp) Path where Feature engineering i
                         stored and passed to training container step

    """

    import pandas as pd
    import numpy as np
    from sklearn.preprocessing import OneHotEncoder
    import pandas as pd
    import numpy as np
    import sys
    import joblib
    global FE_cls
    import dill
    sys.path.append('.')
    from utils.FE import Feature_engineering

    ####################

    loan_data = pd.read_csv(text_path)
    FE_pipeline = Feature_engineering()
    loan_data = FE_pipeline.fit(loan_data)
    loan_data.to_csv(output_csv,index=False)

    FE_cls = FE_pipeline
    with open(FE_path, "wb") as dill_file:
        dill.dump([FE_cls],dill_file)

# Step 4 of Kubeflow pipeline
@dsl.python_component(
    name='train',
```

```
    description='',
    base_image=CONTAINER_NAME  # you can define the base image here, or when you
)
def train(text_path: InputPath(),imputer_path: InputPath(), FE_path :  InputPath(
    """
    Model training step

    Parameters
    ----------
    text_path (str) : Path of input training data
    imputer_path (str) : (internally assigned by kfp) Path where imputer instance
    FE_path (str) : (internally assigned by kfp) Path where Feature engineering p
    mlpipeline_metrics_path : Path where metrics are stored
    """
    import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import f1_score
    import dill
    import sys
    import gcsfs
    import os
    sys.path.append('.')

    # Load train data
    loan_data = pd.read_csv(text_path)

    loan_data['Loan_Status']=loan_data['Loan_Status'].astype('int')
    loan_data = loan_data[loan_data>=0].dropna()

    X = loan_data.iloc[:,loan_data.columns!='Loan_Status']
    y = loan_data.iloc[:,loan_data.columns=='Loan_Status']

    # Split data into train and test set
    Xtrain, Xtest, ytrain, ytest = train_test_split(X,y,test_size=0.10, random_st
    log_reg = LogisticRegression()
    log_reg_model = log_reg.fit(Xtrain,ytrain) # classifier function will train t
    ypred = log_reg_model.predict(Xtest) # Performing perdiction on test test
    f1 = f1_score(y_true=ytest,y_pred=log_reg_model.predict(Xtest)) # Getting f1

    # Log metrics
    import json
    accuracy = 0.9
    metrics = {
    'metrics': [{
      'name': 'accuracy-score', # The name of the metric. Visualized as the colum
      'numberValue':  accuracy, # The value of the metric. Must be a numeric valu
      'format': "PERCENTAGE",   # The optional format of the metric. Supported va
    }]
    }
    with open(mlpipeline_metrics_path, 'w') as f:
        json.dump(metrics, f)

    # Load imputer and feature engineering pipeline which we saved and passed fro
    # previous container steps
    with open(imputer_path, "rb") as imputer_file, open(FE_path, 'rb') as FE_file
        imputer = dill.load(imputer_file)[0]
```

```
        FE_pipeline = dill.load(FE_file)[0]

    # Save model file into current container
    model_file = (r'loan_model.pkl')
    with open(model_file, "wb") as dill_file:
        dill.dump([imputer,FE_pipeline, log_reg_model],dill_file)

    # Connect to GCS
    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = 'secrets.json'
    fs = gcsfs.GCSFileSystem(project='leafy-ether-314809' , token='secrets.json',

    Production_model_f1 = 0.8
    # If no files present then save first model in folder 01(version)
    if len(fs.ls('gs://loan_model_pipeline')) == 0 :
        # Upload model to GCS
        with open("loan_model.pkl", "rb") as local_file:
            with fs.open("gs://loan_model_pipeline/" + "1/loan_model.pkl", "wb")
                gcs_file.write(local_file.read())

    # Save model to new folder if better than production model
    elif f1 > Production_model_f1: # production model f1 score
        gcs_files = sorted([int(i.replace('loan_model_pipeline/','')) for i in fs
        next_folder_num = str(int(gcs_files[-1]) + 1)
        with open("loan_model.pkl", "rb") as local_file:
            with fs.open("gs://loan_model_pipeline/" + next_folder_num + "/loan_m
                gcs_file.write(local_file.read())

        with open("ds_train.yaml", "rb") as local_file:
            with fs.open("gs://loan_model_pipeline/" + next_folder_num + "/pipeli
                gcs_file.write(local_file.read())
```

```
Overwriting pipeline/pipeline_components.py
```

pipeline_publisher.py

In [2]:
```
%%writefile pipeline/pipeline_publisher.py
#This file compiles all the pipeline steps and saves yaml file
from pipeline_components import *
import kfp
import kfp.components as comp
import kfp
import kfp.dsl as dsl
from kfp import compiler
from kfp import components

################################################################################

# Convert read_and_split function to a pipeline operation.
read_split_op = components.func_to_container_op(
    read_and_split,
    base_image=CONTAINER_NAME,
    packages_to_install=['pandas==1.1.4','gcsfs','scikit-learn'] # optional
)

# Convert preprocessing function to a pipeline operation.
preprocess_op = components.func_to_container_op(
    preprocess,
    base_image=CONTAINER_NAME,
    packages_to_install=['pandas==1.1.4','gcsfs','joblib','dill'] # optional
)

# Convert feature engineering function to a pipeline operation.
FE_op = components.func_to_container_op(
    FE,
    base_image=CONTAINER_NAME,
    packages_to_install=['pandas==1.1.4','gcsfs','scikit-learn','dill'] # optiona
)

# Convert training function to a pipeline operation.
train_op = components.func_to_container_op(
    train,
    base_image=CONTAINER_NAME,
    packages_to_install=['pandas==1.1.4','gcsfs','scikit-learn','joblib','dill']
)

################################################################################

@dsl.pipeline(
   name='Calculation pipeline',
   description='pipeline'
)
def ds_pipeline(
   gcs_path: str,
                ):
    #Passing pipeline parameter and a constant value as operation arguments
    read_split = read_split_op(gcs_path) #Returns a dsl.ContainerOp class instanc
    read_split.container.set_image_pull_policy('Always') # Because by default it

    preprocess=preprocess_op(read_split.outputs['output_csv'])
    preprocess.container.set_image_pull_policy('Always')
```

```
    FE = FE_op(preprocess.outputs['output_csv'])
    FE.container.set_image_pull_policy('Always')

    train = train_op(FE.outputs['output_csv'],preprocess.outputs['imputer'],FE.ou
    train.container.set_image_pull_policy('Always')


# Combine pipeline and save yaml file
kfp.compiler.Compiler().compile(
    pipeline_func=ds_pipeline,
    package_path='pipeline/ds_train.yaml')
```

```
Overwriting pipeline/pipeline_publisher.py
```

pipeline_run.py

In [2]:
```
%%writefile pipeline/pipeline_run.py

# This file runs a saved pipeline and also if new model is registered then it sav
# its compiled pipeline file yaml with it

# Check gcs storage
import os
import json
import pandas
import gcsfs
import pandas as pd

# Run kfp pipeline
import kfp
# Set up Kubeflow client using its url
client = kfp.Client('https://2886795272-31380-shadow05.environments.katacoda.com/
run = client.create_run_from_pipeline_package(
        pipeline_file='pipeline/ds_train.yaml',
        arguments = {'gcs_path': 'gs://bucket-306/data/train/dataloan.csv' },expe
        )

client.wait_for_run_completion(run.run_id, 3600)
if client.get_run(run.run_id).run.status == 'Succeeded':
    print("completed")
else:
    print("job failed")
```

```
Overwriting pipeline/pipeline_run.py
```

cloudbuild.yaml

In [4]:
```
%%writefile pipeline/cloudbuild.yaml
steps:
  # this runs files required for training
  - name: 'python'
    env:
    - 'NEW_CONTAINER=${_CONTAINER_NAME}'
    id: Create and Compile pipeline
    entrypoint: /bin/sh
    args:
    - -c
    - "pip install -r requirements.txt && python pipeline/pipeline_publisher.py"
  # In above line we can add execute testing scripts as well

  # this builds base image to use it for kubeflow pipelines
  - name: 'gcr.io/cloud-builders/docker'
    id: base_image_creation
    entrypoint: /bin/sh
    args:
    - -c
    - "bash pipeline/pipeline_base_image_builder.sh && touch NewFile.txt"

  # this runs files required for training
  - name: 'python'
    env:
    - 'NEW_CONTAINER=${_CONTAINER_NAME}'
    id: Run pipeline
    entrypoint: /bin/sh
    args:
    - -c
    - "pip install -r requirements.txt && python pipeline/pipeline_run.py"
```

```
Overwriting pipeline/cloudbuild.yaml
```

Dockerfile

```
This creates dockerfile for creating base image which is used for kubeflow
pipeline components
```

In [ ]:
```
%%writefile pipeline/Dockerfile
FROM python:3.7-slim
COPY ./requirements.txt ./secrets.json ./pipeline/ds_train.yaml ./
COPY ./utils utils
RUN pip install -r requirements.txt
```

pipeline_base_image_builder.sh

```
This file runs few commands required for docker build and push to repository
Note: we can also use google cloud container registry as well, to store image,
Here I have
      used dockerhub
```

In [ ]:
```
%%writefile pipeline/pipeline_base_image_builder.sh
export CONTAINER_NAME="$(git log -1 --pretty=%h)"
docker login --username=$(docker_usrname) --password=$(docker_password)
docker build -t vinodswnt306/new_public_mlops:$CONTAINER_NAME -f ./pipeline/Docke
docker push vinodswnt306/new_public_mlops:$CONTAINER_NAME
```

1.2 Details of “deployment” folder

```
Note : Following code for section 1.2 is not yet implemented, it is just a
sample code
```

```
+---deployment
|       score.py
|       requirements.txt
|       deployment.yaml
|       service.yaml
|       cloudbuild.yaml
|       Dockerfile
|       scoring_image_builder.sh
```

score.py

```
In this file we will write flask app which loads model and predicts on input
json
```

In [ ]:
```
from flask import Flask, jsonify, request
import dill
import os

app = Flask(__name__)

target={0:'Accept', 1:'Reject'}

a,b,c = dill.load('models/loan_model.pkl')

def scorer(text):
    encoded_text = a.transform([text])
    score = c.predict(xgb.DMatrix(encoded_text))
    return score

@app.route('/score', methods=['POST'])
def predict_fn():
    data = request.get_json()['data']
    predictions = scorer(data)
    return jsonify({'predictions ': str(predictions), 'Category ': target.get(pre

@app.route('/')
def hello():
    return 'Welcome to Loan Prediction Application'

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=int(os.environ.get('PORT', 5000)))
```

requirements.txt

```
In this file we will write dependencies that model prediction require
```

In [ ]:
```
flask
scikit-learn==0.22
dill
gunicorn
```

deployment.yaml

```
In this file we will write kubernetes config
```

In [ ]:
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Loan_Status_app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: Loan_Status
  template:
    metadata:
      labels:
        app: Loan_Status
    spec:
      containers:
      - name: Loan-app
        image: gcr.io/gcp-repo-name/Loan_image
        ports:
        - containerPort: 8080
        env:
          - name: PORT
            value: "8080"
```

service.yaml

```
In this file we will write on which service we want to run our flask app
```

In [ ]:
```
apiVersion: v1
kind: Service
metadata:
  name: Loan_Status
spec:
  type: LoadBalancer
  selector:
    app: Loan_Status
  ports:
  - port: 80
    targetPort: 8080
```

cloudbuild.yaml

```
In this file we will write cloud build code for Continuous deployment pipeline
```

```
Steps :
1) Test scoring script
2) Build container with scoring script
3) Deploy container to GKE
```

Dockerfile

```
In this file we will write script to build container which runs score.py file
```

In [ ]:
```
# lightweight python
FROM python:3.7-slim

# Copy local code to the container image.
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY ./deployment/* ./*
COPY ./utils ./utils

RUN ls -la $APP_HOME/

# Install dependencies
RUN pip install -r requirements.txt

ENV PORT 5000

# Run the flask service on container startup
#CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 ComplaintsServer
CMD [ "python", "score.py" ]
```

scoring_image_builder.sh

```
%%writefile deployment/scoring_image_builder.sh
docker login --username=$(docker_usrname) --password=$(docker_password)
docker build -t gcr.io/gcp-repo-name/Loan_image -f ./deployment/Dockerfile .
docker push gcr.io/gcp-repo-name/Loan_image
```

1.3 Details of “tests” folder

```
+---tests
|       test_1_FE.py
|       test_2_impute_class.py
|       test_3_score.py
```

test_1_FE.py

In [ ]:
```
import pytest
import sys, os.path

py_scrpt_dir = (os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
sys.path.append(py_scrpt_dir)
from utils.FE import *

FE_test_inputs = pd.read_csv(r'.\data\FE_test_inputs.csv')

def test_func1():
    global FE_test_inputs
    FE_pipeline = Feature_engineering()
    FE_test_output = FE_pipeline.fit(FE_test_inputs)

    number_of_non_numeric_columns = FE_test_output.select_dtypes(exclude=['int64'
    assert number_of_non_numeric_columns == 0
```

test_2_impute_class.py

In [ ]:
```
import pytest
import sys, os.path

py_scrpt_dir = (os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
sys.path.append(py_scrpt_dir)
from utils.impute_class import *

loan_data = pd.read_csv(r'.\data\dataloan.csv')

def test_func1():
    global loan_data
    imputer = impute()
    loan_data = imputer.fit(loan_data)
    number_of_columns_with_NA = loan_data.isna().sum()[loan_data.isna().sum() > 0

    assert number_of_columns_with_NA == 0
```

test_3_score.py

```
Here we write script to test functions in score.py file
```

In [ ]:
```

```

1.4 Details of “utils” folder

```
+---utils
|       FE.py
|       impute_class.py
```

FE.py

In [ ]:
```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd
import numpy as np

class Feature_engineering:
    def __init__(self):
        self.z=0

    def fit(self,loan_data):

        loan_data.drop(['Loan_ID'],axis=1,inplace=True)

        catgeorical_features = ['Gender','Married','Dependents','Education','Self
        for feature in catgeorical_features:
            loan_data[feature] = loan_data[feature].astype('category')

        X = loan_data.iloc[:,loan_data.columns!='Loan_Status']
        y = loan_data.iloc[:,loan_data.columns=='Loan_Status']

        self.ohe = OneHotEncoder().fit(X.select_dtypes('category'))
        catg_cols_transform = self.ohe.transform(X.select_dtypes('category')).toa
        self.catg_feat_names = X.select_dtypes('category').columns
        dfOneHot = pd.DataFrame(catg_cols_transform, columns = self.ohe.get_featu
        loan_data_OHE = pd.concat([X, dfOneHot], axis=1).drop(self.catg_feat_name

        loan_data = pd.concat([loan_data_OHE,y],axis=1)

        return loan_data


    def transform(self,X):
        X.drop(['Loan_ID'],axis=1,inplace=True)
        catgeorical_features = ['Gender','Married','Dependents','Education','Self
        for feature in catgeorical_features:
            X[feature] = X[feature].astype('category')

        catg_cols_transform = self.ohe.transform(X.select_dtypes('category')).toa
        dfOneHot = pd.DataFrame(catg_cols_transform, columns = self.ohe.get_featu
        loan_data_OHE = pd.concat([X, dfOneHot], axis=1).drop(self.catg_feat_name

        return loan_data_OHE
```

impute_class.py

In [ ]:
```
import pandas as pd
import numpy as np

class impute:
    def __init__(self):
        self.imputer_dict_for_prod = {}

    def fit(self,loan_data):
        cat_df = loan_data.drop(['Loan_ID'],axis=1).iloc[:,0:-1].select_dtypes(ex
        num_df = loan_data.drop(['Loan_ID'],axis=1).iloc[:,0:-1].select_dtypes(ir
        mode_impute_dict = cat_df.mode().iloc[0]
        mean_impute_dict = dict(num_df.mean())

        self.imputer_dict_for_prod = {**mode_impute_dict, **mean_impute_dict}

        cat_df.fillna(cat_df.mode().iloc[0],inplace=True)
        num_df.fillna(num_df.mean(),inplace=True)

        loan_data = pd.concat([cat_df,num_df,loan_data[['Loan_ID','Loan_Status']]

        return loan_data


    def transform(self,df):
        for i,j in self.imputer_dict_for_prod.items():
            df[i].fillna(j, inplace=True)

        return df
```

Step 2 : Setup Cloud Build trigger for CI pipeline

2.1 Details of “train_pipeline/cloudbuild.yaml” file

In [ ]:
```
steps:
  # Docker Build base image for pipeline
  - name: 'gcr.io/cloud-builders/docker'
    id: image
    entrypoint: /bin/sh
    args:
    - -c
    - "bash docker/pipeline_base_image_builder.sh"

  # Compine and run pipeline
  - name: 'python'
    id: run
    entrypoint: /bin/sh
    args:
    - -c
    - "pip install -r requirements.txt && python pipeline/pipeline_publisher.py \
    && python pipeline/pipeline_run.py"
```

2.2 Set up CI pipeline on GCP

In [ ]:
```
Steps :
1) Go to cloud build page on GCP console
2) Click on create trigger
3) Give name for trigger and select and link your desired Git/CSR/BitBucket repos
4) Give path of cloudbuild.yaml
5) Save trigger and exit
```

Step 3 : Setup Cloud Build trigger for CD pipeline

3.1 Details of "deployment /cloudbuild.yaml" file

In [ ]:
```
This will be a trigger based on http request (which we will send in step 4.1 from
```

3.2 Set up CD pipeline on GCP

In [ ]:
```

```

Step 4 : Setup cloud function triggers

4.1 CD pipeline trigger on model update

In [ ]:
```
It will be triggered based on changes in bucket where model is stored
Any new version update will trigger this function and it will run CD pipeline

Steps :

def hello_gcs(event, context):
    This function will check if event contains filename as model.pkl* then it wil
    create invoke cloud build trigger using http request

*because pipeline submits both model and pipeline into bucket. So to avoid double
```

4.2 Setting up cloud function trigger for retraining (run yaml from model folder on data change)

In [ ]:
```
It will be triggered based on changes in bucket where data is stored
Any new version update will trigger this function and it will run CI pipeline

Steps :
1) run pipeline.yaml from latest model folder, using kubeflow client.run

def hello_gcs(event, context):
    """Triggered by a change to a Cloud Storage bucket.
    Args:
         event (dict): Event payload.
         context (google.cloud.functions.Context): Metadata for the event.
    """
    file = event

    # Check gcs storage
    import os
    import json
    import pandas
    import gcsfs
    import pandas as pd
    # Connect to GCS
    import gcsfs
    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = 'secrets.json'
    fs = gcsfs.GCSFileSystem(project='leafy-ether-314809' , token='secrets.json',

    # Download latest pipeline
    gcs_files = sorted([int(i.replace('loan_model_pipeline/','')) for i in fs.ls(
    next_folder_num = str(gcs_files[-1])
    fs.download("gs://loan_model_pipeline/" + next_folder_num + "/model_pipeline.

    # Run kfp pipeline
    import kfp
    # Set up Kubeflow client using its url
    client = kfp.Client('https://2886795272-31380-shadow05.environments.katacoda.
    run = client.create_run_from_pipeline_package(
         pipeline_file='/tmp/pipeline.yaml',
         arguments = {'gcs_path': 'gs://bucket-306/data/train/dataloan.csv' },exp
         )

    print(f"Processing file: {file['name']}.")
```

Monitoring

In [ ]:
```
step 1 : create a python or pyspark script to compare 2 distributions (train vs.
step 2 : create a function to run script (In case of pyspark, submit job to datap
step 3 : create a cloud scheduler to invoke above function on regular time interv
```