**Insurance**

**Machine learning project**

**Data Description**

**Input Features**

The insurance data contains several columns, including:

age: The dataset includes the age of the insured individuals, indicating the age at which the insurance coverage was acquired or the age of the individuals at the time of data recording.

sex: The gender of the insured individuals is recorded as a categorical variable, distinguishing between "male" and "female" policyholders.

bmi: The dataset provides the Body Mass Index (BMI) of the insured individuals. BMI is a numerical measure calculated based on an individual's weight and height, used to assess health risks related to weight.

children: The dataset contains numerical values representing the number of children or dependents covered by the insurance policy of each individual.

smoker: This column is a binary variable indicating whether an individual is a smoker or a non-smoker. The categories in this column are "yes" for smokers and "no" for non-smokers.

region: The geographic region where each insured individual resides is included in the dataset. The region column contains categorical values, such as "northeast," "southeast," "southwest," and "northwest," among others.

**Target Feature**

expenses (Target): The target variable in the data set is labeled as expenses, representing the medical expenses incurred by each insured individual. It contains numerical values and serves as the variable of interest for analysis and prediction.

**Problem Description**

The provided dataset offers a comprehensive view of insurance-related information, presenting essential attributes of insured individuals.

It encompasses their age, gender, Body Mass Index (BMI), the number of covered dependents, smoking habits, and geographic region. With age providing insights into when insurance coverage was acquired or data recorded, the gender column distinguishes between male and female policyholders. The BMI values aid in assessing health risks, while the number of children indicates family coverage. The binary smoker variable classifies individuals as smokers or non-smokers. Lastly, the geographic region allows for regional analysis.

However, the most critical aspect lies in the 'expenses' column, serving as the target variable, which showcases the medical expenses incurred by each individual. This data forms the bedrock for analysis and prediction, enabling valuable insights into factors influencing medical costs and contributing to enhanced decision-making in the realms of healthcare and insurance.

**Tools Used**

* Python 3.8 is used while creating the environment and libraries like NumPy, Pandas, Scikit-learn.
* Pymongo, Exceptions and logger are used for developing the model.
* VS code is used to development the modular model
* MongoDB is used to store and retrieve the data.

MLOPS

* DVC - Pipeline Orchestration
* MlFlow - Experimentation
* BentoML - Local Deployment

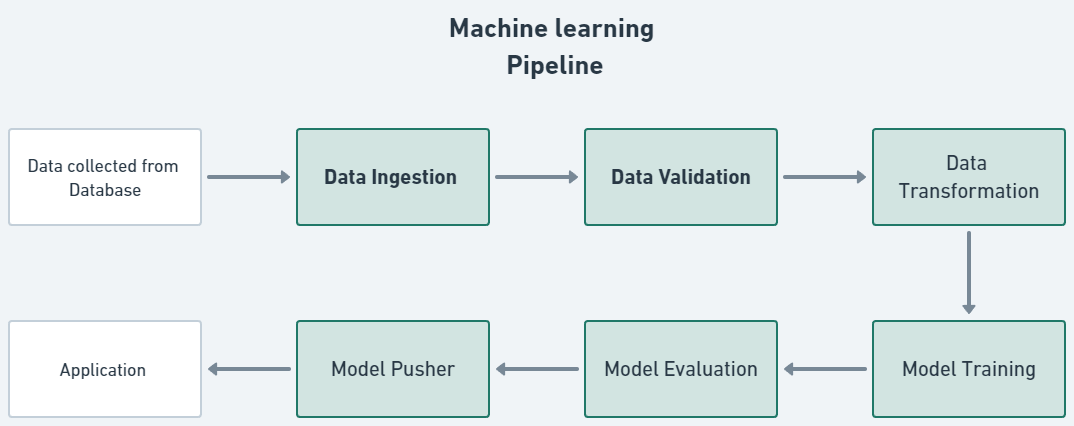
**Cloud Data Base**



Data csv file is downloaded from Mongo Data base

**Mongo\_DB**

**Training Pipeline**

****

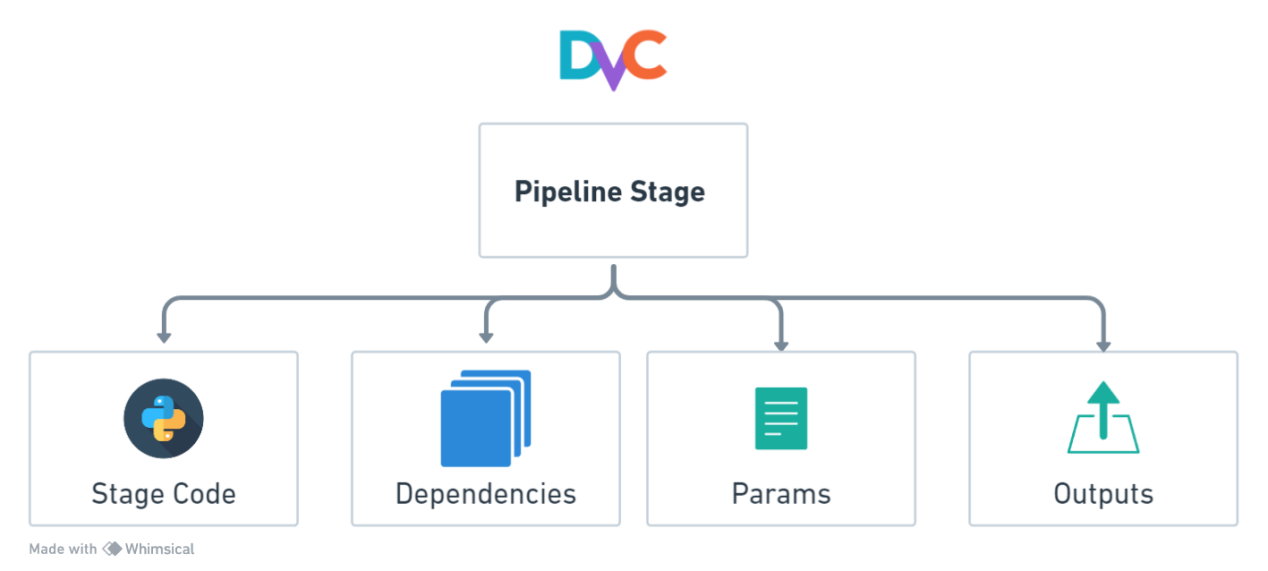
**Components of Pipeline**

* Data Ingestion
* Data Validation
* Data Transformation
* Model Training
* Model Evaluation
* Model Pusher

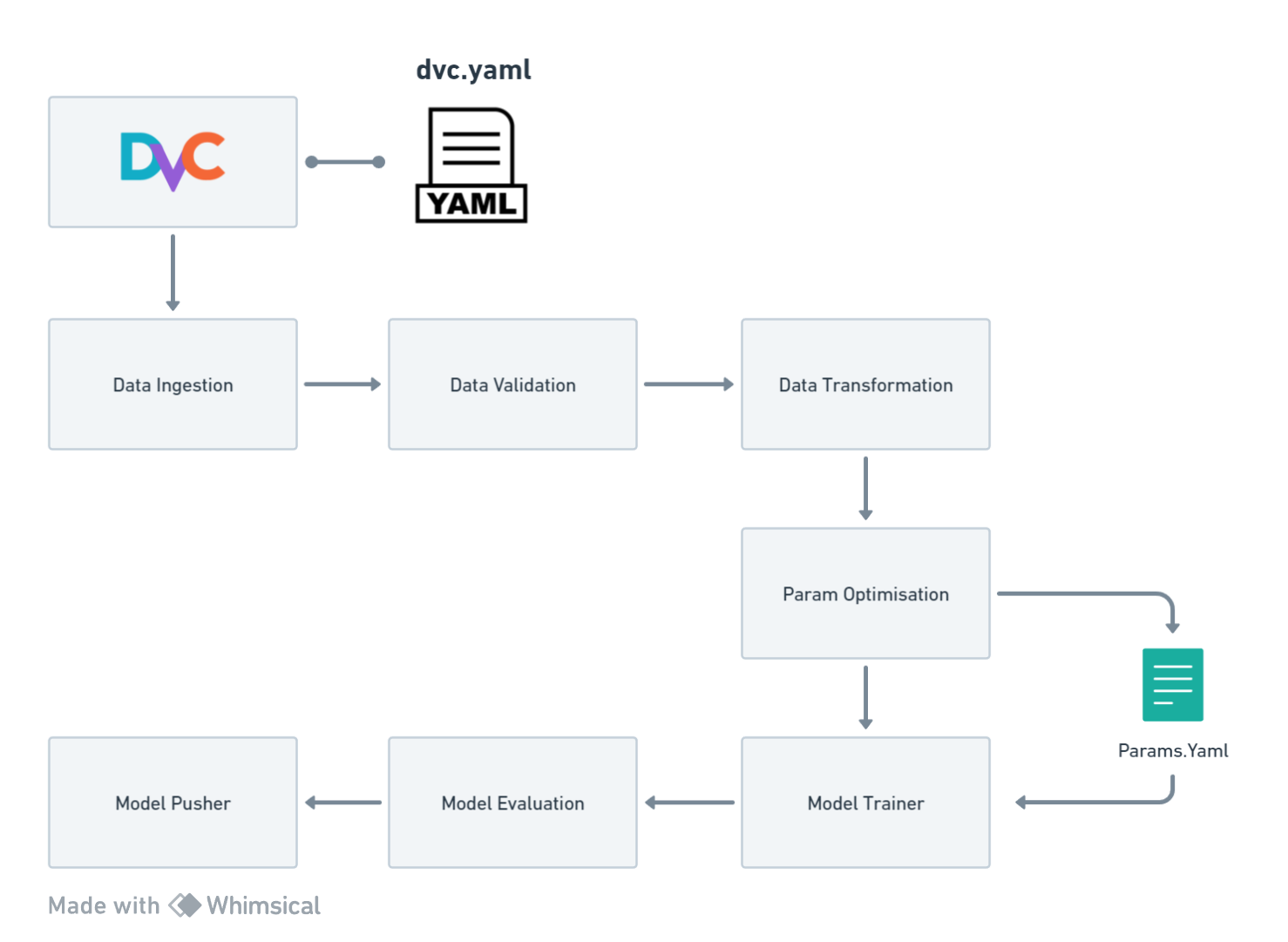
**Pipeline Orchestration**

Tool used - DVC

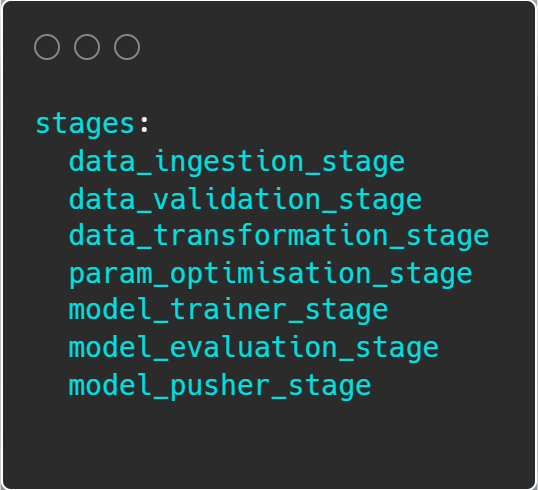
DVC components



Dvc Pipeline

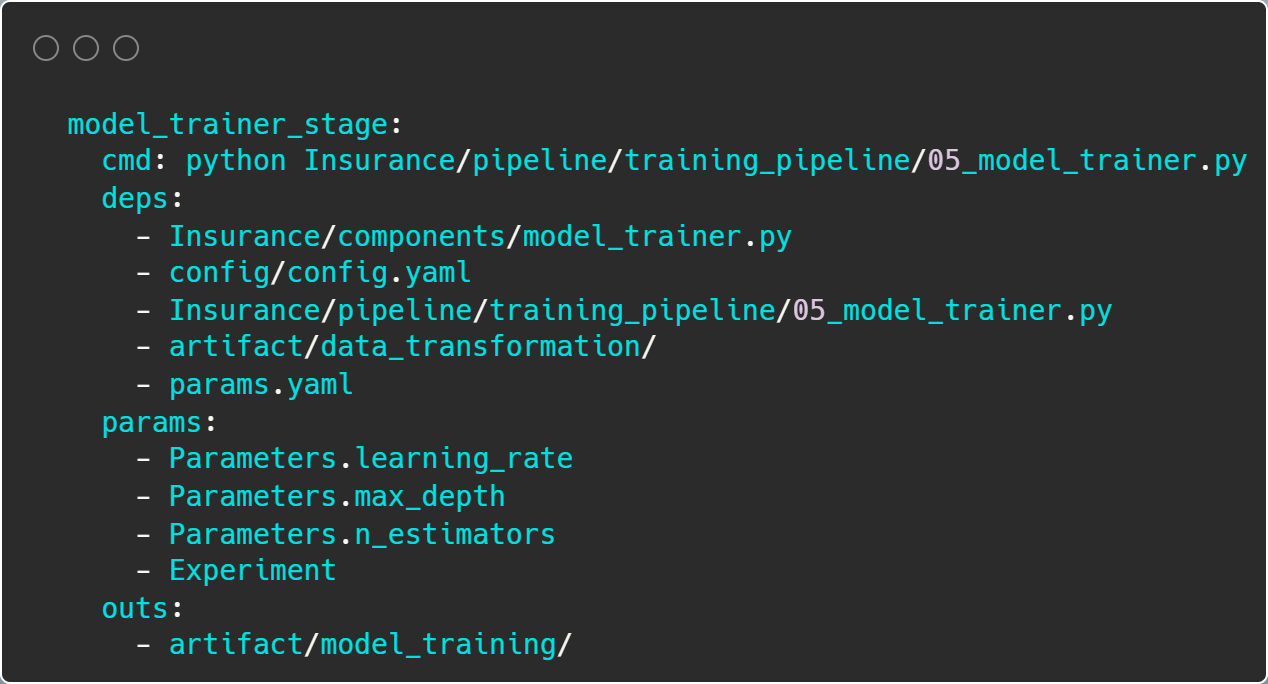


dvc.yaml

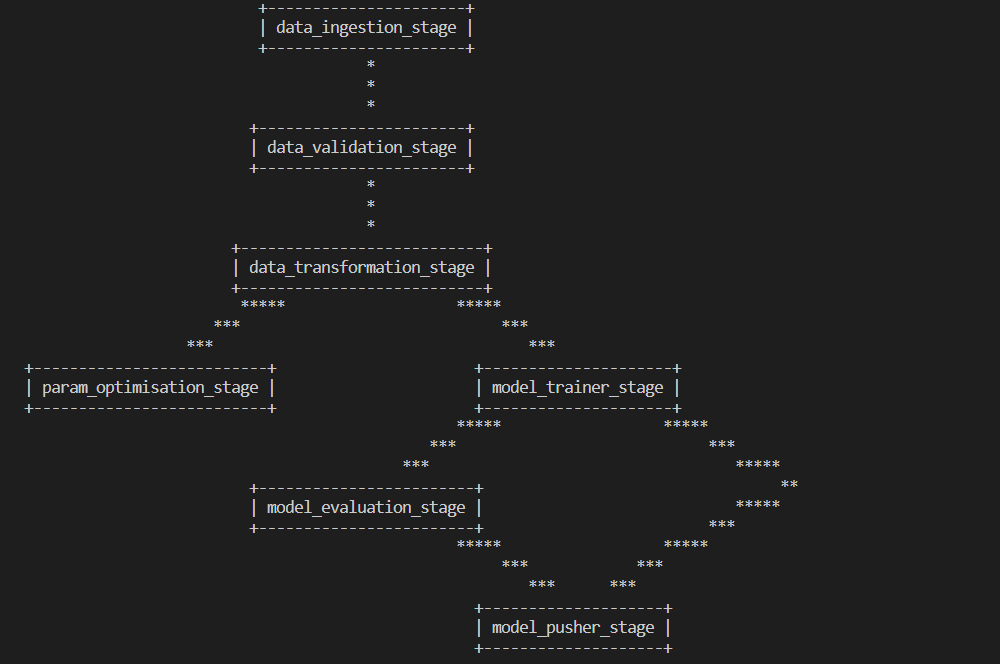


dvc.yaml stage Components

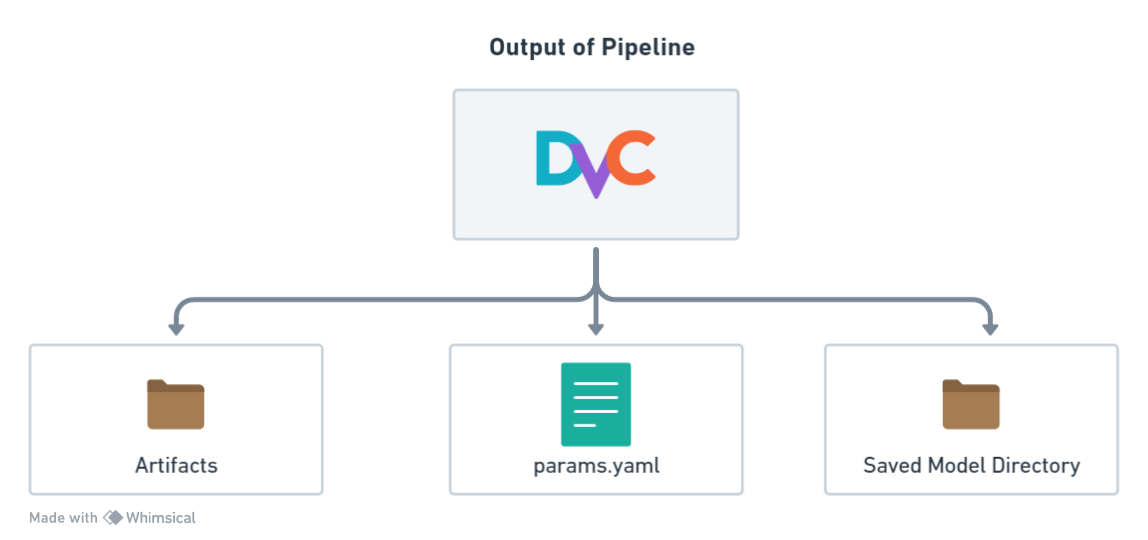
Example



DVC DAG



DVC pipeline output



**Artifact**

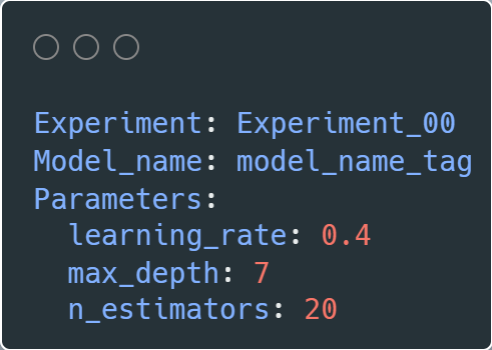
Output obtained are from per stage in Artifact Folder

1. Data Ingestion
2. Data Validation
3. Data Transformation
4. Param Optimisation
5. Model Trainer
6. Model Evaluation
7. Model Pusher

**Params.yaml**

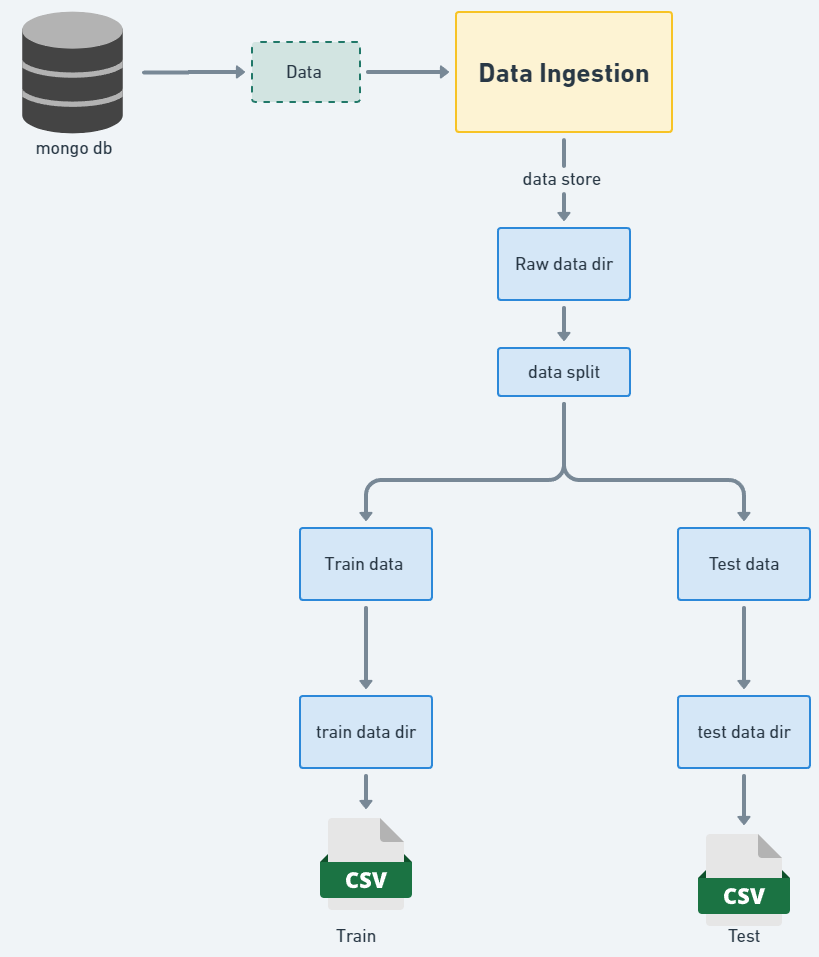
Output obtained from Param Optimization

Paramters if changed in this file will trigger the pipeline from model trainer stage.



**Saved Model Directory**

Once the model training and evaluation are complete, the best-performing model is stored in the current directory This model can then be utilized for local deployment.

**Data Ingestion** 

Data Ingestion class represents a data ingestion process that involves retrieving data from a data source, saving it in a raw data directory, and splitting it into training and test datasets.

The class code flows in following way:

Data Ingestion class takes a necessary elements and that provides the configuration settings for the data ingestion process.

It retrieves data from a data source and saves it in a raw data directory. It accesses the configuration settings to determine the location of the raw data directory and performs the necessary operations to obtain the data from the data source and saves it in the specified directory.

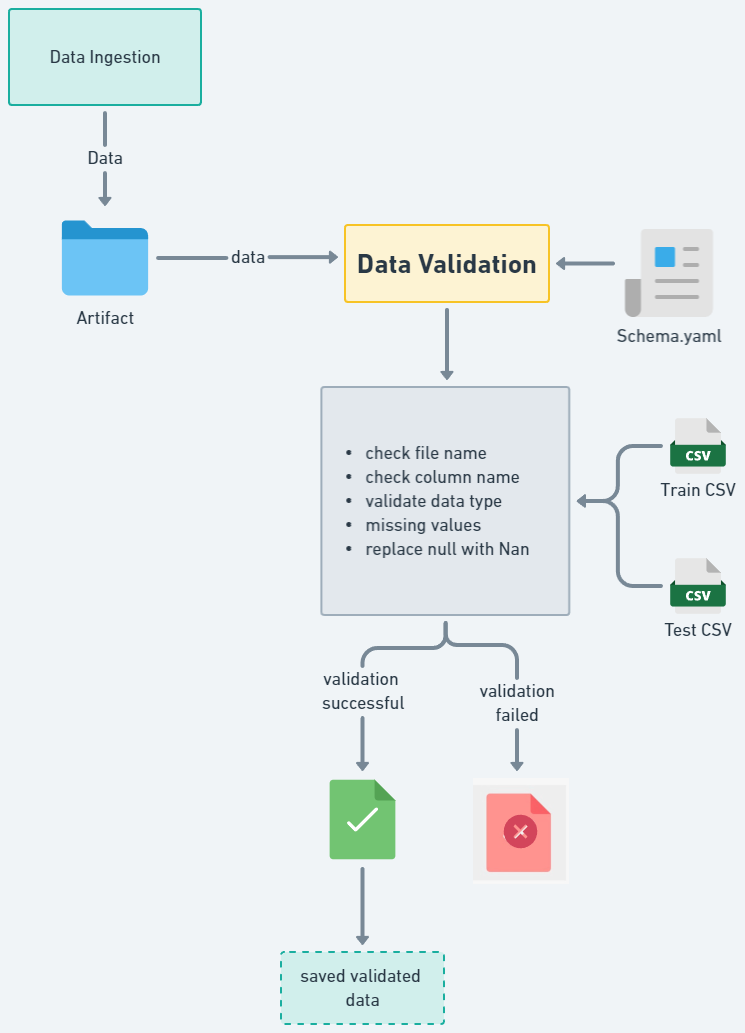
This method splits the raw data into training and test datasets. It reads the data from the raw data directory and split it into two sets:

a. training set

b. test set.

The method determines the split ratio based on the configuration settings.

Throughout the process, the class handles any exceptions that may occur and raises an application exception with an appropriate error message.

**Data Validation** 

This method aims to validate the training and test data stored in the artifact folder as a result of the data ingestion process carried out earlier. Its responsibility is to perform validation checks on the training and test datasets.

The method follows a series of steps to validate the data and logs the validation process. If the validation is successful, it exports the validated datasets to specified paths and returns the paths. However, if the validation fails, it raises an exception.

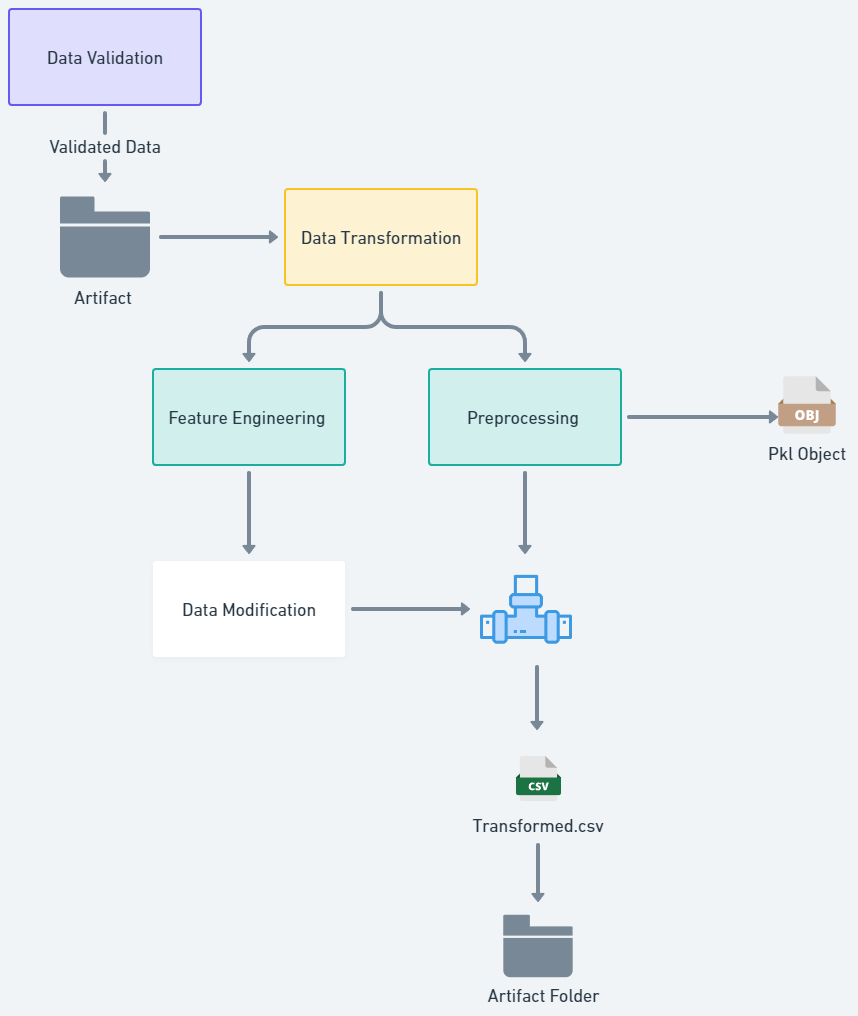
**Validation processes :**

* File name of the downloaded dataset
* Column labels
* Validating Data types
* Missing values whole column
* Replace null values

Each of these methods returns a boolean value as the result of the validation. If the dataset passes all the checks and evaluates to True, it is stored in the artifact folder.

Throughout the process, the class handles any exceptions that may occur and raises an application exception with an appropriate error message.

**Data Transformation**

****

Once the validated data is obtained from the artifact folder, it proceeds for the necessary transformations before being used for model training. These transformations ensure that the data is in a suitable format and structure for the training process.

**Pre\_processing Pipeline**

After feature engineering the data, the next step is pre\_processing . The pre\_processing pipeline applies a series of steps to the training and testing datasets, ensuring they are in a consistent and suitable format for model training.

The pre\_processing pipeline plays a critical role in optimizing the data for model training by standardizing and cleaning it. This enables the model to learn patterns effectively during training and make accurate predictions.

Overall, the data transformation stage ensures that the validated data undergoes necessary transformations, while feature engineering enhances the dataset by creating new features. The pre-processing pipeline then applies a series of standardized steps to prepare the data for model training.

**Output :**

Object Files

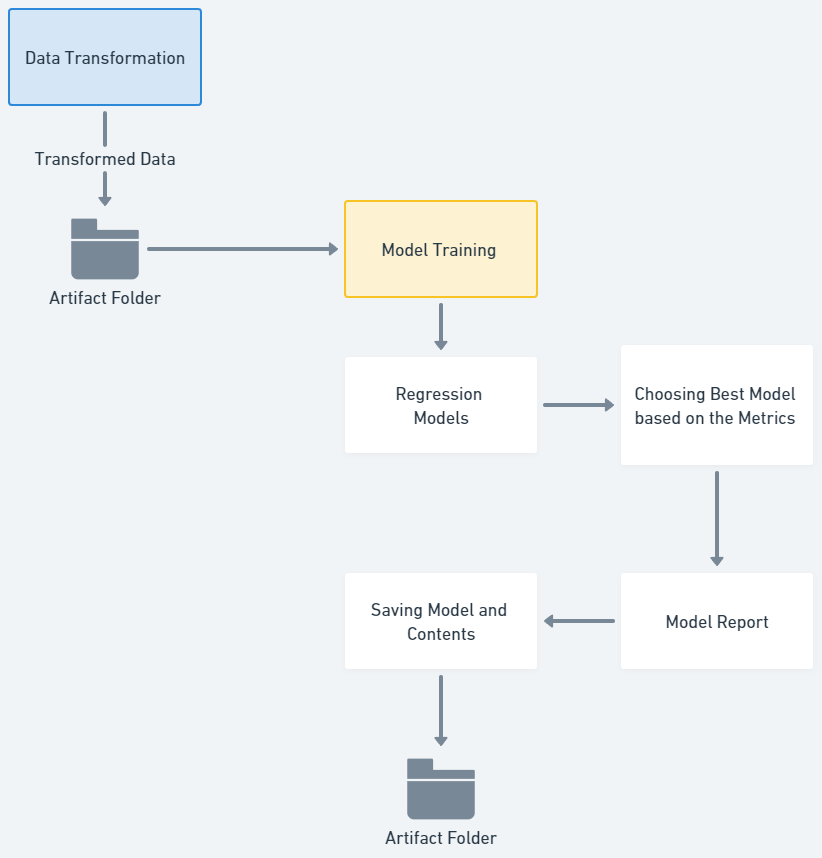
Pre\_processing pipeline ---> pkl object



Transformed Data

* Train Data
* Test Data

**Model Training**

****

Transformed train and test datasets is further used for model training

**Models:**

For this regression problem statement based on the Data analysis we detect few relevant models for model training :

* Gradient Boost
* XG boost
* Random Forest

**Hyper\_parameter Tuning and Grid Search CV:**

Hyper\_parameter tuning involves finding the optimal combination of hyper parameters for a given model.

Grid Search Cross-Validation (Grid Search CV) is a technique that exhaustively searches through a specified hyper\_parameter grid to find the best parameter values for the models. It systematically evaluates different hyper parameter combinations using cross-validation, ensuring the model's performance is robust .

**Metric used** : R\_2 score

**Output :**

**Model Object**

The best model, along with its corresponding best parameters, is saved for future use. This ensures that the optimal model can be easily retrieved and applied to new data.

**Model Report**

Report containing relevant information and insights is also saved. This report provides a concise summary of the model's performance and any additional context or analysis that may be necessary for understanding the model's results.

Model object and report is saved to the artifact directory

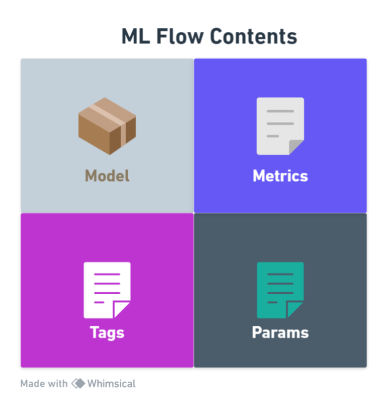
**Model Evaluation**

**Tools Used -** Mlflow Experimentation

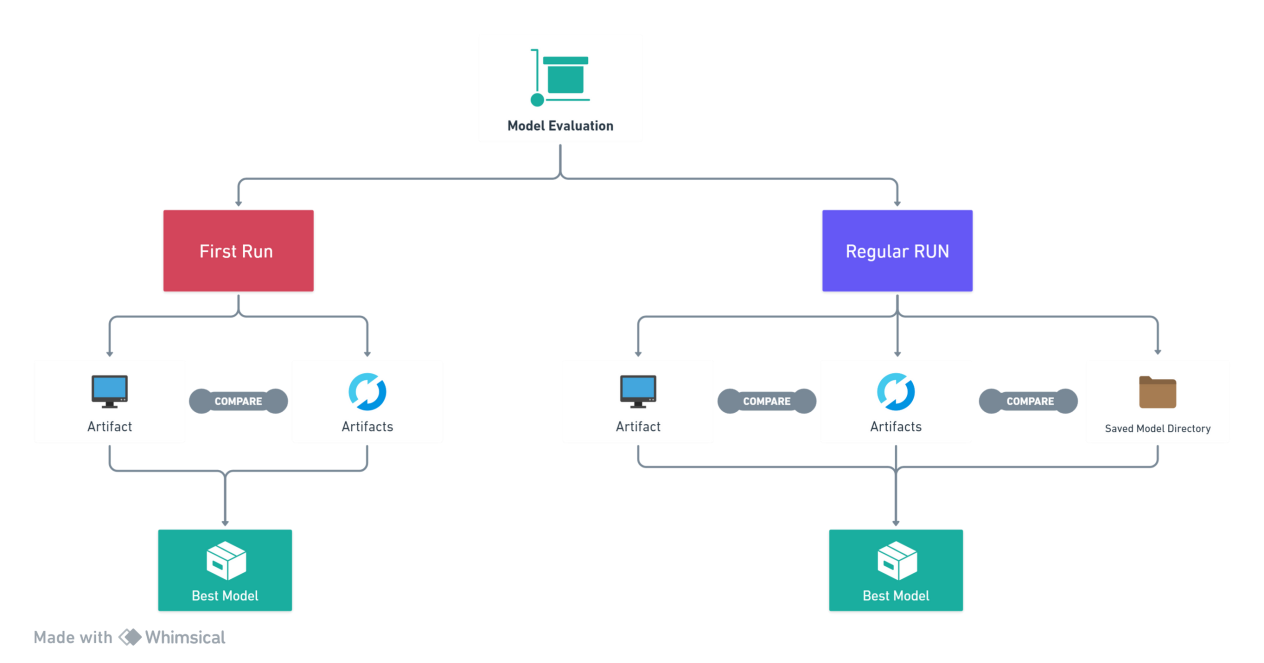
The saved model directory may contain earlier models and other necessary components from previous iterations.

During the evaluation process, we compare the recently trained artifact model with the previously saved models.

MLflow experimentation simplifies and enhances the process of conducting machine learning experiments. It enables users to systematically track, compare, and manage various runs of machine learning scripts, capturing input parameters, output metrics, and model artifacts.



**Model Evaluation**

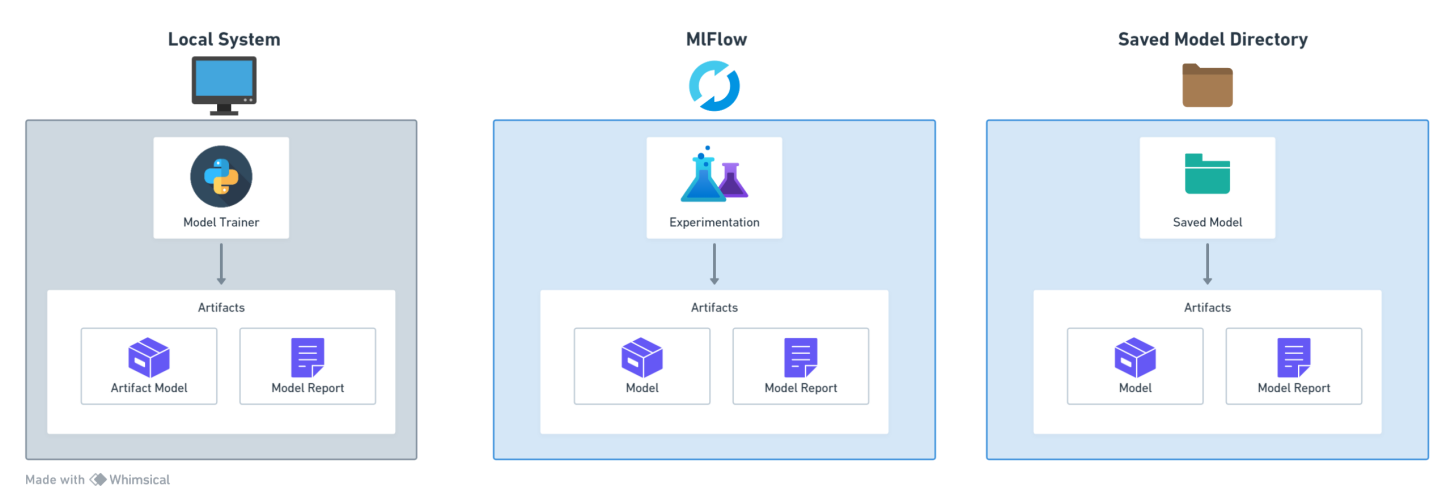


**During model evaluation, two cases may arise**.

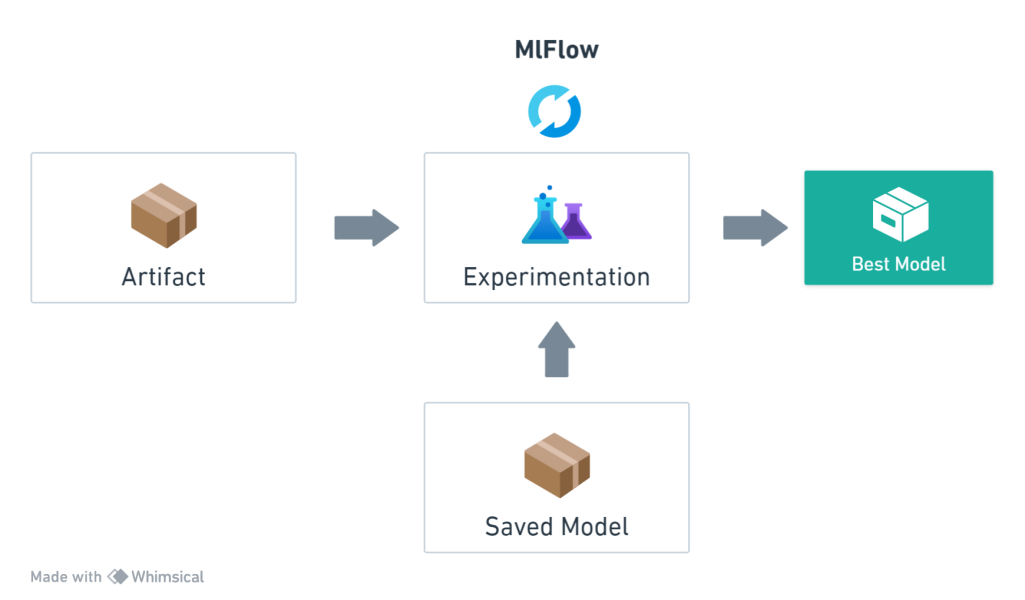
If the pipeline is triggered for the first time, the 'Saved Model' directory is empty. In this scenario, the model artifact is stored in the 'Saved Model' directory as the best model following evaluation from the MLflow experimentation stage.

Alternatively, if a saved model already exists in the current directory from a previously executed pipeline, it is incorporated into the experimentation list and subjected to comparison.

Evaluation Components



**Comparison Metrics for Model Selection**



**Components**

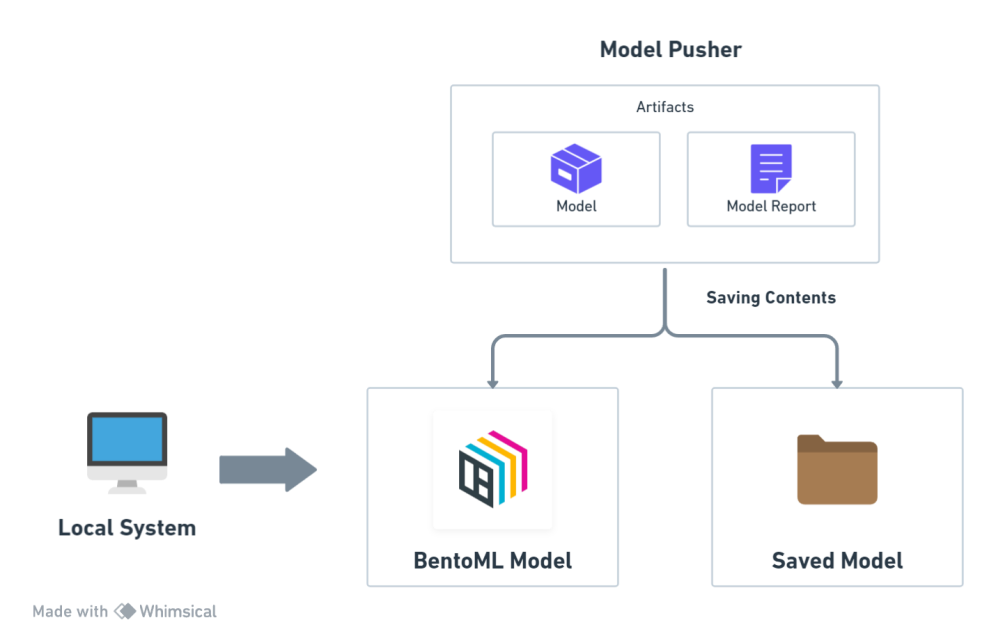
1. Trained model from previous Model trainer stage
2. Saved Model from the earlier runned pipeline added to experimention if exists

Best model is choosen from the Mlflow experimentation stage and saved for deployment

Once the selection is made, we save the chosen model along with its associated model object and a comprehensive model report. This approach ensures that we retain the most optimal model for future use.

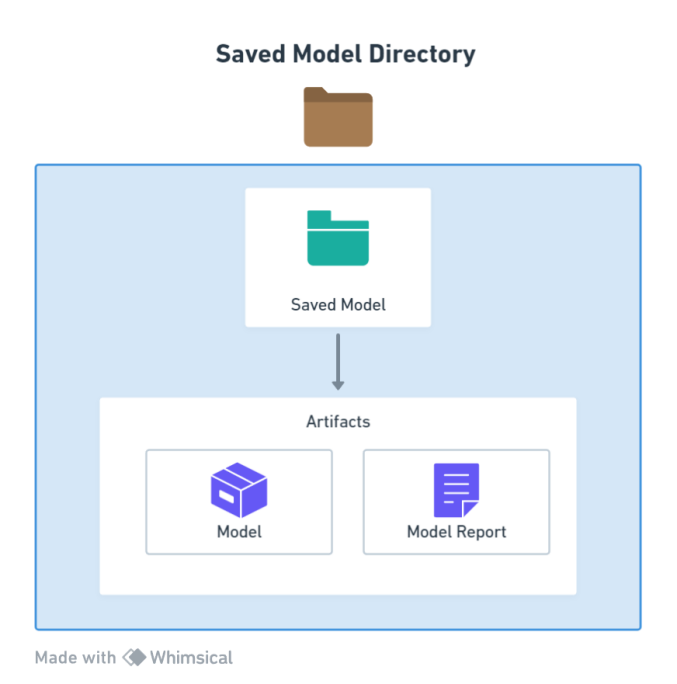
**Model Pusher**

Tools used - BentoML



**Saved Model Directory**

Model selected is stored in saved Model directory for any further cloud deployments or Flask UI access



Upon selecting the models and gathering the relevant information from the evaluation module, we proceed to push the chosen model and its corresponding report to the saved model directory. This ensures that the selected model is readily available for future processes.

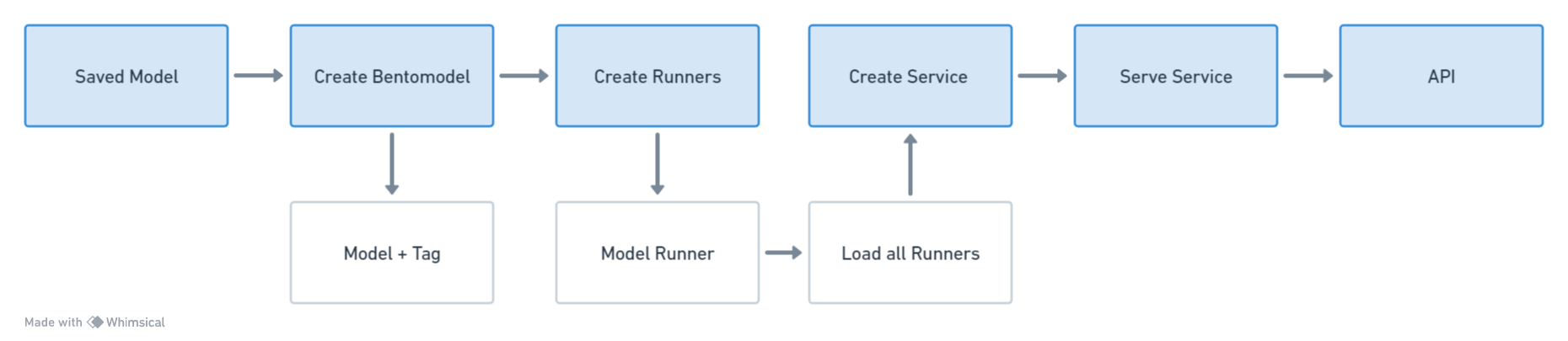
By storing it in the designated directory, we facilitate easy access and utilization of the model in subsequent tasks.

In this project we are working on BentoML it plays a pivotal role in facilitating the deployment of their machine learning models. By utilizing BentoML, we can effectively package their trained models along with the requisite dependencies into deployable artifacts.

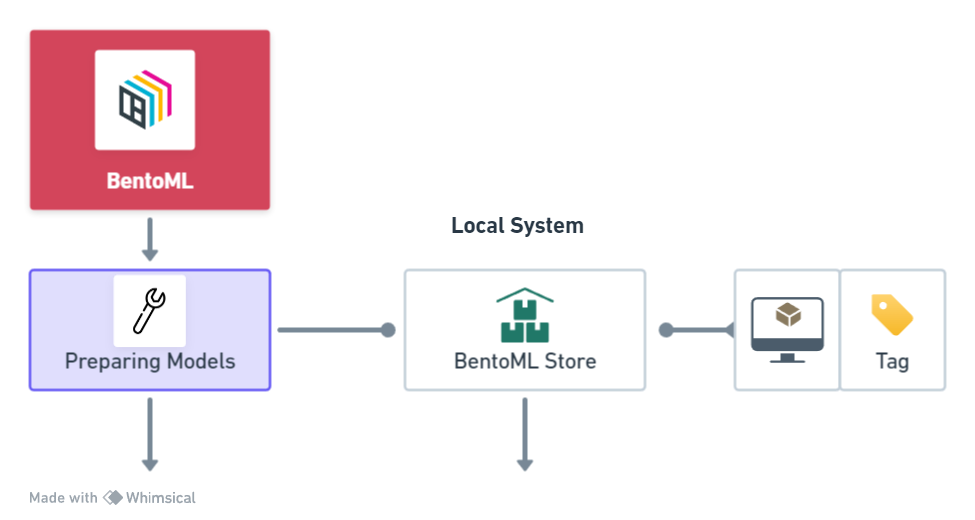
This capability streamlines the sharing and versioning of models, ensuring a consistent and reproducible deployment process across diverse platforms

Model selected from Model Evaluation is now used to create Bento Model in the local system this created model can be found in

BentoML

Workflow

**Create Bento Model of Selected Model**



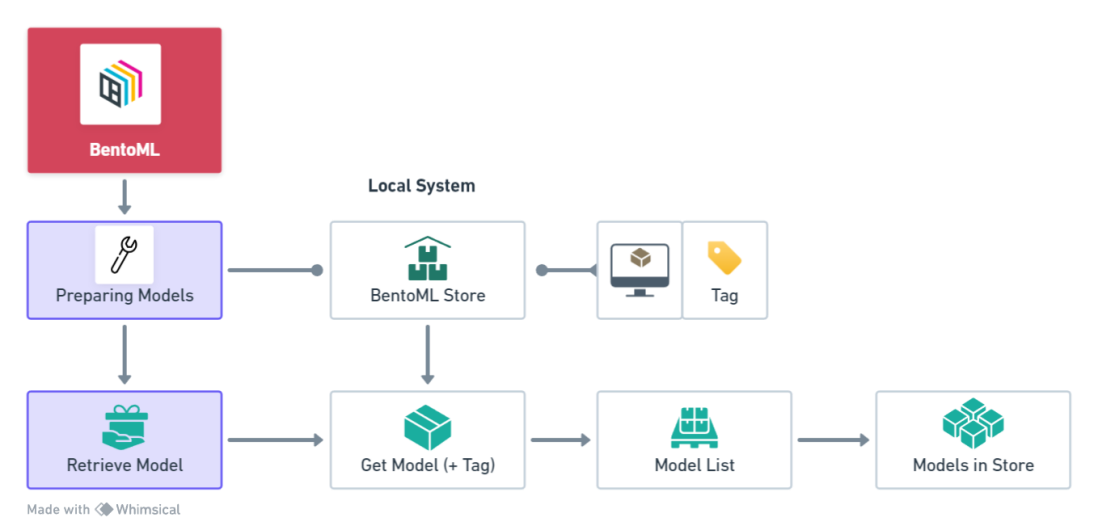
**Code for Bento model Creation**



Code location --

Insurance-->components-->deployment-->bentoml\_folder

Model Created is stored within the local system with name “sklearn\_model” and tag



Models with their names and tags can be accessed from the terminal using code,

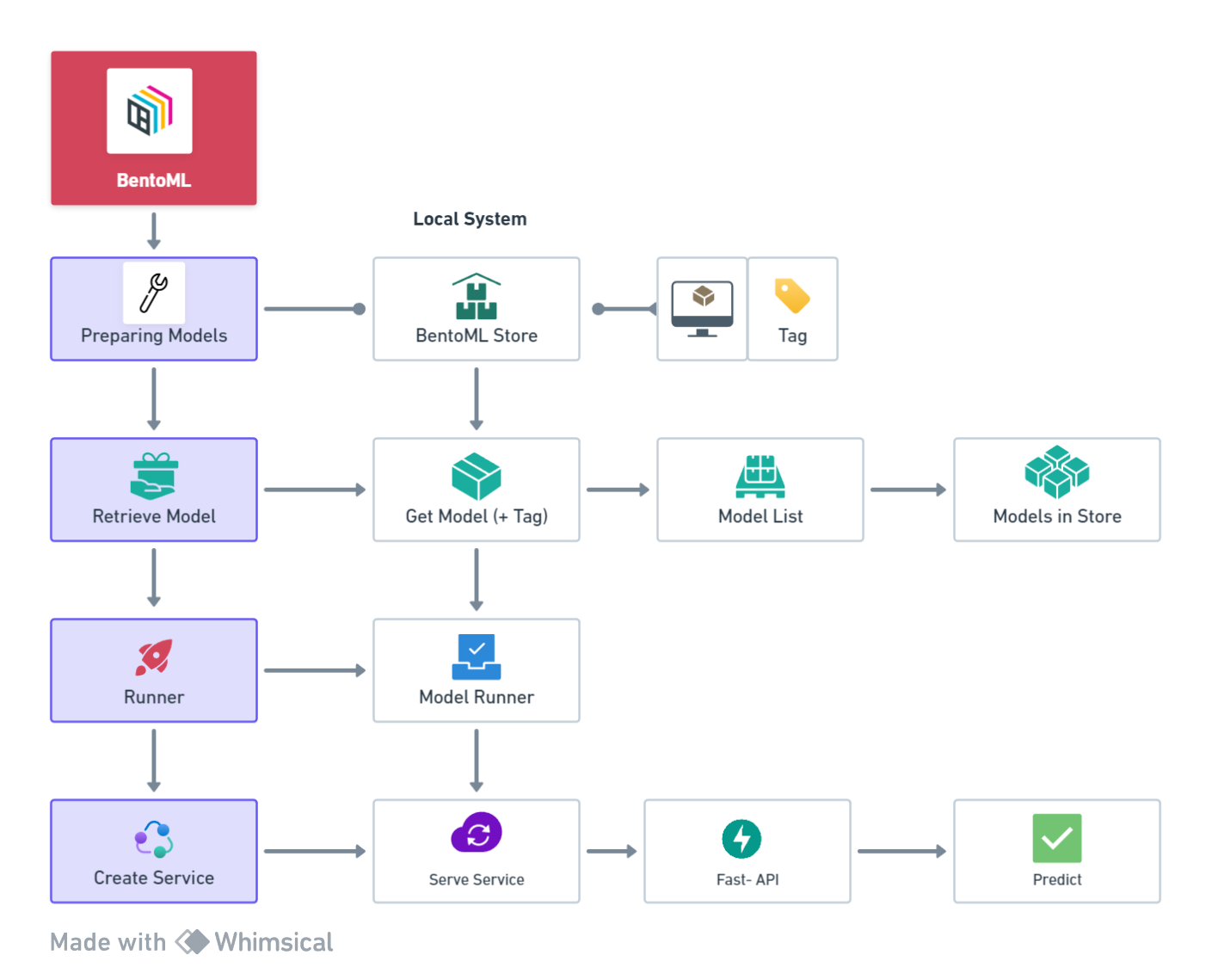
**Output**

model\_names:tag



Bentomodel from the list can be accessed with tag and added to the runner

Model stored location - local disc c --> users--> user\_name-->bentoml folder



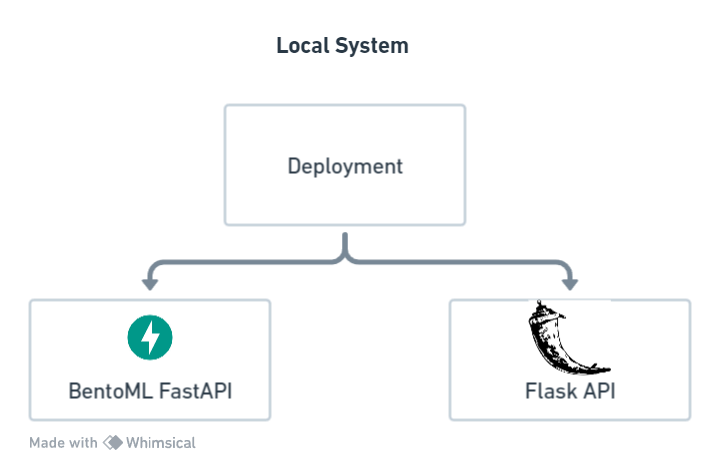
Code Example for setting up runner and Running Service for local Deployment



Code location

Current\_directory --> bento.py

**Deployment (Local)**



**BentoML Fast API (Swagger UI)**

Models stored in the local directory are served to be locally deployed



1. Running the Service

**bento.py** file consists of function to add model to runner and create a service

Command in terminal - **python bento.py**

Model with tagged is accessed and added to runner

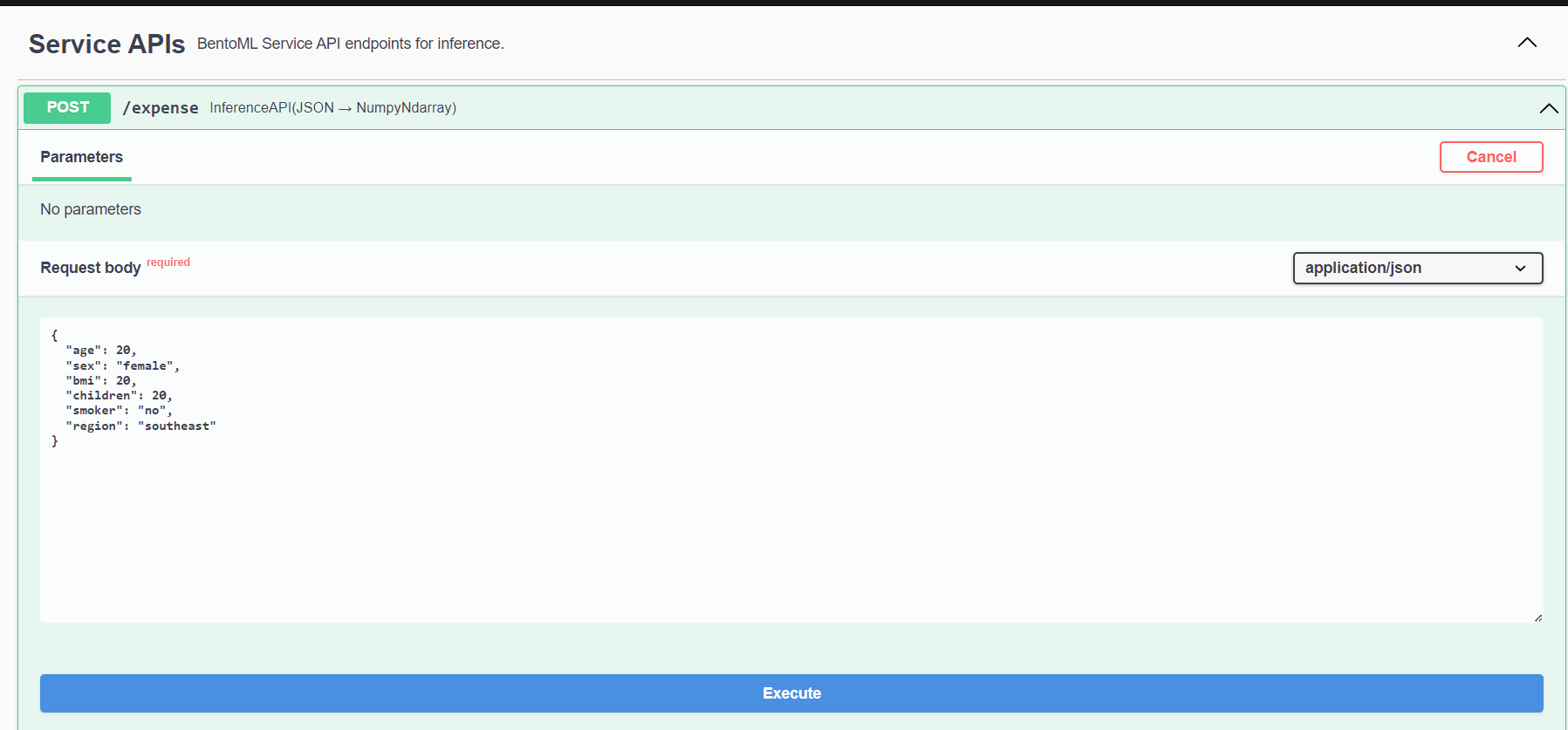
1. Serving the model

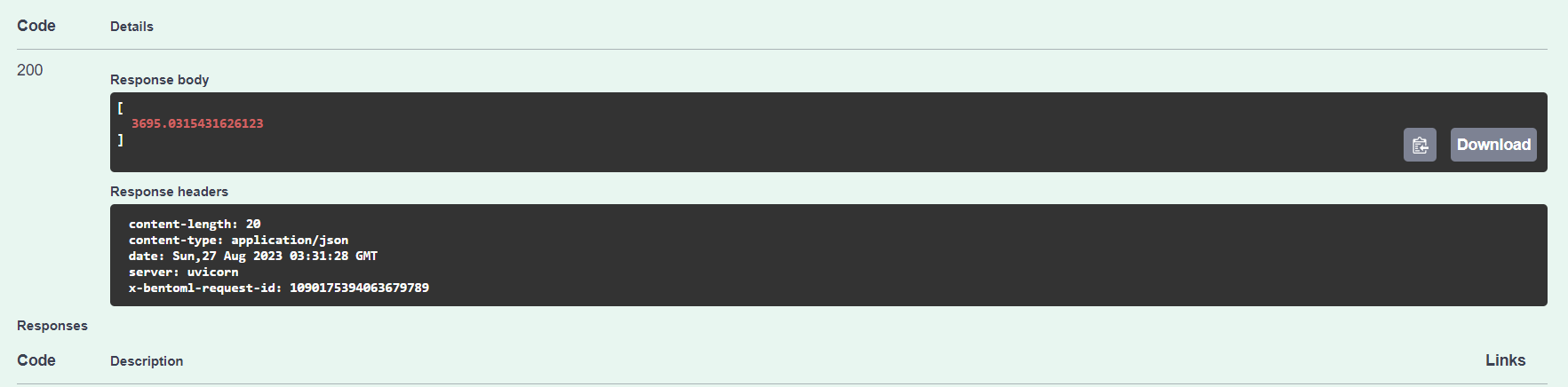
Command (terminal) **- bentoml serve bento.py:svc**

This code runs the service named “svc”

Ui can be accessed at **- “http://localhost:3000”**

**UI**



**Output**

**Code Execution**

1.Enter mongodb necessary credentials in .env file

2.Dump csv from local system to Mongo Data base data\_dump.py

Database name

Collection name

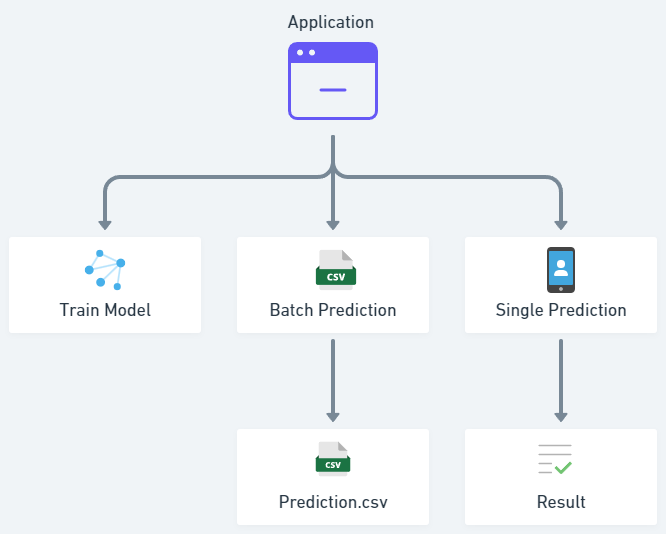
1. Update database and collection name accordingly in config.yaml for data ingestion from dumped location
   1. Command- **python data\_dump.py**
2. DVC
   1. Install dvc
   2. Update code to git repository (Important)
   3. Add dvc using dvc.init
   4. Run dvc pipeline from dvc.yaml command - **dvc repro**

Force run command ---> **dvc repro -- force**

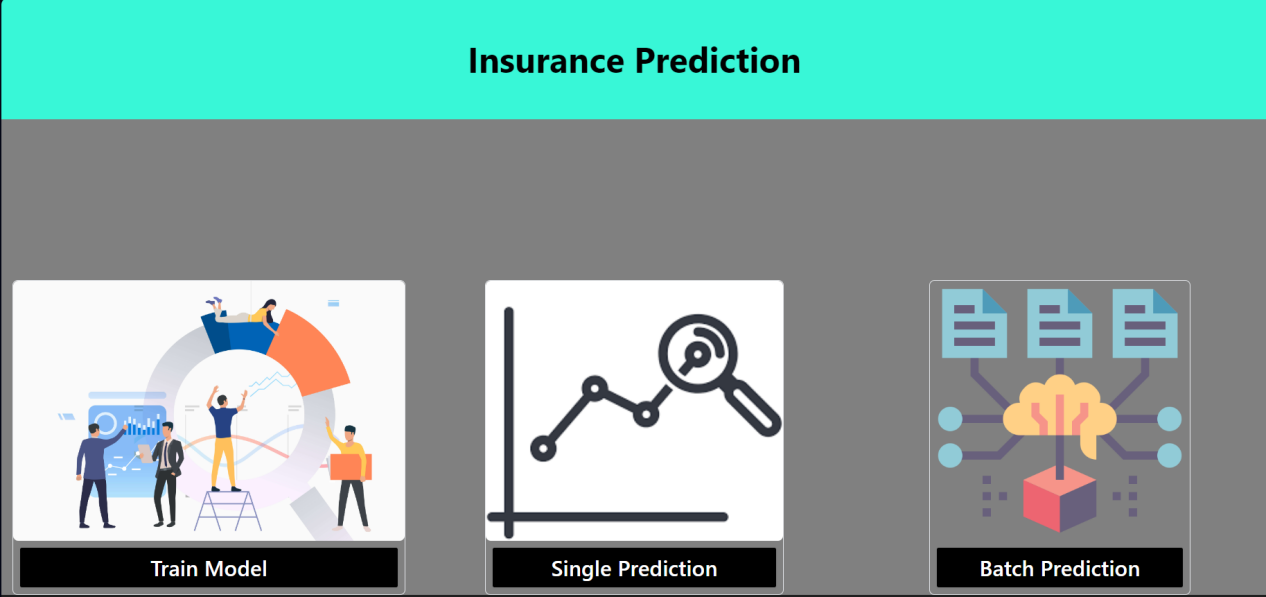
1. Params.yaml file is saved In current directory , parameters changed will trigger pipeline train model based on the given parameters
2. Models are evaluated using Mlflow experimentation and can be viewed
   1. Command - “**mlflow ui”**
3. Selected model is saved in directory and bento model is created
4. Bento model is saved in the local system as model pusher executes from the dvc pipeline
5. All saved models can be viewed from the terminal - **bentoml models list**
   1. Can be also accessed in location - **local\_disc c --> users --> user\_name --> bentoml folder**
6. Model can be accessed with the associated tag , in this project we take the latest model with “**latest**” tag
7. To create service and deploy the model locally
   1. loads the runner --> Command - **python bento.py**
   2. Model Serving - Command - **bentoml serve bento.py:svc**
      1. “svc” is the name of the service in the bento.py python file
8. Url address
   1. **“http://localhost:3000”**
9. Click on Try and enter the necessary input and click execute

**Flask API**

Model Stored in the saved Model Directory is used for flask deployment

**Application** 

**UI**



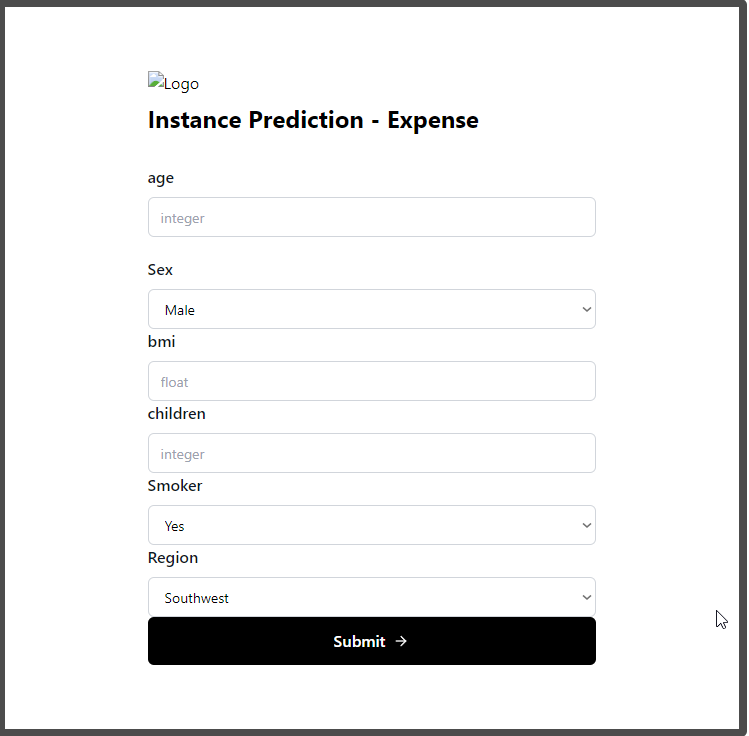
**Train Model**

This triggers an entire training pipeline to train the model

**Instance Prediction**

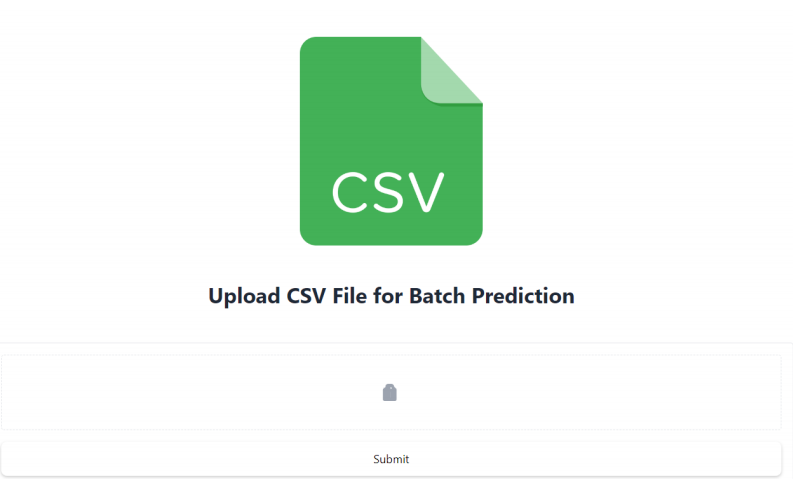
After training the model and saving the necessary contents and information, the system utilizes the user's input to generate predicted results.

These results are then displayed on an HTML web page for easy access and visualization



**Batch Prediction**

In batch prediction, the user provides a CSV file as input, and the system generates a prediction CSV file. The resulting file is then stored at a specified location and can also be uploaded to a MongoDB database for further use or analysis.



**Code Execution (Flask UI)**

1. Enter mongodb necessary credentials in **.env** file
2. Dump csv from local system to Mongo Data base
   * **data\_dump.py**
     + Database name
     + Collection name
3. Update database and collection name accordingly in **config.yaml** for data ingestion from dumped location
4. Run the training pipeline either from terminal or Html UI
   1. Terminal
      1. Python demo.py
   2. UI
      1. Python app.py
      2. Click train pipeline
5. Batch prediction
   1. Run UI ---> python app.py
   2. Click on batch prediction
   3. Upload csv and click submit
   4. Batch prediction file is saved in mentioned location at app.py code
   5. Prediction.csv file is uploaded to the mongo database ( collection and database tags can be changed via app.py code)
6. Instance Prediction
   1. Enter the input feature details
   2. Click submit
   3. Result displayed on html page

**Project Summary**

This project centers on an extensive insurance dataset encompassing vital details such as age, gender, BMI, dependents, smoking tendencies, and geographic location of insured individuals. The core objective of this endeavor is to analyze the dataset, uncover the variables influencing medical expenses, and formulate predictive models that facilitate fine-tuned insurance coverage based on individual characteristics. The pivotal 'expenses' column serves as the dependent variable, denoting the medical expenditures accrued by each individual.

**Key Features:**

**Predictive Model Development**: Employing advanced regression models including Gradient Boosting, XGBoost, and Random Forest, accurate predictors for medical expenses have been constructed.

**Data Integration and Storage**: A streamlined machine learning pipeline has been established, seamlessly interfacing with a MongoDB database. This configuration ensures efficient data ingestion and seamless uploading of prediction outcomes.

**Pipeline Orchestration**: The orchestration of the pipeline is facilitated by the Data Version Control (DVC) system, which ensures consistent and structured data management.

**Model Evaluation and Tracking**: Through the utilization of Mlflow, comprehensive experimentation has been conducted to assess model performance. The selected performance metric is the R2 score, providing valuable insights into the predictive capabilities of the models.

**Deployment**: The resulting project offers the flexibility of deployment within both Flask and FastAPI frameworks, catering to a variety of deployment environments. Furthermore, BentoML serves as a user-friendly interface for streamlined model deployment.