MLOps Project with DVC, Git & MLflow

Complete Integration Guide

A Comprehensive Guide to Building Production MLOps Pipelines with DVC Version Control, Git Integration, and MLflow Experiment Tracking via DagShub

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to MLOps with DagShub

1.1 What is This Project About?

This comprehensive guide demonstrates a complete MLOps workflow that integrates three powerful tools:

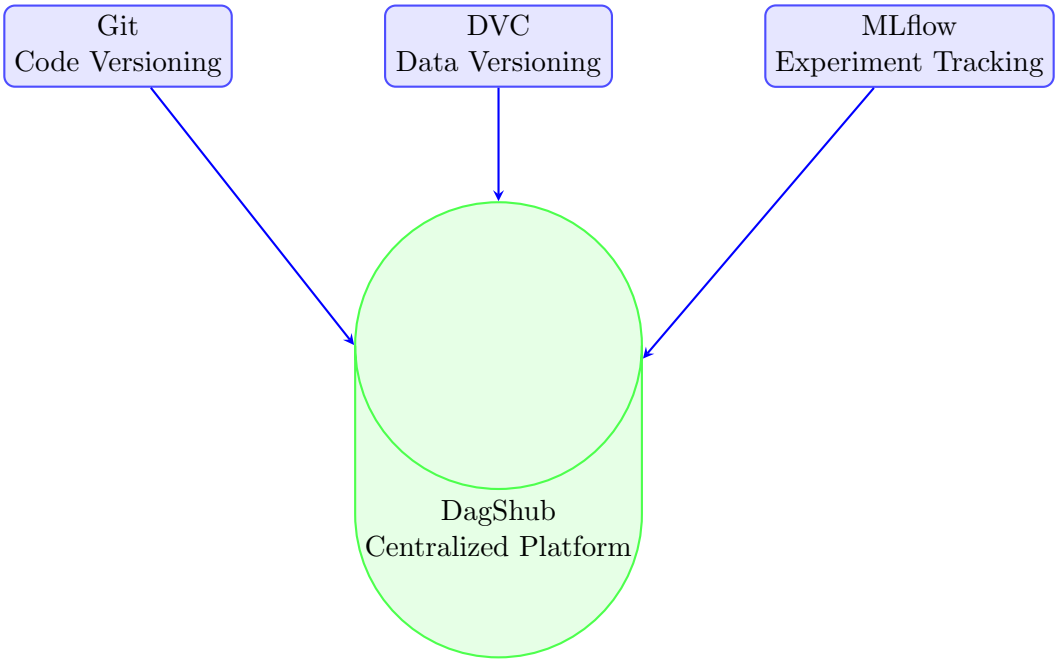
1. **DVC (Data Version Control)**: For versioning data, models, and pipeline artifacts
2. **Git**: For versioning code and metadata
3. **MLflow via DagShub**: For experiment tracking and model registry

Project Goals

By the end of this guide, you will:

- Build a complete ML pipeline with proper logging
- Version control data and models with DVC
- Track experiments with MLflow on DagShub
- Compare multiple model runs systematically
- Freeze and deploy the best performing model
- Enable team collaboration and reproducibility

1.2 The MLOps Stack



1.3 What Makes This Different?

Unlike the previous DVC-only workflow, this project adds:

- **Centralized Experiment Tracking**: MLflow UI hosted on DagShub

- **Team Collaboration**: Everyone sees all experiments
- **Model Comparison**: Visual comparison of multiple runs
- **Experiment Tagging**: Link Git commits to MLflow runs
- **Model Registry**: Track model lineage and deployment status

1.4 Key Differences from Previous Workflow

DVC-Only Workflow	**DVC + MLflow Workflow**
Local experiment tracking	Centralized MLflow tracking
Manual metric comparison	Visual metric comparison
Experiments private to user	All experiments visible to team
Limited experiment history	Full experiment history preserved
No model registry	MLflow model registry

2 Initial Project Setup

2.1 Step 1: Create Python Environment

```
# Create virtual environment
python -m venv venv
```

2.2 Step 2: Activate Environment

```
# Windows
venv\Scripts\activate.bat

# Linux/Mac
source venv/bin/activate
```

2.3 Step 3: Create Source Directory

```
mkdir src
```

2.4 Step 4: Create Requirements File

requirements.txt

```
dvc
numpy
pandas
matplotlib
wordcloud
nltk
scikit-learn
xgboost
pyyaml
mlflow
dagshub
```

Important Note

New Dependencies:

- **mlflow**: Experiment tracking library
- **dagshub**: DagShub integration for MLflow
- **xgboost**: Additional model option (if needed)

2.5 Step 5: Install Requirements

```
pip install -r requirements.txt
```

3 Building Pipeline Components

3.1 Component Overview

The ML pipeline consists of five stages:

1. **Data Ingestion**: Load and split data
2. **Data Pre-Processing**: Clean and transform text
3. **Feature Engineering**: Apply TF-IDF
4. **Model Building**: Train RandomForest
5. **Model Evaluation**: Calculate metrics and log to MLflow

3.2 Data Ingestion Module

src/Data_Ingestion.py

```python
import pandas as pd
import os
from sklearn.model_selection import train_test_split
import logging
import yaml

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Data_Ingestion')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Data_Ingestion.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s',
    params_path)
```

```
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise


def load_data(data_url: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(data_url)
        logger.debug('Data loaded from %s', data_url)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s',
    e)
        raise


def preprocess_data(df: pd.DataFrame) -> pd.DataFrame:
    """Preprocess the data."""
    try:
        df.drop(columns=['Unnamed: 2', 'Unnamed: 3', 'Unnamed:
   4'],
                inplace=True)
        df.rename(columns={'v1': 'target', 'v2': 'text'},
                  inplace=True)
        logger.debug('Data preprocessing completed')
        return df
    except KeyError as e:
        logger.error('Missing column in dataframe: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error during preprocessing: %s
   ', e)
        raise


def save_data(train_data: pd.DataFrame,
              test_data: pd.DataFrame,
              data_path: str) -> None:
    """Save the train and test datasets."""
    try:
        raw_data_path = os.path.join(data_path, 'raw')
        os.makedirs(raw_data_path, exist_ok=True)

        train_data.to_csv(
            os.path.join(raw_data_path, "train.csv"),
```

```
            index=False
        )
        test_data.to_csv(
            os.path.join(raw_data_path, "test.csv"),
            index=False
        )
        logger.debug('Train and test data saved to %s',
                     raw_data_path)
    except Exception as e:
        logger.error('Error while saving data: %s', e)
        raise


def main():
    try:
        params = load_params(params_path='params.yaml')
        test_size = params['data_ingestion']['test_size']

        data_path = 'https://raw.githubusercontent.com/' \
                    'Error-Makes-Clever/MLOPS-Data-Versioning-' \
    \
                    'using-DVC-Project-1/refs/heads/main/' \
                    'Experiments/spam.csv'

        df = load_data(data_url=data_path)
        final_df = preprocess_data(df)

        train_data, test_data = train_test_split(
            final_df, test_size=test_size, random_state=2
        )

        save_data(train_data, test_data, data_path='./data')
    except Exception as e:
        logger.error('Failed to complete data ingestion: %s', e
    )
        print(f"Error: {e}")


if __name__ == '__main__':
    main()
```

3.3 Data Pre-Processing Module

src/Data_Pre_Processing.py - Part 1

```
import os
import logging
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from nltk.stem.porter import PorterStemmer
from nltk.corpus import stopwords
import string
import nltk

```

```
nltk.download('stopwords')
nltk.download('punkt')

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Setting up logger
logger = logging.getLogger('Data_Pre_Processing')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Data_Pre_Processing.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def transform_text(text):
    """
    Transform text: lowercase, tokenize, remove stopwords,
    punctuation, and stem.
    """
    ps = PorterStemmer()

    # Convert to lowercase
    text = text.lower()

    # Tokenize
    text = nltk.word_tokenize(text)

    # Remove non-alphanumeric tokens
    text = [word for word in text if word.isalnum()]

    # Remove stopwords and punctuation
    text = [word for word in text
            if word not in stopwords.words('english')
            and word not in string.punctuation]

    # Stem the words
    text = [ps.stem(word) for word in text]

    # Join back into string
    return " ".join(text)

```

```
65
66 def preprocess_df(df, text_column='text',
67                    target_column='target'):
68     """
69     Preprocess DataFrame: encode target, remove duplicates,
70     transform text.
71     """
72     try:
73         logger.debug('Starting preprocessing for DataFrame')
74
75         # Encode the target column
76         encoder = LabelEncoder()
77         df[target_column] = encoder.fit_transform(df[
    target_column])
78         logger.debug('Target column encoded')
79
80         # Remove duplicate rows
81         df = df.drop_duplicates(keep='first')
82         logger.debug('Duplicates removed')
83
84         # Apply text transformation
85         df.loc[:, text_column] = df[text_column].apply(
86             transform_text
87         )
88         logger.debug('Text column transformed')
89
90         return df
91     except KeyError as e:
92         logger.error('Column not found: %s', e)
93         raise
94     except Exception as e:
95         logger.error('Error during text normalization: %s', e)
96         raise
97
98
99 def main(text_column='text', target_column='target'):
100     """
101     Main function: load raw data, preprocess, save processed.
102     """
103     try:
104         # Load data from data/raw
105         train_data = pd.read_csv('./data/raw/train.csv')
106         test_data = pd.read_csv('./data/raw/test.csv')
107         logger.debug('Data loaded properly')
108
109         # Transform the data
110         train_processed = preprocess_df(train_data,
111                                         text_column,
112                                         target_column)
113         test_processed = preprocess_df(test_data,
114                                        text_column,
115                                        target_column)
116
117         # Store in data/interim
118         data_path = os.path.join("./data", "interim")
119         os.makedirs(data_path, exist_ok=True)
```

```
120
121         train_processed.to_csv(
122             os.path.join(data_path, "train_processed.csv"),
123             index=False
124         )
125         test_processed.to_csv(
126             os.path.join(data_path, "test_processed.csv"),
127             index=False
128         )
129
130         logger.debug('Processed data saved to %s', data_path)
131     except FileNotFoundError as e:
132         logger.error('File not found: %s', e)
133     except pd.errors.EmptyDataError as e:
134         logger.error('No data: %s', e)
135     except Exception as e:
136         logger.error('Failed to complete data transformation: %
    s', e)
137         print(f"Error: {e}")
138
139
140 if __name__ == '__main__':
141     main()
```

3.4 Feature Engineering Module

src/Feature_Engineering.py

```
1  import pandas as pd
2  import os
3  from sklearn.feature_extraction.text import TfidfVectorizer
4  import logging
5  import yaml
6
7  # Ensure the "logs" directory exists
8  log_dir = 'logs'
9  os.makedirs(log_dir, exist_ok=True)
10
11 # Logging configuration
12 logger = logging.getLogger('Feature_Engineering')
13 logger.setLevel('DEBUG')
14
15 console_handler = logging.StreamHandler()
16 console_handler.setLevel('DEBUG')
17
18 log_file_path = os.path.join(log_dir, 'Feature_Engineering.log'
    )
19 file_handler = logging.FileHandler(log_file_path)
20 file_handler.setLevel('DEBUG')
21
22 formatter = logging.Formatter(
23     '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
24 )
25 console_handler.setFormatter(formatter)
```

```
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s',
    params_path)
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise


def load_data(file_path: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        df.fillna('', inplace=True)
        logger.debug('Data loaded and NaNs filled from %s',
                     file_path)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s',
    e)
        raise


def apply_tfidf(train_data: pd.DataFrame,
                test_data: pd.DataFrame,
                max_features: int) -> tuple:
    """Apply TF-IDF to the data."""
    try:
        vectorizer = TfidfVectorizer(max_features=max_features)

        X_train = train_data['text'].values
        y_train = train_data['target'].values
        X_test = test_data['text'].values
        y_test = test_data['target'].values

        X_train_bow = vectorizer.fit_transform(X_train)
        X_test_bow = vectorizer.transform(X_test)
```

```
80
81          train_df = pd.DataFrame(X_train_bow.toarray())
82          train_df['label'] = y_train
83
84          test_df = pd.DataFrame(X_test_bow.toarray())
85          test_df['label'] = y_test
86
87          logger.debug('TF-IDF applied and data transformed')
88          return train_df, test_df
89      except Exception as e:
90          logger.error('Error during TF-IDF transformation: %s',
    e)
91          raise
92
93
94  def save_data(df: pd.DataFrame, file_path: str) -> None:
95      """Save the dataframe to a CSV file."""
96      try:
97          os.makedirs(os.path.dirname(file_path), exist_ok=True)
98          df.to_csv(file_path, index=False)
99          logger.debug('Data saved to %s', file_path)
100     except Exception as e:
101         logger.error('Unexpected error while saving data: %s',
    e)
102         raise
103
104
105 def main():
106     try:
107         params = load_params(params_path='params.yaml')
108         max_features = params['feature_engineering']['
    max_features']
109
110         train_data = load_data('./data/interim/train_processed.
    csv')
111         test_data = load_data('./data/interim/test_processed.
    csv')
112
113         train_df, test_df = apply_tfidf(train_data, test_data,
114                                         max_features)
115
116         save_data(train_df,
117                   os.path.join("./data", "processed",
118                                "train_tfidf.csv"))
119         save_data(test_df,
120                   os.path.join("./data", "processed",
121                                "test_tfidf.csv"))
122     except Exception as e:
123         logger.error('Failed to complete feature engineering: %
    s', e)
124         print(f"Error: {e}")
125
126
127 if __name__ == '__main__':
128     main()
```

3.5 Model Building Module

src/Model_Building_Rf.py

```python
import os
import numpy as np
import pandas as pd
import pickle
import logging
from sklearn.ensemble import RandomForestClassifier
import yaml

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Model_Building')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Model_Building.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s', params_path)
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise


def load_data(file_path: str) -> pd.DataFrame:
```

```
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        logger.debug('Data loaded from %s with shape %s',
                     file_path, df.shape)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except FileNotFoundError as e:
        logger.error('File not found: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s',
    e)
        raise


def train_model(X_train: np.ndarray,
                y_train: np.ndarray,
                params: dict) -> RandomForestClassifier:
    """Train the RandomForest model."""
    try:
        if X_train.shape[0] != y_train.shape[0]:
            raise ValueError(
                "Number of samples in X_train and y_train must
    match"
            )

        logger.debug('Initializing RandomForest with params: %s
    ',
                     params)
        clf = RandomForestClassifier(
            n_estimators=params['n_estimators'],
            random_state=params['random_state'],
            max_depth=params['max_depth']
        )

        logger.debug('Model training started with %d samples',
                     X_train.shape[0])
        clf.fit(X_train, y_train)
        logger.debug('Model training completed')

        return clf
    except ValueError as e:
        logger.error('ValueError during model training: %s', e)
        raise
    except Exception as e:
        logger.error('Error during model training: %s', e)
        raise


def save_model(model, file_path: str) -> None:
    """Save the trained model to a file."""
    try:
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```
106
107         with open(file_path, 'wb') as file:
108             pickle.dump(model, file)
109         logger.debug('Model saved to %s', file_path)
110     except FileNotFoundError as e:
111         logger.error('File path not found: %s', e)
112         raise
113     except Exception as e:
114         logger.error('Error while saving model: %s', e)
115         raise
116
117
118 def main():
119     try:
120         params = load_params('params.yaml')['model_building']
121
122         train_data = load_data('./data/processed/train_tfidf.
    csv')
123         X_train = train_data.iloc[:, :-1].values
124         y_train = train_data.iloc[:, -1].values
125
126         clf = train_model(X_train, y_train, params)
127
128         model_save_path = 'models/model_rf.pkl'
129         save_model(clf, model_save_path)
130
131     except Exception as e:
132         logger.error('Failed to complete model building: %s', e
    )
133         print(f"Error: {e}")
134
135
136 if __name__ == '__main__':
137     main()
```

4 MLflow Integration with DagShub

4.1 What is DagShub?

DagShub is a platform for data scientists and ML engineers that provides:

- **Hosted MLflow**: No need to run your own MLflow server
- **Git Integration**: Direct connection to GitHub repositories
- **DVC Support**: Native DVC integration
- **Team Collaboration**: Shared experiment tracking
- **Free Tier**: Generous free plan for small teams

4.2 Connecting to DagShub

4.2.1 Step 1: Create DagShub Account

1. Go to `https://dagshub.com`
2. Sign up with GitHub account
3. Create a new repository or connect existing one

4.2.2 Step 2: Obtain Tracking URI

After connecting your repository to DagShub:

```
# Your tracking URI will look like:
https://dagshub.com/username/repo-name.mlflow
```

4.3 Model Evaluation with MLflow

src/Model_Evaluation_Rf.py - Complete Implementation

```python
import os
import numpy as np
import pandas as pd
import pickle
import json
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, roc_auc_score)
import logging
import mlflow
import yaml
import dagshub

# Initialize DagShub connection
dagshub.init(
    repo_owner='Error-Makes-Clever',
    repo_name='MLOPS-Dagshub-DVC-Git-Project',
    mlflow=True
)

# Set MLflow tracking URI
```

```
mlflow.set_tracking_uri(
    "https://dagshub.com/Error-Makes-Clever/"
    "MLOPS-Dagshub-DVC-Git-Project.mlflow"
)

# Set experiment name
mlflow.set_experiment("MLOPS-Project-with-Random-Forest")

# Ensure the "logs" directory exists
log_dir = 'logs'
os.makedirs(log_dir, exist_ok=True)

# Logging configuration
logger = logging.getLogger('Model_Evaluation')
logger.setLevel('DEBUG')

console_handler = logging.StreamHandler()
console_handler.setLevel('DEBUG')

log_file_path = os.path.join(log_dir, 'Model_Evaluation.log')
file_handler = logging.FileHandler(log_file_path)
file_handler.setLevel('DEBUG')

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)


def load_params(params_path: str) -> dict:
    """Load parameters from a YAML file."""
    try:
        with open(params_path, 'r') as file:
            params = yaml.safe_load(file)
        logger.debug('Parameters retrieved from %s',
    params_path)
        return params
    except FileNotFoundError:
        logger.error('File not found: %s', params_path)
        raise
    except yaml.YAMLError as e:
        logger.error('YAML error: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error: %s', e)
        raise


def load_model(file_path: str):
    """Load the trained model from a file."""
    try:
        with open(file_path, 'rb') as file:
```

```
            model = pickle.load(file)
        logger.debug('Model loaded from %s', file_path)
        return model
    except FileNotFoundError:
        logger.error('File not found: %s', file_path)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading model: %s', e)
        raise


def load_data(file_path: str) -> pd.DataFrame:
    """Load data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        logger.debug('Data loaded from %s', file_path)
        return df
    except pd.errors.ParserError as e:
        logger.error('Failed to parse the CSV file: %s', e)
        raise
    except Exception as e:
        logger.error('Unexpected error while loading data: %s', e)
        raise


def evaluate_model(clf, X_test: np.ndarray, y_test: np.ndarray) -> dict:
    """Evaluate the model and return metrics."""
    try:
        y_pred = clf.predict(X_test)
        y_pred_proba = clf.predict_proba(X_test)[:, 1]

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        auc = roc_auc_score(y_test, y_pred_proba)

        metrics_dict = {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'auc': auc
        }
        logger.debug('Model evaluation metrics calculated')
        return metrics_dict
    except Exception as e:
        logger.error('Error during model evaluation: %s', e)
        raise


def save_metrics(metrics: dict, file_path: str) -> None:
    """Save the evaluation metrics to a JSON file."""
    try:
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```
130
131         with open(file_path, 'w') as file:
132             json.dump(metrics, file, indent=4)
133         logger.debug('Metrics saved to %s', file_path)
134     except Exception as e:
135         logger.error('Error while saving metrics: %s', e)
136         raise
137
138
139 def main():
140     try:
141         params = load_params('params.yaml')
142
143         clf = load_model('./models/model_rf.pkl')
144         test_data = load_data('./data/processed/test_tfidf.csv'
    )
145
146         X_test = test_data.iloc[:, :-1].values
147         y_test = test_data.iloc[:, -1].values
148
149         metrics = evaluate_model(clf, X_test, y_test)
150
151         # MLflow tracking
152         with mlflow.start_run():
153             # Log metrics
154             mlflow.log_metric('accuracy', metrics['accuracy'])
155             mlflow.log_metric('precision', metrics['precision'
    ])
156             mlflow.log_metric('recall', metrics['recall'])
157             mlflow.log_metric('auc', metrics['auc'])
158
159             # Log parameters
160             mlflow.log_param('n_estimators',
161                              params['model_building']['
    n_estimators'])
162             mlflow.log_param('random_state',
163                              params['model_building']['
    random_state'])
164             mlflow.log_param('max_depth',
165                              params['model_building']['max_depth'
    ])
166             mlflow.log_param('max_features',
167                              params['feature_engineering']['
    max_features'])
168             mlflow.log_param('test_size',
169                              params['data_ingestion']['test_size'
    ])
170
171         save_metrics(metrics, 'reports/metrics.json')
172     except Exception as e:
173         logger.error('Failed to complete model evaluation: %s',
     e)
174         print(f"Error: {e}")
175
176
177 if __name__ == '__main__':
```

```
178     main()
```

4.4 Understanding the MLflow Integration

4.4.1 Key Components

1. **dagshub.init()**: Establishes connection to DagShub

```
dagshub.init(
    repo_owner='username',
    repo_name='project-name',
    mlflow=True
)

```

2. **mlflow.set_tracking_uri()**: Points MLflow to DagShub

```
mlflow.set_tracking_uri(
    "https://dagshub.com/username/project.mlflow"
)

```

3. **mlflow.set_experiment()**: Groups related runs

```
mlflow.set_experiment("MLOPS-Project-with-Random-Forest")

```

4. **mlflow.start_run()**: Creates a new experiment run

```
with mlflow.start_run():
    mlflow.log_metric('accuracy', 0.95)
    mlflow.log_param('n_estimators', 100)

```

MLflow Concepts

Experiment: A collection of related runs (e.g., all RandomForest experiments)

Run: A single execution of your ML code with specific parameters

Metrics: Performance measurements (accuracy, precision, recall)

Parameters: Hyperparameters used in the run (n_estimators, max_depth)

Artifacts: Files generated during the run (models, plots, reports)

5 Configuration Files

5.1 Step 7: Create .gitignore

.gitignore

```
# Virtual environment
venv/
env/

# Data directories (DVC will track these)
data/
models/
reports/

# Logs
logs/

# Python cache
__pycache__/
*.pyc
*.pyo

# IDE
.vscode/
.idea/

# OS
.DS_Store
Thumbs.db
```

Important Note

Important: The `data/`, `models/`, and `reports/` directories are excluded from Git because DVC will handle versioning for these large files.

5.2 Step 8: Initialize DVC and Git

```
# Initialize Git
git init

# Initialize DVC
dvc init
```

What happens:

- Git creates `.git/` directory
- DVC creates `.dvc/` directory
- DVC modifies `.gitignore` to exclude cache

5.3 Step 9: Create DVC Pipeline

dvc.yaml

```yaml
stages:
  data_ingestion:
    cmd: python src/Data_Ingestion.py
    deps:
      - src/Data_Ingestion.py
    params:
      - data_ingestion.test_size
    outs:
      - data/raw

  data_preprocessing:
    cmd: python src/Data_Pre_Processing.py
    deps:
      - data/raw
      - src/Data_Pre_Processing.py
    outs:
      - data/interim

  feature_engineering:
    cmd: python src/Feature_Engineering.py
    deps:
      - data/interim
      - src/Feature_Engineering.py
    params:
      - feature_engineering.max_features
    outs:
      - data/processed

  model_building:
    cmd: python src/Model_Building_Rf.py
    deps:
      - data/processed
      - src/Model_Building_Rf.py
    params:
      - model_building.n_estimators
      - model_building.random_state
      - model_building.max_depth
    outs:
      - models/model_rf.pkl

  model_evaluation:
    cmd: python src/Model_Evaluation_Rf.py
    deps:
      - models/model_rf.pkl
      - src/Model_Evaluation_Rf.py
    metrics:
      - reports/metrics.json
```

5.4 Step 10: Create Parameters File

params.yaml

```yaml
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 22
  random_state: 2
  max_depth: 11
```

6 Initial Git Commit and DVC Setup

6.1 Step 11: Check Git Status

```
git status
```

Expected Output:

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .dvc/.gitignore
        new file:   .dvc/config
        new file:   .dvcignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        dvc.yaml
        params.yaml
        requirements.txt
        src/
```

6.2 Step 12: Check DVC Status

```
dvc status
```

Expected Output:

```
data_ingestion:
    changed deps:
        modified:           src\Data_Ingestion.py
        new:                params.yaml
    changed outs:
        deleted:            data\raw

data_preprocessing:
    changed deps:
        deleted:            data\raw
        modified:           src\Data_Pre_Processing.py
    changed outs:
        deleted:            data\interim

feature_engineering:
    changed deps:
        deleted:            data\interim
        modified:           src\Feature_Engineering.py
        new:                params.yaml
    changed outs:
        deleted:            data\processed
```

```
model_building:
    changed deps:
        deleted:            data\processed
        modified:           src\Model_Building_Rf.py
        new:                params.yaml
    changed outs:
        deleted:            models\model_rf.pkl

model_evaluation:
    changed deps:
        deleted:            models\model_rf.pkl
        modified:           src\Model_Evaluation_Rf.py
    changed outs:
        deleted:            reports\metrics.json
```

Important Note

This output is normal! DVC is detecting that the pipeline has never been run. All outputs are listed as "deleted" because they don't exist yet.

6.3 Step 13: Make Initial Git Commit

```
# Stage all files
git add .

# Commit
git commit -m "Initial commit: set up ML pipeline components"

# Push to remote (after creating GitHub repo)
git remote add origin https://github.com/username/project.git
git push -u origin main
```

6.4 Step 14: Configure DVC Remote Storage

6.4.1 Create Local Storage Directory

```
# Create S3 directory for local storage
mkdir S3
```

6.4.2 Add as DVC Remote

```
# Add local directory as DVC remote
dvc remote add -d dvc_origin S3

# Verify remote configuration
dvc remote list
```

Expected Output:

```
dvc_origin    S3    (default)
```

6.4.3 Check DVC Configuration

```
# View .dvc/config file
cat .dvc/config
```

Content:

```
[core]
    remote = dvc_origin

['remote "dvc_origin"']
    url = S3
```

6.4.4 Add S3 to .gitignore

```
# Add to .gitignore
echo "S3/" >> .gitignore
```

Warning

Important: The S3 directory should NOT be tracked by Git. It will contain all your data versions and can become very large.

7 Running the Pipeline and First Experiment

7.1 Step 15: Execute the Pipeline

```
dvc repro
```

7.2 Pipeline Execution Output

Complete Pipeline Output

```
Running stage 'data_ingestion':
> python src/Data_Ingestion.py
2025-12-20 18:28:06,435 - Data_Ingestion - DEBUG - Parameters
    retrieved from params.yaml
2025-12-20 18:28:07,002 - Data_Ingestion - DEBUG - Data loaded
    from https://raw.githubusercontent.com/...
2025-12-20 18:28:07,004 - Data_Ingestion - DEBUG - Data
    preprocessing completed
2025-12-20 18:28:07,034 - Data_Ingestion - DEBUG - Train and
    test data saved to ./data\raw
Updating lock file 'dvc.lock'

Running stage 'data_preprocessing':
> python src/Data_Pre_Processing.py
[nltk_data] Downloading package stopwords...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt...
[nltk_data]   Package punkt is already up-to-date!
2025-12-20 18:28:16,011 - Data_Pre_Processing - DEBUG - Data
    loaded properly
2025-12-20 18:28:16,012 - Data_Pre_Processing - DEBUG - Starting
    preprocessing for DataFrame
2025-12-20 18:28:16,015 - Data_Pre_Processing - DEBUG - Target
    column encoded
2025-12-20 18:28:16,023 - Data_Pre_Processing - DEBUG -
    Duplicates removed
2025-12-20 18:28:40,913 - Data_Pre_Processing - DEBUG - Text
    column transformed
Updating lock file 'dvc.lock'

Running stage 'feature_engineering':
> python src/Feature_Engineering.py
2025-12-20 18:28:56,433 - Feature_Engineering - DEBUG -
    Parameters retrieved from params.yaml
2025-12-20 18:28:56,452 - Feature_Engineering - DEBUG - Data
    loaded and NaNs filled
2025-12-20 18:28:56,607 - Feature_Engineering - DEBUG - TF-IDF
    applied and data transformed
Updating lock file 'dvc.lock'

Running stage 'model_building':
```

```
> python src/Model_Building_Rf.py
2025-12-20 18:29:04,072 - Model_Building - DEBUG - Parameters
    retrieved from params.yaml
2025-12-20 18:29:04,101 - Model_Building - DEBUG - Data loaded
    with shape (4152, 26)
2025-12-20 18:29:04,102 - Model_Building - DEBUG - Initializing
    RandomForest model
2025-12-20 18:29:04,103 - Model_Building - DEBUG - Model training
    started with 4152 samples
2025-12-20 18:29:04,211 - Model_Building - DEBUG - Model training
    completed
2025-12-20 18:29:04,214 - Model_Building - DEBUG - Model saved
    to models/model_rf.pkl
Updating lock file 'dvc.lock'

Running stage 'model_evaluation':
> python src/Model_Evaluation_Rf.py
Accessing as Error-Makes-Clever
Initialized MLflow to track repo
    "Error-Makes-Clever/MLOPS-Dagshub-DVC-Git-Project"
Repository Error-Makes-Clever/MLOPS-Dagshub-DVC-Git-Project
    initialized!
2025-12-20 18:29:20,460 - Model_Evaluation - DEBUG - Parameters
    retrieved from params.yaml
2025-12-20 18:29:20,685 - Model_Evaluation - DEBUG - Model loaded
2025-12-20 18:29:20,692 - Model_Evaluation - DEBUG - Data loaded
2025-12-20 18:29:20,714 - Model_Evaluation - DEBUG - Model
    evaluation metrics calculated
 View run handsome-asp-382 at:
    https://dagshub.com/.../runs/beaa07ff...
 View experiment at:
    https://dagshub.com/.../experiments/0
2025-12-20 18:29:25,787 - Model_Evaluation - DEBUG - Metrics
    saved to reports/metrics.json
Updating lock file 'dvc.lock'

To track the changes with git, run:
    git add dvc.lock

To enable auto staging, run:
    dvc config core.autostage true

Use `dvc push` to send your updates to remote storage.
```

7.3 Understanding the Output

1. **Stage Execution**: Each stage runs sequentially
2. **dvc.lock Updates**: After each stage, DVC updates the lock file
3. **MLflow Integration**: Model evaluation connects to DagShub

4. **Run Link**: DagShub provides a direct link to view the run

7.4 Step 16: Understanding DVC Cache

After running the pipeline, check your project structure:

```
project/
+-- .dvc/
|   +-- cache/          # Contains cached artifacts
|   +-- config
|   +-- .gitignore
+-- S3/                 # Remote storage (local directory)
+-- data/
|   +-- raw/
|   +-- interim/
|   +-- processed/
+-- models/
|   +-- model_rf.pkl
+-- reports/
|   +-- metrics.json
+-- logs/
|   +-- *.log files
+-- dvc.lock            # Generated lock file
```

What's in Each Directory?

.dvc/cache/:

- Stores all versions of tracked data locally
- Content-addressable storage (files named by hash)
- Acts as local backup

S3/:

- Remote storage location
- Mirrors .dvc/cache/ structure
- Enables team sharing

dvc.lock:

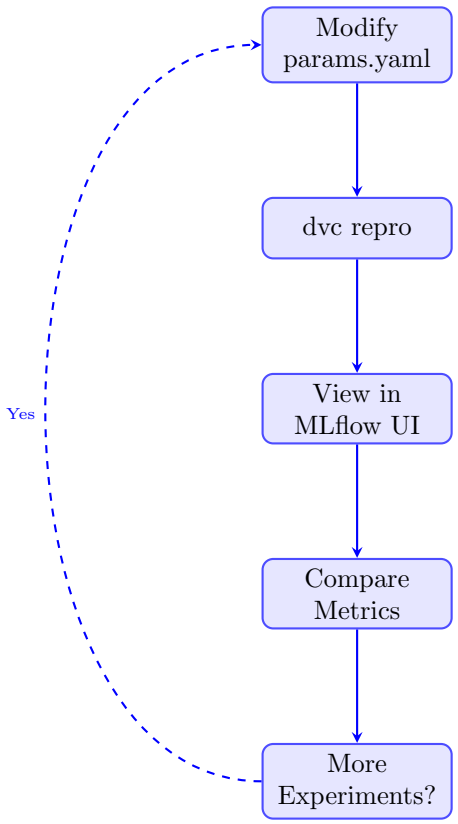
- Records exact file versions (MD5 hashes)
- Ensures reproducibility
- Tracked by Git

8 Running Multiple Experiments

8.1 Step 17: Experimentation Phase

Now we'll run multiple experiments with different hyperparameters to find the best model.

8.2 Experiment Workflow



8.3 Running Experiment Variations

8.3.1 Experiment 1: Initial Configuration

```yaml
# params.yaml - Run 1
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 22
  random_state: 2
  max_depth: 11
```

```
dvc repro
```

8.3.2 Experiment 2: Fewer Trees

```yaml
# params.yaml - Run 2
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 18    # Changed
  random_state: 2
  max_depth: 5        # Changed
```

```
dvc repro
```

8.3.3 Experiment 3: More Features

```yaml
# params.yaml - Run 3
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 50    # Changed

model_building:
  n_estimators: 22
  random_state: 2
  max_depth: 11
```

```
dvc repro
```

8.3.4 Experiment 4: Different Split

```yaml
# params.yaml - Run 4
data_ingestion:
  test_size: 0.20     # Changed

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 30    # Changed
  random_state: 2
  max_depth: 15       # Changed
```

```
dvc repro
```

8.3.5 Experiment 5: Balanced Configuration

```yaml
# params.yaml - Run 5
data_ingestion:
  test_size: 0.25

```

```yaml
feature_engineering:
  max_features: 40     # Changed

model_building:
  n_estimators: 25     # Changed
  random_state: 2
  max_depth: 12        # Changed
```

```
dvc repro
```

9 Step 18: Analyzing Experiments in MLflow

9.1 Accessing the MLflow UI

Navigate to your DagShub MLflow tracking URI:

```
https://dagshub.com/Error-Makes-Clever/
    MLOPS-Dagshub-DVC-Git-Project.mlflow
```

9.2 Experiment Results Summary

Run	AUC	Accuracy	Precision	Recall
Run 1 (able-ray-853)	0.9187	0.9378	0.8302	0.6947
Run 2 (handsome-asp-382)	0.8656	0.8919	0.8776	0.2756
Run 3 (placid-gnu-625)	0.8846	0.8922	0.8704	0.2765
Run 4 (wise-elk-465)	0.8994	0.9254	0.7933	0.6263
Run 5 (receptive-crow-130)	0.9178	0.9334	0.8414	0.6421

9.3 Detailed Run Analysis

9.3.1 Run 1: able-ray-853

Run 1 Metrics

Performance:

- AUC: 0.9187
- Accuracy: 0.9378
- Precision: 0.8302
- Recall: 0.6947

Parameters:

- n_estimators: 22
- max_depth: 11
- max_features: 35
- test_size: 0.25

Analysis: Best overall performance with highest AUC and accuracy.

9.3.2 Run 2: handsome-asp-382

Run 2 Metrics

Performance:

- AUC: 0.8656
- Accuracy: 0.8919
- Precision: 0.8776

- Recall: 0.2756

Parameters:

- n_estimators: 18
- max_depth: 5
- max_features: 35
- test_size: 0.25

Analysis: High precision but very low recall. Model is too conservative.

9.3.3 Run 5: receptive-crow-130

Run 5 Metrics

Performance:

- AUC: 0.9178
- Accuracy: 0.9334
- Precision: 0.8414
- Recall: 0.6421

Parameters:

- n_estimators: 25
- max_depth: 12
- max_features: 40
- test_size: 0.25

Analysis: Second-best performance, close to Run 1.

10 Step 19: Deciding What "Best" Means

10.1 Defining Success Criteria

Before committing anything, define one primary metric based on your use case:

Metric Priority	Use Case
Recall	Spam detection - catch all spam (few false negatives)
Precision	Medical diagnosis - avoid false positives
Accuracy	General classification with balanced classes
AUC	Ranking, threshold tuning, imbalanced data
F1-Score	Balance between precision and recall

10.2 Analysis for Spam Classification

For spam detection, we typically want:

1. High AUC (good separation between classes)
2. High accuracy (overall correctness)
3. Balanced precision and recall

10.3 Ranking the Runs

Run	AUC	Accuracy	Recommendation
Run 1	0.9187	0.9378	**Winner ✓**
Run 5	0.9178	0.9334	Runner-up
Run 4	0.8994	0.9254	Good
Run 3	0.8846	0.8922	Acceptable
Run 2	0.8656	0.8919	Poor recall

Important Note

Winner: Run 1 (able-ray-853)

Justification:

- Highest AUC (0.9187) - best class separation
- Highest accuracy (0.9378) - best overall performance
- Balanced precision (0.8302) and recall (0.6947)
- Reliable and reproducible

11 Step 20: Re-creating the Winning Run

11.1 Why Re-create?

Warning

Critical MLOps Principle:

Earlier runs were **exploratory**. Now you are making a **decision**.

Why re-create:

- Makes the decision explicit
- Creates a clean Git commit representing the approved model
- Enables future rollback to this exact state
- Documents which experiment was promoted to production

Don't skip this step! It's the difference between "I ran some experiments" and "This is our production model."

11.2 Obtaining Winning Parameters

Go to MLflow UI → Click on Run 1 (able-ray-853) → View Parameters:

Parameter	Value
test_size	0.25
n_estimators	22
random_state	2
max_depth	11
max_features	35

11.3 Step 21: Update params.yaml

params.yaml - Winning Configuration

```yaml
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 22
  random_state: 2
  max_depth: 11
```

11.4 Step 22: Run Pipeline with Winning Parameters

```
dvc repro
```

Expected Output:

```
Stage 'data_ingestion' didn't change, skipping
Stage 'data_preprocessing' didn't change, skipping
Stage 'feature_engineering' didn't change, skipping
Stage 'model_building' is cached - skipping run,
    checking out outputs
Updating lock file 'dvc.lock'

Stage 'model_evaluation' is cached - skipping run,
    checking out outputs
Updating lock file 'dvc.lock'

To track the changes with git, run:
    git add dvc.lock

To enable auto staging, run:
    dvc config core.autostage true

Use `dvc push` to send your updates to remote storage.
```

Why No New MLflow Run?

DVC detected that:

- Code did not change
- Parameters did not change
- Data did not change

Result: DVC used cached outputs. No new execution = no new MLflow run.

This is correct behavior! We're verifying the winning configuration, not running a new experiment.

11.5 Step 23: Verify Metrics (Sanity Check)

```
# Check metrics.json
cat reports/metrics.json
```

Expected Content:

```
{
    "accuracy": 0.9378200438917337,
    "precision": 0.8301886792452831,
    "recall": 0.6947368421052632,
    "auc": 0.9186759379331932
}
```

Important Note

Verification:

- Metrics match (or are very close to) Run 1
- No accidental changes

- Configuration is reproducible

If metrics differ wildly → check random seed and data consistency.

12 Step 24: The Critical Commit

12.1 Why This Commit Matters

The Freezing Commit

This is the **most important commit** in your MLOps workflow.

What this commit represents:

- ✓ This model is **approved**
- ✓ This state is **reproducible**
- ✓ This is **rollback-safe**
- ✓ This is **deployable**

What gets committed:

- Code (src/)
- Pipeline definition (dvc.yaml)
- Parameters (params.yaml)
- Lock file (dvc.lock) - contains exact data/model versions

What does NOT get committed:

- Data files (tracked by DVC)
- Model files (tracked by DVC)
- Logs

12.2 Check Git Status

```
git status
```

Expected Output:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes)
        modified:   .dvc/config
        modified:   dvc.yaml
        modified:   requirements.txt
        modified:   src/Model_Evaluation_Rf.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dvc.lock
        logs/
```

```
no changes added to commit
```

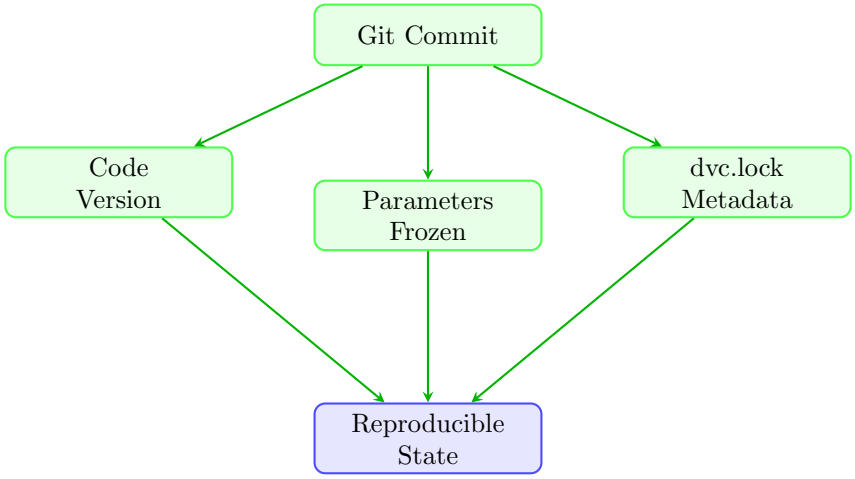
12.3 Stage and Commit

```
# Stage all changes
git add .

# Commit with descriptive message
git commit -m "Freeze best RF model (AUC=0.918, accuracy=0.937)"

# Push to remote
git push origin main
```

12.4 What This Commit Means



13 Step 25: Push Data and Model to DVC Remote

13.1 Why Push to DVC Remote?

Warning

Git vs DVC Storage:

What Git has:

- Code
- dvc.lock (metadata with hashes)
- Parameters

What Git does NOT have:

- Actual data files
- Actual model files

Problem: Without `dvc push`, teammates cannot reproduce your work!

13.2 Execute DVC Push

```
dvc push
```

Expected Output:

```
Collecting                              |8.00 [00:00, 2.50entry/s]
Pushing                                 |8.00 [00:02, 3.20file/s]
8 files pushed
```

13.3 What Gets Pushed

Local Location	DVC Remote (S3)
data/raw/	Uploaded to S3/
data/interim/	Uploaded to S3/
data/processed/	Uploaded to S3/
models/model_rf.pkl	Uploaded to S3/
reports/metrics.json	Uploaded to S3/

13.4 Verification

```
# Check S3 directory
ls S3/

# Should see hash-named directories
```

Complete State Now

GitHub has:

- Code (src/)

- Pipeline (dvc.yaml)
- Parameters (params.yaml)
- Metadata (dvc.lock)

DVC Remote (S3) has:

- All data versions
- All model versions
- All reports

Result: Complete reproducibility ✓

14 Step 26: Linking Git Commits to MLflow Runs

14.1 Why Tag MLflow Runs?

The Problem

Situation:

- You have 5 MLflow runs
- You committed Run 1 as the winner
- Weeks later: "Which MLflow run is our production model?"

Solution: Tag the MLflow run with the Git commit hash!

Benefit:

- Clear link between MLflow experiment and Git commit
- Easy to identify production model
- Audit trail for compliance

14.2 Create Tagging Script

Adding_tag_to_runs.py

```python
import subprocess
import mlflow
from mlflow.tracking import MlflowClient
import dagshub

# Authenticate with DagShub
dagshub.init(
    repo_owner="Error-Makes-Clever",
    repo_name="MLOPS-Dagshub-DVC-Git-Project",
    mlflow=True
)

# Set tracking URI
mlflow.set_tracking_uri(
    "https://dagshub.com/Error-Makes-Clever/"
    "MLOPS-Dagshub-DVC-Git-Project.mlflow"
)

# Initialize MLflow client
client = MlflowClient()

# Run ID from MLflow UI (Run 1)
run_id = "eaa2537460244123bc672bee9c22da05"

# Get current Git commit hash
commit = subprocess.check_output(
    ["git", "rev-parse", "HEAD"]
).decode().strip()

```

```
# Add tag to MLflow run
client.set_tag(run_id, "git_commit", commit)

# Optional: Add more tags
client.set_tag(run_id, "status", "production")
client.set_tag(run_id, "approved_by", "data_science_team")
client.set_tag(run_id, "deployment_date", "2025-12-20")

print(f"Tags successfully added to run {run_id}")
print(f"   Git commit: {commit}")
```

14.3 Understanding the Script

14.3.1 Key Components

1. **Get Git Commit Hash**:

```
commit = subprocess.check_output(
    ["git", "rev-parse", "HEAD"]
).decode().strip()

```

2. **Set Tag in MLflow**:

```
client.set_tag(run_id, "git_commit", commit)

```

3. **Additional Metadata Tags**:

```
client.set_tag(run_id, "status", "production")
client.set_tag(run_id, "approved_by", "team_name")

```

14.4 Finding the Run ID

Option 1: From MLflow UI:

1. Go to DagShub MLflow interface
2. Click on Run 1 (able-ray-853)
3. Copy the Run ID from URL or run details

Option 2: From Console Output:

```
View run at: .../runs/eaa2537460244123bc672bee9c22da05
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                     This is the Run ID
```

14.5 Step 27: Execute Tagging Script

```
python Adding_tag_to_runs.py
```

Expected Output:

```
Tags successfully added to run eaa2537460244123bc672bee9c22da05
   Git commit: a3f8e2c9d1b4e5f6g7h8i9j0k1l2m3n4
```

14.6 Step 28: Verify Tags in MLflow UI

1. Go to MLflow UI
2. Click on Run 1 (able-ray-853)
3. Scroll to "Tags" section
4. Verify tags appear:

Tag Key	Tag Value
git_commit	a3f8e2c9d1b4e5f6...
status	production
approved_by	data_science_team
deployment_date	2025-12-20

Important Note

Best Practice: Always tag production runs with:

- Git commit hash
- Deployment status
- Approval information
- Date/time of promotion

This creates a complete audit trail for compliance and debugging.

15 Step 29: Multi-Model Experimentation

15.1 Scenario: Team with Multiple Models

Real-World Team Setup

Scenario:

- Data Scientist A: Working on RandomForest
- Data Scientist B: Working on XGBoost
- Data Scientist C: Working on Neural Networks

Goal: Everyone shares the same infrastructure but tracks separate experiments.

15.2 Using Different Experiment Names

15.2.1 RandomForest Experiments

```
# src/Model_Evaluation_Rf.py
mlflow.set_experiment("MLOPS-Project-with-Random-Forest")
```

15.2.2 XGBoost Experiments

```
# src/Model_Evaluation_Xgb.py
mlflow.set_experiment("MLOPS-Project-with-XGBoost")
```

15.2.3 Neural Network Experiments

```
# src/Model_Evaluation_NN.py
mlflow.set_experiment("MLOPS-Project-with-Neural-Networks")
```

15.3 Benefits of Separate Experiments

1. **Organization**: Experiments grouped by model type
2. **Clear Ownership**: Each team member manages their experiment
3. **Easy Comparison**: Compare within model family first
4. **Scalability**: Add new model types without conflicts

15.4 Viewing All Experiments

In the DagShub MLflow UI:

Experiment	Runs	Owner
MLOPS-Project-with-Random-Forest	5	Data Scientist A
MLOPS-Project-with-XGBoost	8	Data Scientist B
MLOPS-Project-with-Neural-Networks	12	Data Scientist C

Important Note

Key Advantage:
Everyone in your team can:

- See all experiments
- Compare across models
- Learn from each other's work
- Avoid duplicate efforts

This is impossible with local-only experiment tracking!

16 Reproducing the Project on a New Machine

16.1 The Reproducibility Challenge

Warning

Common Scenario:

A new team member joins or you need to reproduce the model on a production server.

Requirements:

- Exact code version
- Exact data version
- Exact model version
- Same environment

Solution: Git + DVC + proper setup

16.2 Important Note About Local DVC Remote

Critical: Local S3 Directory Limitation

In this project, the DVC remote is configured as a **local directory**:

```
S3/  (located at: S:\MLOPS-Dagshub-DVC-Git-Project\S3)
```

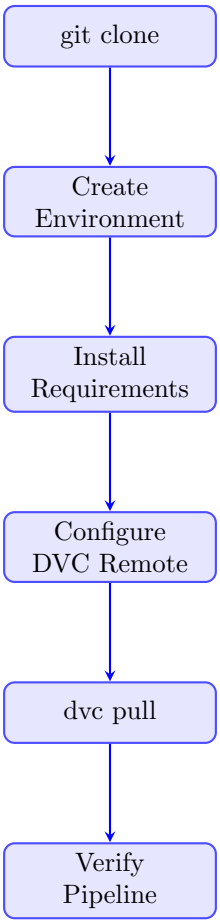
Problem: This is a local path, NOT accessible from other machines!

Options for Team Sharing:

1. Place S3 folder on **shared network drive**
2. Use **NAS / file server**
3. Map the **same path on all machines** (e.g., Z:\Shared\S3)
4. **Replace with real cloud remote** (AWS S3, GCS, Azure Blob)

Important: If S3 directory is NOT accessible, `dvc pull` will fail!

16.3 Reproduction Workflow



16.4 Step-by-Step Reproduction

16.4.1 Step 1: Clone Repository

```
# Clone Git repository
git clone https://github.com/Error-Makes-Clever/
    MLOPS-Dagshub-DVC-Git-Project.git

cd MLOPS-Dagshub-DVC-Git-Project
```

What this does:

- Downloads all code files
- Downloads dvc.yaml (pipeline definition)
- Downloads dvc.lock (file version metadata)
- Downloads params.yaml (hyperparameters)
- Does NOT download data/models (DVC handles this)

16.4.2 Step 2: Create and Activate Environment

```
# Create virtual environment
python -m venv venv

# Activate (Windows)
venv\Scripts\activate

# Activate (Linux/Mac)
source venv/bin/activate
```

Verify activation:

```
# Check Python path (should point to venv)
which python    # Linux/Mac
where python    # Windows
```

16.4.3 Step 3: Install Requirements

```
pip install -r requirements.txt
```

Expected packages:

- dvc
- numpy, pandas
- matplotlib, wordcloud
- nltk
- scikit-learn
- xgboost
- pyyaml
- mlflow
- dagshub

Verify installation:

```
# Check installed packages
pip list

# Verify DVC
dvc version

# Verify MLflow
mlflow --version
```

16.4.4 Step 4: Configure DVC Remote Access

```
# Check current remote configuration
dvc remote list
```

Expected Output:

```
dvc_origin    S3
```

Important Note

Understanding the Remote Path:
The remote path in the repository is relative (`S3`), which means it expects the S3 folder to be in the same parent directory as the project.

Original setup:

```
S:\MLOPS-Dagshub-DVC-Git-Project\
    +-- S3/                    (DVC remote)
    +-- project/               (Git repository)
```

For reproduction: You need to ensure the S3 directory is accessible at the same relative location OR update the remote URL.

If path needs updating:

Update DVC Remote Path

```
# Option 1: Use shared network drive (Windows)
dvc remote modify dvc_origin url Z:\Shared\MLOPS\S3

# Option 2: Use UNC path (Windows)
dvc remote modify dvc_origin url \\server\share\S3

# Option 3: Use mounted path (Linux/Mac)
dvc remote modify dvc_origin url /mnt/shared/S3

# Option 4: Use absolute local path
dvc remote modify dvc_origin url /path/to/shared/S3

# Verify updated configuration
dvc remote list -v
```

Check .dvc/config after modification:

```
cat .dvc/config
```

Expected content:

```
[core]
    remote = dvc_origin

['remote "dvc_origin"']
    url = /path/to/shared/S3
```

16.4.5 Step 5: Pull Data and Models

```
dvc pull
```

Expected Output:

```
Collecting                          |8.00 [00:00, 10.0entry/s]
Fetching
Building workspace index            |8.00 [00:00, 100entry/s]
Comparing indexes                   |8.00 [00:00, 200entry/s]
Applying changes                    |8.00 [00:01, 5.00file/s]
8 files fetched
```

What gets downloaded:

- `data/raw/train.csv`
- `data/raw/test.csv`
- `data/interim/train_processed.csv`
- `data/interim/test_processed.csv`
- `data/processed/train_tfidf.csv`
- `data/processed/test_tfidf.csv`
- `models/model_rf.pkl`
- `reports/metrics.json`

How DVC Pull Works

Process:

1. DVC reads `dvc.lock` to get file hashes
2. Checks if files exist in local cache (`.dvc/cache/`)
3. If missing, downloads from remote storage (S3)
4. Copies files from cache to workspace
5. Verifies checksums match

Result: Exact same data/models as the original machine!

16.4.6 Step 6: Verify Pipeline Status

```
dvc status
```

Expected Output:

```
Data and pipelines are up to date.
```

If you see changes:

```
data_ingestion:
    changed deps:
        modified:  src/Data_Ingestion.py
```

Possible causes:

- Line ending differences (Windows vs Linux)
- Incomplete pull
- Local modifications

Solution:

```
# Discard local changes
git checkout .

# Re-pull data
dvc pull

# Check status again
dvc status
```

16.4.7 Step 7 (Optional): Re-run Pipeline

```
dvc repro
```

Expected Result:

```
Stage 'data_ingestion' didn't change, skipping
Stage 'data_preprocessing' didn't change, skipping
Stage 'feature_engineering' didn't change, skipping
Stage 'model_building' is cached - skipping run
Stage 'model_evaluation' is cached - skipping run

Data and pipelines are up to date.
```

What this means:

- All stages are cached (no re-execution)
- No new MLflow runs created
- Guaranteed reproducibility ✓
- Pipeline is working correctly ✓

16.5 Verification Checklist

Item	Status	Command
Code matches	✓	`git log --oneline`
Data matches	✓	`dvc status`
Model exists	✓	`ls models/`
Metrics match	✓	`cat reports/metrics.json`
Environment ready	✓	`pip list`
Pipeline works	✓	`dvc repro`

16.6 Manual Verification

16.6.1 Verify Metrics

```
cat reports/metrics.json
```

Expected content:

```
{
    "accuracy": 0.9378200438917337,
    "precision": 0.8301886792452831,
    "recall": 0.6947368421052632,
    "auc": 0.9186759379331932
}
```

16.6.2 Verify Parameters

```
cat params.yaml
```

Expected content:

```
data_ingestion:
  test_size: 0.25

feature_engineering:
  max_features: 35

model_building:
  n_estimators: 22
  random_state: 2
  max_depth: 11
```

16.6.3 Test Model Loading

test_model.py

```python
import pickle
import pandas as pd

# Load model
with open('models/model_rf.pkl', 'rb') as f:
    model = pickle.load(f)

print(f"Model type: {type(model)}")
print(f"Number of estimators: {model.n_estimators}")
print(f"Max depth: {model.max_depth}")

# Load test data
test_data = pd.read_csv('data/processed/test_tfidf.csv')
X_test = test_data.iloc[:, :-1].values
y_test = test_data.iloc[:, -1].values

# Make predictions
predictions = model.predict(X_test)
print(f"Predictions shape: {predictions.shape}")
```

```python
print(f"First 10 predictions: {predictions[:10]}")

print("\nModel loaded and working correctly!")
```

```
python test_model.py
```

Expected output:

```
Model type: <class 'sklearn.ensemble._forest.RandomForestClassifier'>
Number of estimators: 22
Max depth: 11
Predictions shape: (1384,)
First 10 predictions: [0 0 0 1 0 0 0 0 0 0]

Model loaded and working correctly!
```

16.7 Why This Matters in MLOps

Reproducibility Benefits

Without dvc pull:

- Data is missing
- Models cannot be loaded
- Results cannot be verified
- Team collaboration breaks
- Production deployment fails

With dvc pull:

- Exact data restored
- Approved model retrieved
- Everyone works on same version
- Perfect reproducibility
- Confident deployment

16.8 Troubleshooting Reproduction Issues

16.8.1 Issue 1: "Remote not found"

Error:

```
ERROR: failed to pull data from the cloud -
    [Errno 2] No such file or directory: 'S3'
```

Solution:

1. Confirm S3 directory location with team

2. Update remote URL:

```
dvc remote modify dvc_origin url <correct-path>
```

3. Verify access permissions

4. Try pulling again

16.8.2 Issue 2: "Permission denied"

Error:

```
ERROR: failed to pull data - [Errno 13] Permission denied
```

Solution:

- Check read permissions on shared directory
- On Linux/Mac: `ls -la /path/to/S3`
- On Windows: Check folder properties → Security tab
- Contact IT for network drive access

16.8.3 Issue 3: "Checksum mismatch"

Error:

```
ERROR: checksum mismatch for 'data/raw/train.csv'
```

Solution:

```
# Remove corrupted cache
rm -rf .dvc/cache

# Re-pull data
dvc pull
```

16.8.4 Issue 4: "Python package conflicts"

Error:

```
ImportError: cannot import name 'X' from 'Y'
```

Solution:

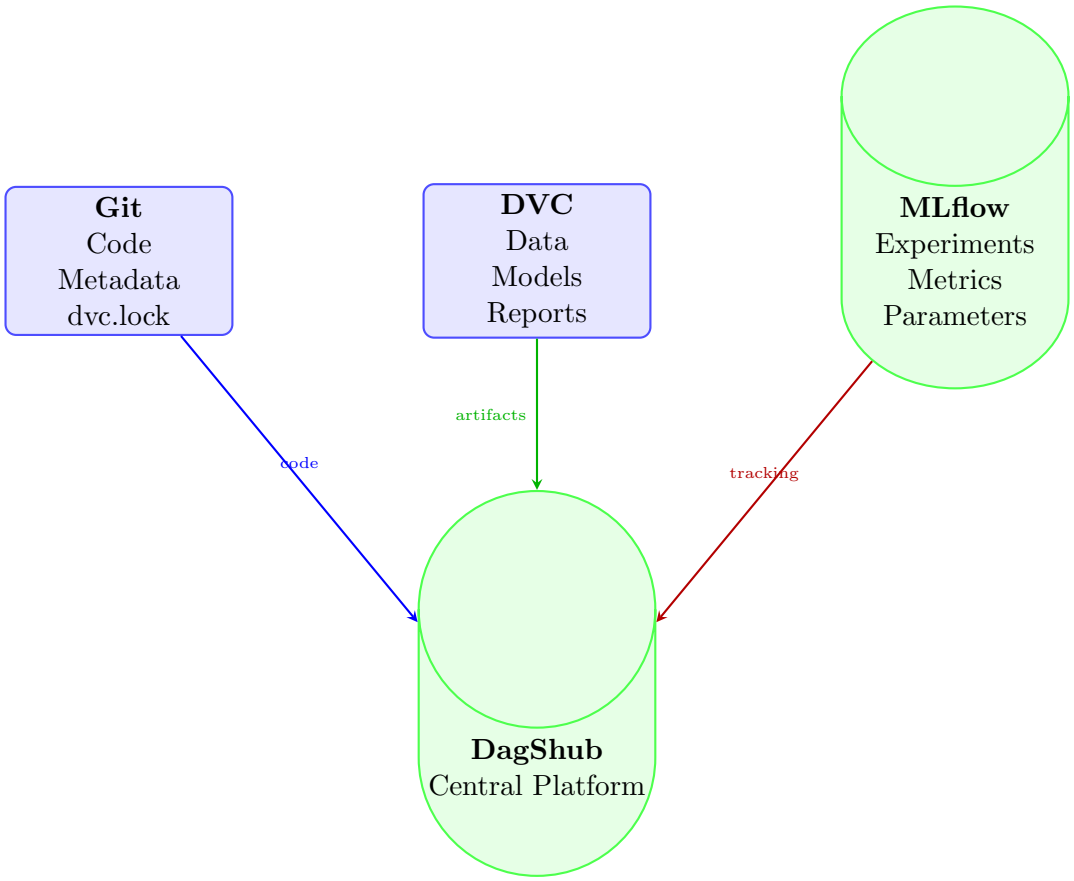
```
# Recreate environment
deactivate
rm -rf venv
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate

# Reinstall exact versions
pip install -r requirements.txt

# Verify
pip list
```

17 Complete Mental Model

17.1 The Three-System Architecture



17.2 Understanding Each Component

17.2.1 Git: Code and Metadata

Git Responsibilities

What Git tracks:

- Source code (src/)
- Pipeline definition (dvc.yaml)
- Parameters (params.yaml)
- Lock file (dvc.lock) - contains hashes
- Configuration files
- Documentation (README.md)

What Git does NOT track:

- Data files (too large)
- Model files (binary)

- Logs
- Cache directories

Key Commands:

- `git add .` - Stage changes
- `git commit -m "message"` - Save snapshot
- `git push` - Upload to GitHub
- `git checkout <hash>` - Switch versions

17.2.2 DVC: Data and Models

DVC Responsibilities

What DVC tracks:

- All data versions (raw, processed)
- All model versions
- Reports and metrics files
- Pipeline outputs

How DVC works:

- Stores actual files in remote storage (S3)
- Stores metadata (hashes) in Git (dvc.lock)
- Local cache (.dvc/cache/) for speed
- Content-addressable storage

Key Commands:

- `dvc add <file>` - Track file
- `dvc push` - Upload to remote
- `dvc pull` - Download from remote
- `dvc repro` - Run pipeline

17.2.3 MLflow: Experiment Tracking

MLflow Responsibilities

What MLflow tracks:

- All experiment runs
- Metrics (accuracy, precision, etc.)

- Parameters (hyperparameters)
- Artifacts (optional)
- Tags and metadata

Benefits:

- Visual comparison of experiments
- Centralized tracking (team-wide)
- Experiment history preserved
- Easy metric comparison

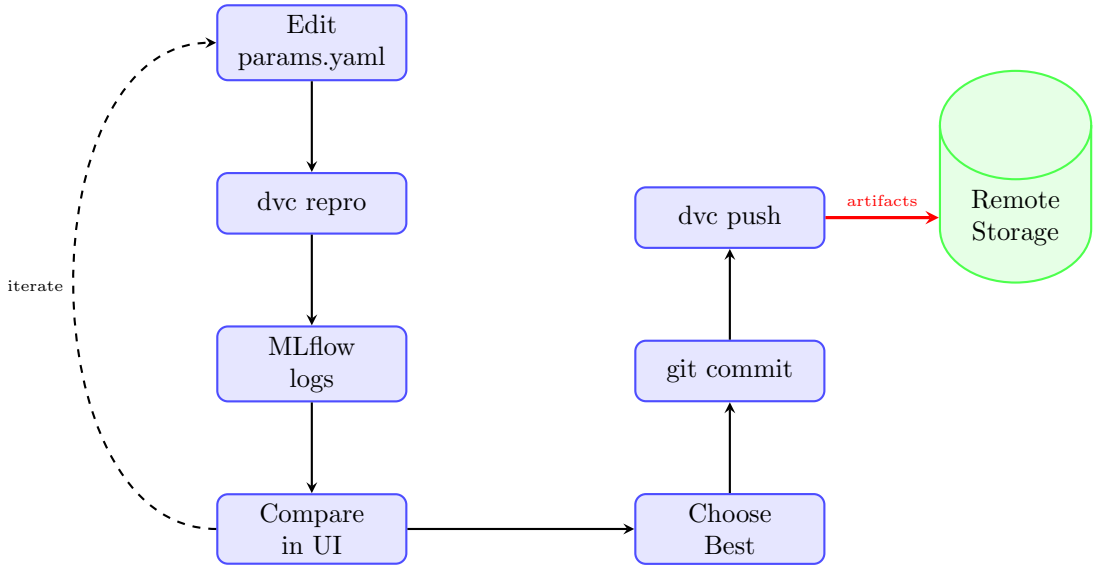
Key Functions:

- `mlflow.set_experiment()` - Group runs
- `mlflow.start_run()` - Begin tracking
- `mlflow.log_metric()` - Log metric
- `mlflow.log_param()` - Log parameter

17.3 Data Flow Summary

Command	What Happens
`git commit`	Saves code, params.yaml, dvc.lock to Git
`git push`	Uploads Git commits to GitHub
`dvc push`	Uploads data/models to DVC remote (S3)
`dvc pull`	Downloads data/models from DVC remote
`dvc repro`	Runs pipeline, updates dvc.lock
`mlflow.log_*`	Logs metrics/params to MLflow (DagShub)
`git checkout`	Restores code to specific commit

17.4 Complete Workflow Visualization



17.5 Key Concepts Review

Concept	Meaning
Experiment	Collection of related runs (e.g., all RandomForest tests)
Run	Single execution with specific parameters
dvc.lock	File with exact versions (hashes) of all data/models
params.yaml	Configuration file with hyperparameters
DVC Remote	Storage location for data/models (S3, GCS, local)
Freezing	Committing approved model configuration to Git
Tagging	Linking MLflow runs to Git commits
Reproducibility	Ability to recreate exact results on any machine

18 MLOps Best Practices

18.1 Experiment Management

18.1.1 Naming Conventions

Good Experiment Names

✓ "MLOPS-Project-with-Random-Forest"
✓ "MLOPS-Project-with-XGBoost"
✓ "MLOPS-Project-with-BERT-Embeddings"
✓ "Spam-Detection-RandomForest-v2"

× "experiment1"
× "test"
× "my_experiment"
× "asdf"

18.1.2 Commit Message Best Practices

Good Commit Messages

✓ Freeze best RF model (AUC=0.918, accuracy=0.937)
✓ Experiment: increase max_depth to 15, acc +0.02
✓ Fix: correct data leakage in preprocessing
✓ Add: XGBoost baseline (AUC=0.901)
✓ Refactor: modularize feature engineering

× Update
× Fix bug
× Changes
× WIP
× temp

18.1.3 Parameter Documentation

```yaml
# params.yaml with comments
data_ingestion:
  test_size: 0.25      # Standard 75/25 split

feature_engineering:
  max_features: 35     # Optimal from grid search

model_building:
  n_estimators: 22     # Best balance: speed vs accuracy
  random_state: 2      # For reproducibility
  max_depth: 11        # Prevents overfitting
```

18.2 DVC Remote Management

18.2.1 Production Setup Recommendations

Warning

For Production: Use Cloud Storage

Why local directories don't scale:

- Not accessible across team
- No disaster recovery
- Access control issues
- Cannot use in CI/CD pipelines
- No versioning/backup

Recommended Solutions:

- AWS S3 (most popular, cost-effective)
- Google Cloud Storage (GCS)
- Azure Blob Storage
- MinIO (self-hosted S3-compatible)

18.2.2 Migrating to AWS S3

AWS S3 Setup

```
# Install AWS support
pip install dvc[s3] awscli

# Configure AWS credentials
aws configure
# Enter: Access Key ID
# Enter: Secret Access Key
# Enter: Region (e.g., us-east-1)
# Enter: Output format (json)

# Remove local remote
dvc remote remove dvc_origin

# Add S3 remote
dvc remote add -d s3remote s3://my-bucket/dvc-storage

# Verify configuration
dvc remote list

# Push data to S3
dvc push

# Commit updated config
git add .dvc/config
git commit -m "Migrate to AWS S3 remote storage"
```

```
git push
```

18.3 MLflow Best Practices

18.3.1 What to Log

Always Log	Consider Logging
Primary metrics (accuracy, AUC)	Training time
All hyperparameters	Memory usage
Data split ratios	Number of features
Random seeds	Data version/hash
Model type	Hardware info (GPU/CPU)
Feature names	Dataset size

18.3.2 Comprehensive Tagging Strategy

Complete Tagging Example

```python
from mlflow.tracking import MlflowClient
import subprocess

client = MlflowClient()
run_id = "your-run-id"

# Tag with Git info
commit_hash = subprocess.check_output(
    ["git", "rev-parse", "HEAD"]
).decode().strip()
branch = subprocess.check_output(
    ["git", "rev-parse", "--abbrev-ref", "HEAD"]
).decode().strip()

client.set_tag(run_id, "git_commit", commit_hash)
client.set_tag(run_id, "git_branch", branch)

# Tag with deployment info
client.set_tag(run_id, "status", "production")
client.set_tag(run_id, "environment", "production")

# Tag with approval metadata
client.set_tag(run_id, "approved_by", "john_doe")
client.set_tag(run_id, "approval_date", "2025-12-20")
client.set_tag(run_id, "model_version", "v2.1.0")

# Tag with business context
client.set_tag(run_id, "use_case", "spam_detection")
client.set_tag(run_id, "department", "email_security")
client.set_tag(run_id, "priority", "high")

print(f"All tags added to run {run_id}")
```

18.4 Workflow Best Practices

18.4.1 Before Starting Experiments

1. **Define success criteria**:
   - What metric matters most?
   - What's acceptable performance?
   - What's the baseline?
2. **Set baseline**:
   - Run simplest model first
   - Document baseline performance
   - Use as comparison point
3. **Document assumptions**:
   - Data assumptions
   - Business constraints
   - Resource limitations
4. **Plan experiments**:
   - Create experiment matrix
   - Prioritize high-impact experiments
   - Estimate time/resources

18.4.2 During Experimentation

1. **Change one variable at a time**:
   - Isolate effects
   - Understand what works
   - Build intuition
2. **Log everything**:
   - All parameters
   - All metrics
   - Observations/notes
   - Unexpected results
3. **Keep detailed notes**:
   - Why you tried something
   - What you observed
   - Hypotheses for next steps
4. **Compare systematically**:
   - Use MLflow UI
   - Create comparison tables
   - Visualize trends

18.4.3 After Choosing Best Model

1. **Verify reproducibility**:
   - Run pipeline again
   - Confirm metrics match
   - Check random seed
2. **Document decision**:
   - Why this model?
   - What alternatives considered?
   - Trade-offs made
3. **Freeze configuration**:
   - Git commit with clear message
   - Include metrics in message
   - Tag as milestone
4. **Push artifacts**:
   - `dvc push` to remote
   - Verify upload successful
   - Test pull on different machine
5. **Tag MLflow run**:
   - Link to Git commit
   - Add production status
   - Include approval info
6. **Update documentation**:
   - README with latest model info
   - Wiki with experiment summary
   - Deployment guide

18.5 Team Collaboration Guidelines

18.5.1 Communication Best Practices

- **Share experiment plans**: Before starting work
- **Review each other's runs**: Learn together
- **Discuss surprises**: Unexpected results
- **Document learnings**: Wiki or shared docs
- **Regular sync meetings**: Weekly experiment reviews
- **Clear ownership**: Who owns which experiment
- **Celebrate wins**: Share successful experiments

18.5.2 Avoiding Conflicts

- **Use separate experiments**: One per person/model type
- **Coordinate on shared data**: Don't modify raw data
- **Review before freezing**: Get team approval
- **Clear ownership**: Who decides what to freeze
- **Communicate changes**: Notify team of updates
- **Use branches**: Feature branches for big changes

18.6 Code Quality Best Practices

18.6.1 Logging Standards

```
import logging

# Configure logger
logger = logging.getLogger(__name__)
logger.setLevel('DEBUG')

# Good logging practice
logger.debug('Loading data from %s', file_path)
logger.info('Training started with %d samples', n_samples)
logger.warning('Missing %d features, using defaults', n_missing)
logger.error('Failed to load model: %s', error_msg)

# Bad logging practice
logger.info('Loading data')  # Not informative
print('Training started')    # Should use logger
```

18.6.2 Error Handling

```
def load_data(file_path: str) -> pd.DataFrame:
    """Load data with comprehensive error handling."""
    try:
        # Attempt operation
        df = pd.read_csv(file_path)
        logger.info(f'Loaded {len(df)} rows from {file_path}')
        return df

    except FileNotFoundError:
        logger.error(f'File not found: {file_path}')
        raise

    except pd.errors.ParserError as e:
        logger.error(f'Failed to parse CSV: {e}')
        raise

    except Exception as e:
        logger.error(f'Unexpected error: {e}')
        raise
```

18.6.3 Type Hints and Documentation

```python
from typing import Tuple
import numpy as np

def train_model(
    X_train: np.ndarray,
    y_train: np.ndarray,
    params: dict
) -> RandomForestClassifier:
    """
    Train a RandomForest classifier.

    Args:
        X_train: Training features, shape (n_samples, n_features)
        y_train: Training labels, shape (n_samples,)
        params: Dictionary containing:
            - n_estimators: Number of trees
            - max_depth: Maximum tree depth
            - random_state: Random seed

    Returns:
        Trained RandomForestClassifier instance

    Raises:
        ValueError: If X_train and y_train shapes don't match

    Example:
        >>> params = {'n_estimators': 100, 'max_depth': 10}
        >>> model = train_model(X_train, y_train, params)
    """
    # Implementation
    pass
```

19 Comprehensive Troubleshooting Guide

19.1 DVC Issues

19.1.1 Issue 1: Failed to Push Data

Error Message:

```
ERROR: failed to push data to the cloud -
    [Errno 2] No such file or directory: 'S3'
```

Causes:

1. Remote storage path doesn't exist
2. Incorrect remote URL configuration
3. Permission issues
4. Network connectivity problems

Solutions:

```
# Step 1: Verify remote configuration
dvc remote list -v

# Step 2: Check if remote path exists
# For local directory:
ls /path/to/S3

# For AWS S3:
aws s3 ls s3://bucket-name/

# Step 3: Update remote URL if needed
dvc remote modify dvc_origin url <correct-path>

# Step 4: Verify permissions
# Check read/write access to remote location

# Step 5: Try verbose push
dvc push -v
```

19.1.2 Issue 2: DVC Pull Fails

Error Message:

```
ERROR: failed to pull data - file not found in remote
```

Causes:

1. Data was never pushed to remote
2. Remote configuration changed
3. Cache corruption

Solutions:

```
# Solution 1: Push from original machine
# (On machine with data)
dvc push

# Solution 2: Check cloud status
dvc status -c

# Solution 3: Clear and re-pull cache
rm -rf .dvc/cache
dvc fetch
dvc checkout

# Solution 4: Verify remote accessibility
dvc remote list -v
```

19.1.3 Issue 3: Pipeline Won't Re-run

Symptom:

```
Stage 'model_building' didn't change, skipping
Data and pipelines are up to date.
```

Cause: DVC detects no changes (this is actually correct behavior!)

Solutions:

```
# Force re-run entire pipeline
dvc repro --force

# Force re-run specific stage only
dvc repro --force model_building

# Force stage and all downstream stages
dvc repro --force --downstream model_building

# Remove specific stage output to trigger re-run
dvc remove model_building.dvc
dvc repro
```

19.1.4 Issue 4: Checksum Mismatch

Error Message:

```
ERROR: checksum mismatch for 'data/raw/train.csv'
```

Cause: File modified outside DVC or cache corruption

Solutions:

```
# Solution 1: Restore from remote
dvc checkout data/raw/train.csv

# Solution 2: Clear cache and re-pull
rm -rf .dvc/cache
dvc pull
```

```
# Solution 3: If file was intentionally modified
dvc add data/raw/train.csv
git add data/raw/train.csv.dvc
git commit -m "Update data"
```

19.2 MLflow Issues

19.2.1 Issue 5: MLflow Not Logging Runs

Symptom: Script completes but no runs appear in DagShub

Diagnosis and Solutions:

1. **Check DagShub Initialization**:

```python
import dagshub

# Must be at top of script
dagshub.init(
    repo_owner='username',
    repo_name='project-name',
    mlflow=True
)

```

2. **Verify Tracking URI**:

```python
import mlflow

# Check tracking URI
print(mlflow.get_tracking_uri())

# Should output:
# https://dagshub.com/username/project.mlflow

# If incorrect, set it:
mlflow.set_tracking_uri(
    "https://dagshub.com/username/project.mlflow"
)

```

3. **Check Internet Connection**:

```
# Test DagShub accessibility
curl https://dagshub.com

# Check if blocked by firewall
ping dagshub.com
```

4. **Verify MLflow Start Run**:

```python
# Ensure you're using context manager
with mlflow.start_run():
    mlflow.log_metric('accuracy', 0.95)
    mlflow.log_param('n_estimators', 100)

```

```
# Run gets finalized when exiting context

```

19.2.2 Issue 6: Cannot Find Run ID

Problem: Need run ID for tagging but don't know it

Solution:

```python
import mlflow
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Get experiment by name
experiment = mlflow.get_experiment_by_name(
    "MLOPS-Project-with-Random-Forest"
)

# List all runs in experiment
runs = client.search_runs(
    experiment_ids=[experiment.experiment_id],
    order_by=["metrics.accuracy DESC"]  # Sort by accuracy
)

# Print run IDs with metrics
for run in runs:
    print(f"Run ID: {run.info.run_id}")
    print(f"Metrics: {run.data.metrics}")
    print(f"Params: {run.data.params}")
    print("---")
```

19.3 Git Issues

19.3.1 Issue 7: dvc.lock Merge Conflicts

Symptom:

```
CONFLICT (content): Merge conflict in dvc.lock
```

Cause: Multiple people modified pipeline simultaneously
Solutions:

```
# Option 1: Accept their version
git checkout --theirs dvc.lock
dvc repro
git add dvc.lock
git commit -m "Resolve dvc.lock conflict"

# Option 2: Accept your version
git checkout --ours dvc.lock
dvc repro
git add dvc.lock
git commit -m "Resolve dvc.lock conflict"

# Option 3: Re-run pipeline (recommended)
```

```
git checkout --theirs dvc.lock
dvc repro --force
git add dvc.lock
git commit -m "Resolve conflict by re-running pipeline"
```

19.3.2 Issue 8: Large Files Committed to Git

Symptom:

```
remote: error: File data/train.csv is 150 MB;
    this exceeds GitHub's file size limit
```

Cause: Data files not in .gitignore, tracked by Git instead of DVC
Solutions:

```
# Step 1: Remove from Git tracking
git rm --cached data/train.csv

# Step 2: Add to .gitignore
echo "data/" >> .gitignore

# Step 3: Track with DVC
dvc add data/train.csv

# Step 4: Commit
git add data/train.csv.dvc .gitignore
git commit -m "Move data tracking from Git to DVC"

# If already pushed to Git:
# Use git filter-branch or BFG Repo-Cleaner
git filter-branch --tree-filter 'rm -f data/train.csv' HEAD
```

19.4 Environment Issues

19.4.1 Issue 9: Import Errors

Error Message:

```
ImportError: No module named 'pandas'
ModuleNotFoundError: No module named 'sklearn'
```

Solutions:

```
# Verify virtual environment is activated
which python    # Should show venv path
where python    # Windows

# Reinstall requirements
pip install -r requirements.txt

# Check installed packages
pip list | grep pandas

# If specific package missing
```

```
pip install pandas scikit-learn

# If version conflicts
pip install --upgrade pip
pip install -r requirements.txt --force-reinstall
```

19.4.2 Issue 10: NLTK Data Not Found

Error Message:

```
LookupError: Resource 'stopwords' not found
```

Solution:

```python
import nltk

# Download required NLTK data
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('punkt_tab')

# Verify download
from nltk.corpus import stopwords
print(len(stopwords.words('english')))  # Should print ~150

# Alternative: Download all NLTK data
nltk.download('all')  # Warning: Large download!
```

19.5 Performance Issues

19.5.1 Issue 11: Slow DVC Operations

Symptom: Very slow push/pull operations

Solutions:

```
# Use parallel jobs for faster transfers
dvc push -j 4    # Use 4 parallel jobs
dvc pull -j 8    # Use 8 parallel jobs

# Use hardlinks/symlinks for faster cache
dvc config cache.type hardlink,symlink

# Move cache to faster storage (SSD)
dvc cache dir /path/to/ssd/cache

# For AWS S3, tune buffer size
dvc remote modify myremote buffer_size 10485760
```

19.5.2 Issue 12: Large Cache Size

Symptom: .dvc/cache/ consuming too much disk space

Solutions:

```
# Remove unused cache files
dvc gc --workspace    # Keep only workspace files
dvc gc --cloud        # Keep only files in remote

# Check cache size
du -sh .dvc/cache

# Move cache to external drive
dvc cache dir /mnt/external/cache

# Use shared cache for team
dvc config cache.dir /shared/team/cache
dvc config cache.shared group
```

20 Quick Reference Guide

20.1 Essential Commands

DVC Commands

```
dvc init                          # Initialize DVC
dvc remote add -d name url        # Add default remote
dvc remote list                   # List remotes
dvc remote list -v                # List with URLs
dvc add <file>                    # Track file
dvc push                          # Upload to remote
dvc pull                          # Download from remote
dvc fetch                         # Download without checkout
dvc checkout                      # Update workspace
dvc repro                         # Run pipeline
dvc repro --force                 # Force re-run
dvc status                        # Check status
dvc status -c                     # Compare with cloud
dvc diff                          # Show differences
dvc dag                           # Visualize pipeline
dvc gc                            # Garbage collect cache
```

Git Commands

```
git init                          # Initialize repository
git add .                         # Stage all changes
git add <file>                    # Stage specific file
git commit -m "message"           # Commit changes
git push origin main              # Push to remote
git pull origin main              # Pull from remote
git status                        # Check status
git log --oneline                 # View commit history
git checkout <commit>             # Switch to commit
git checkout main                 # Return to main
git branch                        # List branches
git diff                          # Show differences
```

MLflow Commands

```
# In Python code:
import mlflow

mlflow.set_tracking_uri(url)   # Set tracking server
mlflow.set_experiment(name)    # Set experiment
mlflow.start_run()              # Start tracking run
mlflow.log_metric(key, value)  # Log metric
mlflow.log_param(key, value)   # Log parameter
mlflow.log_artifact(path)      # Log file
mlflow.end_run()                # End run

# Using context manager (recommended):
with mlflow.start_run():
    mlflow.log_metric('accuracy', 0.95)
    mlflow.log_param('n_estimators', 100)
```

20.2 Complete Workflow Summary

Phase 1: Experimentation

```
# 1. Modify parameters
vim params.yaml

# 2. Run pipeline
dvc repro

# 3. Check MLflow UI for results
# https://dagshub.com/username/project.mlflow

# 4. Repeat steps 1-3 for multiple experiments

# 5. Compare all runs in MLflow UI
# Choose best performing model
```

Phase 2: Freezing Best Model

```
# 6. Set params.yaml to winning configuration
# (Copy parameters from MLflow UI)

# 7. Verify reproducibility
dvc repro
cat reports/metrics.json

# 8. Freeze the decision
git add .
git commit -m "Freeze best model (AUC=0.918, acc=0.937)"
git push origin main

# 9. Push data and models
```

```
dvc push

# 10. Tag MLflow run (optional)
python tag_run.py
```

Phase 3: Reproduction on New Machine

```
# 11. Clone repository
git clone https://github.com/username/project.git
cd project

# 12. Setup environment
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate
pip install -r requirements.txt

# 13. Configure DVC remote (if needed)
dvc remote modify dvc_origin url <shared-path>

# 14. Pull data and models
dvc pull

# 15. Verify everything is ready
dvc status          # Should say "up to date"
dvc repro           # Should use cache (no re-run)
```

20.3 Project Structure Reference

```
MLOPS-Dagshub-DVC-Git-Project/
|
+-- .dvc/                       # DVC configuration
|   +-- cache/                  # Local data cache
|   +-- config                  # Remote storage config
|
+-- .git/                       # Git repository
|
+-- S3/                         # Local DVC remote (gitignored)
|
+-- src/                        # Source code
|   +-- Data_Ingestion.py
|   +-- Data_Pre_Processing.py
|   +-- Feature_Engineering.py
|   +-- Model_Building_Rf.py
|   +-- Model_Evaluation_Rf.py
|
+-- data/                       # Data files (DVC tracked)
|   +-- raw/
|   +-- interim/
|   +-- processed/
|
```

```
+-- models/                    # Model files (DVC tracked)
|   +-- model_rf.pkl
|
+-- reports/                   # Reports (DVC tracked)
|   +-- metrics.json
|
+-- logs/                      # Log files (gitignored)
|   +-- *.log
|
+-- dvc.yaml                   # Pipeline definition
+-- dvc.lock                   # Pipeline lock file
+-- params.yaml                # Hyperparameters
+-- requirements.txt           # Python dependencies
+-- .gitignore                 # Git ignore rules
+-- .dvcignore                 # DVC ignore rules
+-- README.md                  # Documentation
```

20.4 Key Files Explained

File	Purpose
dvc.yaml	Defines pipeline stages, dependencies, outputs, parameters
dvc.lock	Records exact file versions (MD5 hashes) for reproducibility
params.yaml	Centralized hyperparameter configuration
.gitignore	Tells Git what NOT to track (data/, models/, logs/)
.dvc/config	DVC remote storage configuration
.dvc/cache/	Local cache storing all data versions
requirements.txt	Python package dependencies with versions

20.5 Common Command Combinations

Frequently Used Sequences

```
# Complete experiment cycle
dvc repro && dvc push && git add . && \
git commit -m "Experiment X" && git push

# Quick status check
dvc status && git status

# Full pipeline re-run with push
dvc repro --force && dvc push && \
git add dvc.lock && git commit -m "Re-run pipeline" && git push

# Sync with team
git pull && dvc pull

# Clean up experiments and cache
dvc gc --workspace && dvc gc --cloud

# View pipeline and check status
dvc dag && dvc status
```

21 Comparison: DVC-Only vs DVC+MLflow+Git Workflow

21.1 Architecture Comparison

DVC-Only Workflow	DVC + MLflow + Git Workflow
Code Versioning	
Git for code versioning	Git for code versioning
Data & Model Versioning	
DVC for data/model versioning	DVC for data/model versioning
Experiment Tracking	
Local metrics.json file	Centralized MLflow server (DagShub)
Manual experiment comparison	Visual comparison in MLflow UI
Private to individual user	Shared across entire team
Limited experiment history	Complete experiment history
No run tagging	Git commit tagging support
Model Registry	
Manual model management	MLflow model registry
No deployment tracking	Deployment status tracking

21.2 Feature-by-Feature Comparison

Feature	DVC-Only	DVC+MLflow
Code versioning	✓	✓
Data versioning	✓	✓
Model versioning	✓	✓
Pipeline automation	✓	✓
Parameter tracking	✓	✓
Reproducibility	✓	✓
Team collaboration	Limited	✓
Visual metric comparison	×	✓
Centralized tracking	×	✓
Experiment history	Limited	✓
Run tagging	×	✓
Model registry	×	✓
Real-time monitoring	×	✓
Cross-model comparison	Manual	Automatic
Deployment tracking	×	✓

21.3 Workflow Process Comparison

21.3.1 Experimentation Phase

DVC-Only	DVC + MLflow
1. Modify params.yaml	1. Modify params.yaml
2. Run `dvc repro`	2. Run `dvc repro`
3. Check metrics.json manually	3. View results in MLflow UI
4. Copy metrics to spreadsheet	4. Compare visually in UI
5. Repeat for each experiment	5. All runs automatically tracked
6. Manually compare results	6. Sort/filter by any metric
7. Choose best from spreadsheet	7. Click to view best run

21.3.2 Model Freezing Phase

DVC-Only	DVC + MLflow
1. Identify best params manually 2. Update params.yaml 3. Run `dvc repro` 4. `git commit` 5. `dvc push` 6. No linking to experiments	1. Click best run in MLflow UI 2. Copy parameters from UI 3. Update params.yaml 4. Run `dvc repro` 5. `git commit` 6. `dvc push` 7. Tag MLflow run with Git commit 8. Mark as "production" in MLflow

21.4 Team Collaboration Comparison

Aspect	DVC-Only	DVC + MLflow
Experiment Visibility	Private/Local	Centralized/Shared
Metric Sharing	Manual (email/Slack)	Automatic (MLflow UI)
Best Model Decision	Individual/unclear	Transparent/documented
Experiment History	Lost after cleanup	Permanently preserved
Cross-team Learning	Difficult	Easy
Onboarding New Members	Manual explanation	Self-service in UI
Avoiding Duplicate Work	Hard to coordinate	Clear visibility

21.5 Use Case Recommendations

21.5.1 When to Use DVC-Only

- **Solo projects**: Individual research or learning
- **Simple pipelines**: Few experiments, straightforward workflows
- **Limited resources**: No budget for hosted services
- **Privacy constraints**: Cannot use external services
- **Prototype phase**: Early exploration, not production-ready
- **Academic research**: Focus on reproducibility over collaboration

21.5.2 When to Use DVC + MLflow + Git

- **Team projects**: Multiple data scientists collaborating
- **Production systems**: Models deployed to production
- **Complex experiments**: Many hyperparameter variations
- **Model comparison**: Testing multiple model architectures
- **Compliance needs**: Audit trails required
- **Continuous improvement**: Ongoing model iterations
- **Knowledge sharing**: Cross-functional teams
- **Model monitoring**: Track production model performance

21.6 Real-World Scenario Comparison

21.6.1 Scenario: Finding Best Model from 20+ Experiments

DVC-Only Approach

Process:

1. Run 20 experiments over several days
2. Copy metrics from each metrics.json to Excel
3. Manually create comparison charts
4. Search through Git commits to find parameters
5. Risk: Easy to lose track of which experiment was which
6. Risk: metrics.json overwritten between experiments

Time: 30+ minutes of manual work
Error prone: Yes, easy to lose experiment data

DVC + MLflow Approach

Process:

1. Run 20 experiments (automatically logged)
2. Open MLflow UI
3. Click "Compare" and select all runs
4. Sort by any metric (e.g., AUC)
5. View parallel coordinates plot
6. Click best run to see all details

Time: 2 minutes
Error prone: No, all data preserved automatically

21.6.2 Scenario: Reproducing 6-Month-Old Model

DVC-Only Approach

Challenges:

- Find Git commit (if tagged)
- No experiment metadata preserved
- No link to original metrics
- Manual verification needed
- Risk: Was this really the production model?

Confidence: Moderate

DVC + MLflow Approach

Process:

1. Search MLflow for "status:production"
2. Find run with "deployment_date: 2024-06"
3. See Git commit hash in tags
4. View exact metrics from that run
5. `git checkout <commit_hash>`
6. `dvc checkout`
7. Verified reproduction

Confidence: High

21.7 Cost-Benefit Analysis

Factor	DVC-Only	DVC + MLflow
Setup Time	30 minutes	45-60 minutes
Learning Curve	Moderate	Steeper initially
Per-Experiment Overhead	None	Minimal (auto-logged)
Experiment Comparison Time	15-30 minutes	2-5 minutes
Team Onboarding	Manual explanation	Self-service UI
Maintenance	Low	Low (hosted on DagShub)
Cost (DagShub Free Tier)	$0	$0
Scalability	Limited	Excellent
ROI for 1-person project	Good	Moderate
ROI for 3+ person team	Poor	Excellent

21.8 Migration Path

21.8.1 Upgrading from DVC-Only to DVC+MLflow

If you already have a DVC-only project, migration is straightforward:

1. **Install additional dependencies**:

```
pip install mlflow dagshub
```

2. **Add MLflow logging to evaluation script**:

```
# At top of Model_Evaluation.py
import mlflow
import dagshub

dagshub.init(repo_owner='...', repo_name='...', mlflow=True)
mlflow.set_tracking_uri("...")
mlflow.set_experiment("...")

# In main function, add:
with mlflow.start_run():
    mlflow.log_metric('accuracy', accuracy)
    mlflow.log_param('n_estimators', params['n_estimators'])
    # ... log all metrics and parameters

```

3. **No changes needed to**:
   - DVC pipeline (dvc.yaml)
   - Data versioning
   - Git workflow
   - Other pipeline stages
4. **Run pipeline**:

```
dvc repro
```

5. **Verify in MLflow UI**: Check that run appears
6. **Commit changes**:

```
git add .
git commit -m "Add MLflow experiment tracking"
git push
```

Result: All future experiments automatically tracked in MLflow, while maintaining all DVC benefits!

22 Conclusion and Next Steps

22.1 What You've Accomplished

By completing this guide, you now have a production-ready MLOps pipeline that:

Core Achievements

Version Control:

- ✓ Code versioned with Git
- ✓ Data versioned with DVC
- ✓ Models versioned with DVC
- ✓ Complete reproducibility guaranteed

Experiment Management:

- ✓ Centralized tracking with MLflow
- ✓ Visual metric comparison
- ✓ Complete experiment history
- ✓ Production model clearly identified

Team Collaboration:

- ✓ Shared experiment visibility
- ✓ Clear communication channel (DagShub)
- ✓ Avoid duplicate work
- ✓ Knowledge sharing enabled

Production Readiness:

- ✓ Automated pipeline execution
- ✓ Deployment tracking capability
- ✓ Audit trail for compliance
- ✓ Rollback capability to any version

22.2 Key Takeaways

22.2.1 The Three-System Architecture

Remember the Triangle

Git → Code + Metadata
DVC → Data + Models
MLflow → Experiments + Metrics

Each system has a specific purpose. Together, they provide complete MLOps infrastructure.

22.2.2 Critical Best Practices

1. **Experiment freely, freeze deliberately**:
   - Run as many experiments as needed
   - Only commit when you have a decision
   - Tag production runs explicitly
2. **Always push artifacts**:
   - `git push` for code
   - `dvc push` for data/models
   - Both are necessary for team work
3. **Link everything together**:
   - Git commits contain code + metadata
   - DVC tracks exact data versions
   - MLflow runs tagged with Git commits
   - Complete traceability
4. **Document your decisions**:
   - Clear commit messages
   - Experiment notes in MLflow
   - README updates
   - Why you chose this model

22.3 Common Pitfalls to Avoid

Warning

Watch Out For:

1. **Forgetting to push DVC artifacts**:
   - Always run `dvc push` after committing
   - Team cannot reproduce without artifacts
   - Set up reminder or automation
2. **Not linking MLflow runs to Git commits**:
   - Lost connection between code and experiments
   - Hard to reproduce later
   - Tag runs immediately
3. **Committing data files to Git**:
   - Verify .gitignore before committing

- Data should only be in DVC
- Check `git status` carefully

4. **Modifying params.yaml without re-running**:
   - Changes not reflected until `dvc repro`
   - dvc.lock becomes out of sync
   - Always repro after param changes
5. **Losing local DVC cache**:
   - Cache is critical for speed
   - Back up cache directory
   - Or ensure remote is always accessible

22.4 Next Steps and Extensions

22.4.1 Immediate Next Steps

1. **Add more models**:
   - XGBoost classifier
   - Logistic Regression baseline
   - Neural network (LSTM/BERT)
   - Compare in MLflow UI
2. **Improve monitoring**:

```
# Log additional metrics
mlflow.log_metric('train_time', train_time)
mlflow.log_metric('inference_time', inference_time)
mlflow.log_metric('model_size_mb', model_size)

```

3. **Add data validation**:
   - Check data quality in ingestion
   - Validate schema consistency
   - Alert on data drift
4. **Enhance visualizations**:

```
# Log confusion matrix
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(cm)
disp.plot()
plt.savefig('confusion_matrix.png')
mlflow.log_artifact('confusion_matrix.png')

```

22.4.2 Advanced Extensions

1. **CI/CD Integration**:
   - GitHub Actions for automated testing
   - Automatic pipeline runs on push
   - Model validation gates
   - Automated deployment to staging
2. **Model Registry**:

```
# Register model in MLflow
mlflow.sklearn.log_model(
    model,
    "spam_classifier",
    registered_model_name="SpamDetector"
)

# Transition to production
from mlflow.tracking import MlflowClient
client = MlflowClient()
client.transition_model_version_stage(
    name="SpamDetector",
    version=1,
    stage="Production"
)
```

3. **Hyperparameter Optimization**:

```
import optuna

def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 10, 100)
    max_depth = trial.suggest_int('max_depth', 5, 20)

    # Train and evaluate
    model = train_model(n_estimators, max_depth)
    score = evaluate_model(model)

    # Log to MLflow
    with mlflow.start_run(nested=True):
        mlflow.log_params(trial.params)
        mlflow.log_metric('accuracy', score)

    return score

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)
```

4. **Production Deployment**:
   - Deploy model as REST API (FastAPI)
   - Container with Docker
   - Kubernetes orchestration

- Model serving with MLflow

5. **Monitoring and Alerting**:
   - Track prediction distribution
   - Monitor model performance drift
   - Alert on accuracy degradation
   - Trigger retraining pipeline

22.5 Learning Resources

22.5.1 Official Documentation

- **DVC**: `https://dvc.org/doc`
- **MLflow**: `https://mlflow.org/docs/latest/index.html`
- **DagShub**: `https://dagshub.com/docs`
- **Git**: `https://git-scm.com/doc`

22.5.2 Community and Support

- **DVC Discord**: Active community for questions
- **MLflow Slack**: Official Slack workspace
- **DagShub Community**: Forum and discussions
- **Stack Overflow**: Tagged questions for all tools

22.5.3 Advanced Topics to Explore

1. **MLOps Maturity Models**:
   - Google's MLOps maturity framework
   - Microsoft's ML lifecycle
   - AWS well-architected ML
2. **Feature Stores**:
   - Feast (open source)
   - Tecton
   - AWS SageMaker Feature Store
3. **Model Monitoring**:
   - Evidently AI
   - WhyLabs
   - Arize AI
4. **Experiment Orchestration**:
   - Kubeflow Pipelines
   - Apache Airflow
   - Prefect

22.6 Final Thoughts

The MLOps Journey

This guide represents a **production-ready foundation** for MLOps, but MLOps is a journey, not a destination.

You now have:

- A reproducible workflow
- Team collaboration capability
- Experiment tracking infrastructure
- Clear path to production

Continue improving:

- Add more automation
- Enhance monitoring
- Improve documentation
- Share knowledge with team

Remember: The best MLOps system is one that your team actually uses. Start simple, iterate, and scale as needed.

22.7 Project Checklist

Use this checklist to verify your MLOps implementation:

Component	Status
Version Control	
Git repository initialized	✓
Code properly committed	✓
.gitignore configured	✓
Remote repository connected	✓
Data Versioning	
DVC initialized	✓
Data tracked with DVC	✓
DVC remote configured	✓
Artifacts pushed to remote	✓
Pipeline	
dvc.yaml created	✓
All stages defined	✓
params.yaml configured	✓
Pipeline runs successfully	✓
dvc.lock committed	✓
Experiment Tracking	
MLflow integrated	✓
DagShub connected	✓
Metrics logged	✓
Parameters logged	✓
Multiple experiments run	✓
Best Model	
Best model identified	✓
Configuration frozen	✓
Git commit created	✓
DVC artifacts pushed	✓
MLflow run tagged	✓
Documentation	
README updated	✓
Commit messages clear	✓
Experiment notes recorded	✓
Reproducibility	
Tested on new machine	✓
Team can reproduce	✓
All dependencies listed	✓

22.8 Get In Touch

Questions or Feedback?

Email: `sujil9480@gmail.com`

This guide is continuously updated. Check the repository for the latest version.

Version 1.0 - December 2025