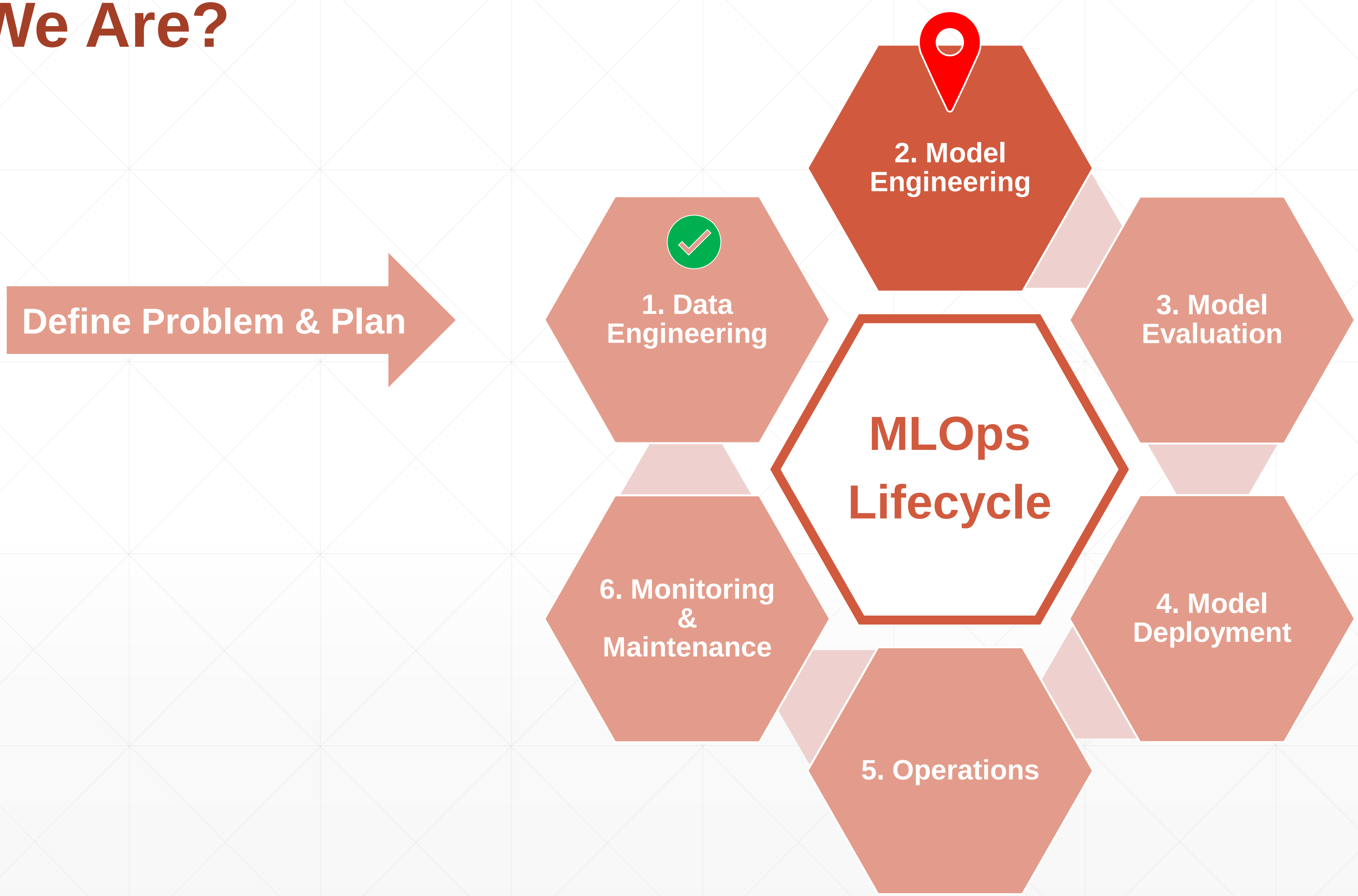
The Hoang

Model Development in MLOps

Understand and Implement Production-Grade Machine Learning Operations

Where We Are?


Define Problem & Plan
1. Data Engineering
2. Model Engineering
3. Model Evaluation
4. Model Deployment
5. Operations
6. Monitoring & Maintenance
MLOps Lifecycle

Recap 1/

Stage 1: Business Understanding & Problem Definition

- Identify the business problem.
- Define success metrics (e.g., accuracy, precision, recall).
- Understand constraints (data availability, computational resources).

DONE

Recap 2/

Stage 2: Data Engineering & Feature Engineering

- **Data Collection**: Gather relevant data from various sources (databases, APIs, logs).
- **Data Preprocessing**: Handle missing values, outliers, and inconsistencies.
- **Feature Engineering**: Transform raw data into meaningful features.
- **Data Versioning**: Use tools like **DVC**, **LakeFS**, or **MLflow** for data versioning.

DONE

Recap 3/

Stage 3: Model Development & Experimentation

- **Select Model Architecture**: Choose between traditional ML models (e.g., Random Forest, XGBoost) or deep learning models (CNNs, RNNs).
- **Hyperparameter Tuning**: Optimize parameters using techniques like grid search, random search, or Bayesian optimization.
- **Logging Experiments**: Use **MLflow**, **Weights & Biases**, or **TensorBoard** to track experiments.
- **Code Versioning**: Store code in **GitHub**, **GitLab**, or **Bitbucket**.

We Are Here

Recap 4/

Stage 4: Model Training & Evaluation

- **Train Model**: Use GPUs or TPUs for faster training.
- **Evaluate Performance**: Compute metrics like accuracy, RMSE, F1-score.
- **A/B Testing**: Compare different models on a validation set.
- **Bias & Fairness Testing**: Ensure the model does not introduce bias.

Coming Soon

Recap 5/

Stage 5: Model Packaging & Versioning

- **Convert Model**: Save models in formats like **ONNX**, **TF SavedModel**, or **MLflow Model Format**.
- **Version Control**: Use **MLflow Model Registry** or **DVC** for model versioning.
- **Containerization**: Package models using **Docker** for portability.

Coming Soon

Recap 6/

Stage 6: Model Deployment

- **Deploy as REST API**: Use **FastAPI**, **Flask**, or **TensorFlow Serving**.
- **Deploy to Cloud**: Use **AWS SageMaker**, **Azure ML**, or **Google Vertex AI**.
- **Deploy with Kubernetes**: Use **Kubeflow Serving** for scalable deployment.

Coming Soon

Recap 7/

Stage 7: Model Monitoring & Retraining

- **Monitor Performance**: Use **Prometheus**, **Grafana**, or **EvidentlyAI**.
- **Detect Data Drift**: Identify changes in data distribution.
- **Automate Retraining**: Set up pipelines for periodic model retraining.

Learning Objectives

- MLOps Model Development
- Reproducibility of Experiments
- Hands-On Practice with Mlflow Components

Mlflow Models

An MLflow model is a standard format that packages a machine learning model and its metadata, making it easy to deploy across different environments.

Each MLflow model contains:

- **Artifacts**: The actual model file (e.g., .pkl, .onnx, .h5, etc.)
- **Metadata**: Details about the model, such as the ML framework, version, and dependencies.
- **Flavors**: Standardized ways to load and use the model in different tools.

Mlflow Models - Structure

When you save a model using mlflow.log_model() or mlflow.save_model(), it is stored in the following directory structure:

```
model/
│── MLmodel            # Metadata about the model
│── model.pkl          # Serialized model file (format varies by framework)
│── conda.yaml         # Environment dependencies (optional)
│── requirements.txt   # Python dependencies (optional)
└── code/              # Source code dependencies (if logged)
```

Mlflow Models - Contents

glamorous-deer-515

Overview | Model metrics | System metrics | Traces | **Artifacts**

- model
  - MLmodel
  - conda.yaml
  - model.pkl
  - python_env.yaml
  - requirements.txt

model/MLmodel 526B

Path: mlflow-artifacts:/0/9d6b630c831740309c66de44bec6a4da/artifacts/model/MLmodel


```yaml
artifact_path: model
flavors:
  python_function:
    env:
      conda: conda.yaml
      virtualenv: python_env.yaml
    loader_module: mlflow.sklearn
    model_path: model.pkl
    predict_fn: predict
    python_version: 3.10.12
  sklearn:
    code: null
    pickled_model: model.pkl
    serialization_format: cloudpickle
    sklearn_version: 1.6.1
mlflow_version: 2.20.2
model_size_bytes: 835
model_uuid: 8be0fd417e8d4118a3a6410349354d1f
run_id: 9d6b630c831740309c66de44bec6a4da
utc_time_created: '2025-03-17 16:18:51.698802'
```

MLflow Flavors

A flavor is a standardized way of saving and loading models in different frameworks. MLflow supports multiple flavors, including:

MLflow Flavor	Description
mlflow.sklearn	Scikit-learn models
mlflow.tensorflow	TensorFlow/Keras models
mlflow.pytorch	PyTorch models
mlflow.xgboost	XGBoost models
mlflow.lightgbm	LightGBM models
mlflow.onnx	ONNX models
mlflow.spark	Apache Spark MLlib models
mlflow.h2o	H2O.ai models
mlflow.statsmodels	StatsModels models
mlflow.pyfunc	Generic Python function mode

MLflow Flavors

Example with Scikit-learn Flavor:

```python
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load dataset
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)

# Log model in MLflow
mlflow.sklearn.log_model(model, artifact_path="random_forest_model")
```

Autolog

```python
# Automatically log model and metrics
mlflow.FLAVOR.autolog()
```

```python
# Scikit-learn built-in flavor
mlflow.sklearn.autolog()
```

```python
# Import scikit-learn
import mlflow
from sklearn.linear_model import \
    LinearRegression

# Using auto-logging
mlflow.sklearn.autolog()
```

```python
# Train the model
lr = LinearRegression()
lr.fit(X, y)
```

Model will be logged automatically on `model.fit()`

Common Metrics

- Regression
  - mean squared error
  - root mean squared error
  - mean absolute error
  - r2 score
- Classification
  - precision score
  - recall score
  - f1 score
  - accuracy score

Common parameters

```
MODEL.get_params()
```

```python
# Train the model
lr = LinearRegression()
lr.fit(X, y)
# Get params
params = lr.get_params(deep=True)
params
```

```
{'copy_X': True, 'fit_intercept': True, 'n_jobs': None,
    'normalize': 'deprecated', 'positive': False}
```

The Model API

```
# Save a model to the local filesystem
mlflow.sklearn.save_model(model, path)
```

```
# Log a model as an artifact to MLflow Tracking.
mlflow.sklearn.log_model(model, artifact_path)
```

```
# Load a model from local filesystem or from MLflow Tracking.
mlflow.sklearn.load_model(model_uri)
```

The Model API - Save

```
# Model
lr = LogisticRegression()
lr.fit(X, y)

# Save model locall
mlflow.sklearn.save_model(lr, "local_path")
```

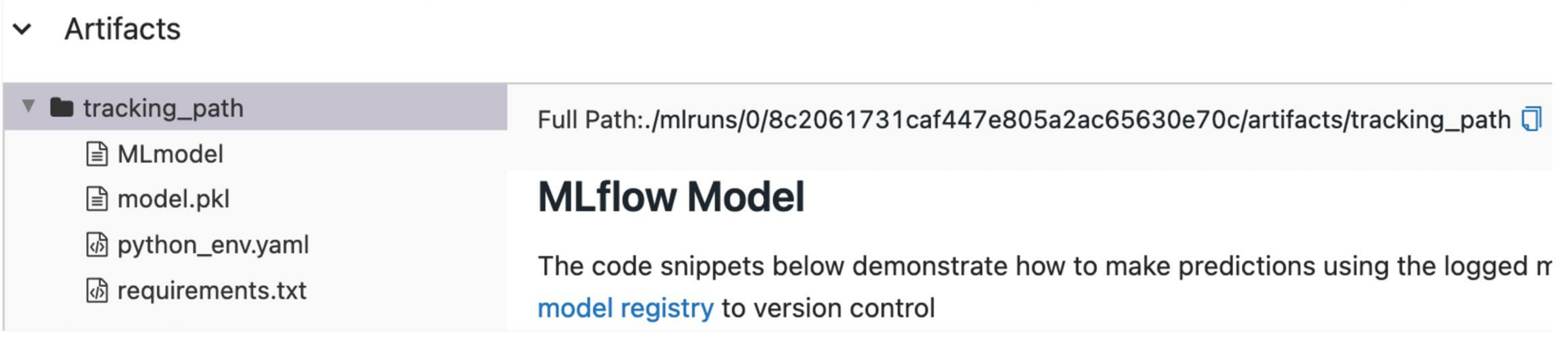
```
ls local_path/
```

```
MLmodel             model.pkl        requirements.txt        python_env.yaml
```

The Model API - Log

```python
# Model
lr = LogisticRegression(n_jobs=n_jobs)
lr.fit(X, y)

# Log model
mlflow.sklearn.log_model(lr, "tracking_path")
```



The Model API - Load

- Local Filesystem - `relative/path/to/local/model` or `/Users/me/path/to/local/model`
- MLflow Tracking - `runs:/<mlflow_run_id>/run-relative/path/to/model`
- S3 Support - `s3://my_bucket/path/to/model`

```
# Load model from local path
model = mlflow.sklearn.load_model("local_path")

# Show model
model
```

```
LogisticRegression()
```

The Model API - Load

- Local Filesystem - `relative/path/to/local/model` or `/Users/me/path/to/local/model`
- MLflow Tracking - `runs:/<mlflow_run_id>/run-relative/path/to/model`
- S3 Support - `s3://my_bucket/path/to/model`

```python
# Pass run_id as f-string literal
model = mlflow.sklearn.load_model(f"runs:/{run_id}/tracking_path")
# Show model
model
```

```
LogisticRegression()
```

Let’s Practice – MLflow Models

Custom Python models

If none of the built-in flavors fit your needs, you can define a custom MLflow model using the mlflow.pyfunc flavor.

- Built in Flavor - `python_function`
- `mlflow.pyfunc`
  - `save_model()`
  - `log_model()`
  - `load_model()`

Custom Python model class

If none of the built-in flavors fit your needs, you can define a custom MLflow model using the mlflow.pyfunc flavor.

- Custom model class
  - `MyClass(mlflow.pyfunc.PythonModel)`
- PythonModel class
  - `load_context()` - loads artifacts when `mlflow.pyfunc.load_model()` is called
  - `predict()` - takes model input and performs user defined evaluation

Custom Python model class - Example

```
import mlflow.pyfunc

# Define the model class
class CustomPredict(mlflow.pyfunc.PythonModel):
    # Load artifacts
    def load_context(self, context):
        self.model = mlflow.sklearn.load_model(context.artifacts["custom_model"])
    # Evaluate input using custom_function()
    def predict(self, context, model_input):
        prediction = self.model.predict(model_input)
        return custom_function(prediction)
```

Custom Python model – Save & Logging

```python
# Save model to local filesystem
mlflow.pyfunc.save_model(path="custom_model", python_model=CustomPredict())
```

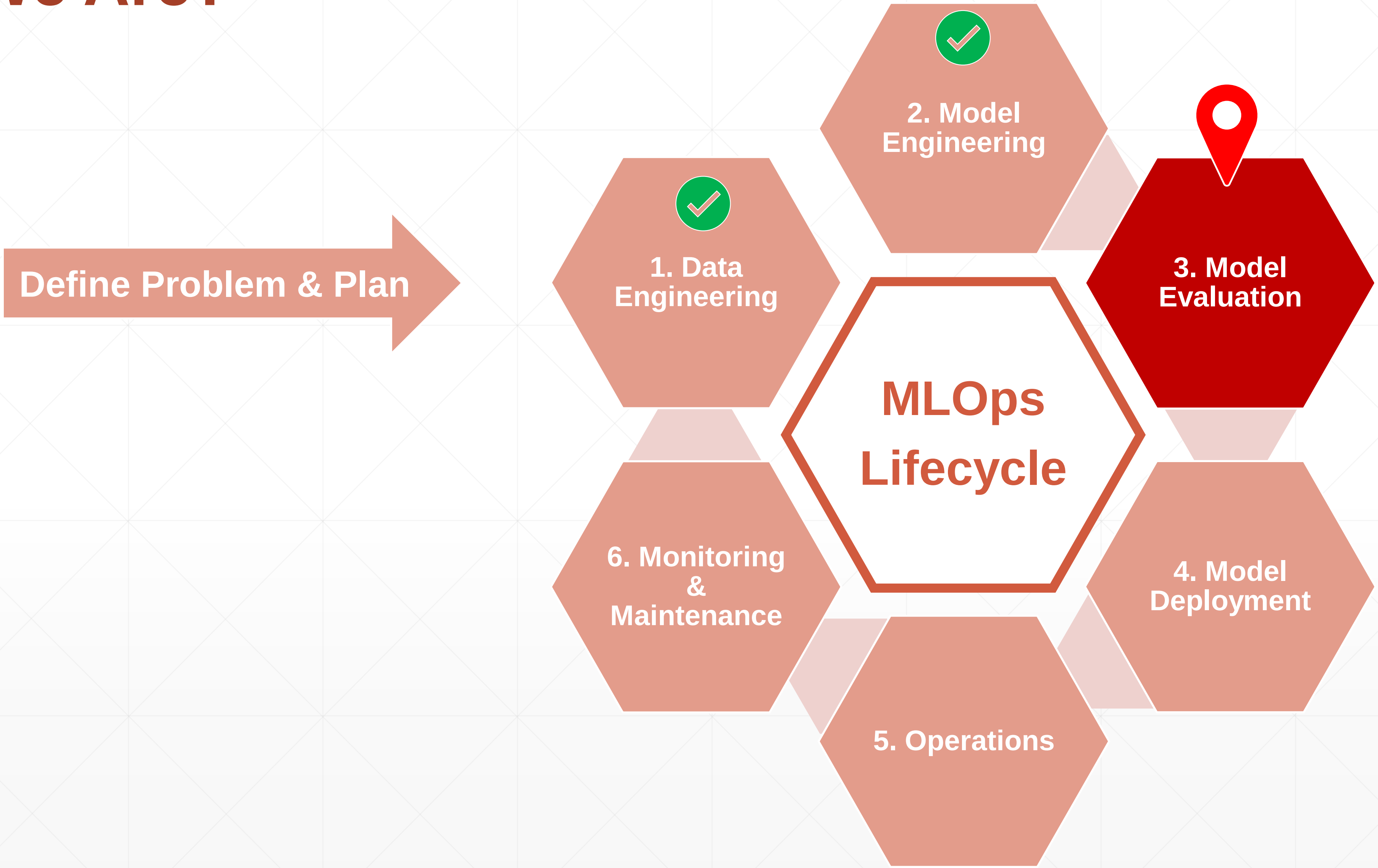
```python
# Log model to MLflow Tracking
mlflow.pyfunc.log_model(artifact_path="custom_model", python_model=CustomPredict())
```

Custom Python model – Load

```
# Load model from local filesystem
mlflow.pyfunc.load_model("local")
```

```
# Load model from MLflow Tracking
mlflow.pyfunc.load_model("runs:/run_id/tracking_path")
```

Where We Are?



Define Problem & Plan
1. Data Engineering
2. Model Engineering
3. Model Evaluation
4. Model Deployment
5. Operations
6. Monitoring & Maintenance
MLOps Lifecycle

Model Evaluation

MLflow provides built-in support for evaluating models using the **mlflow.evaluate()** API.

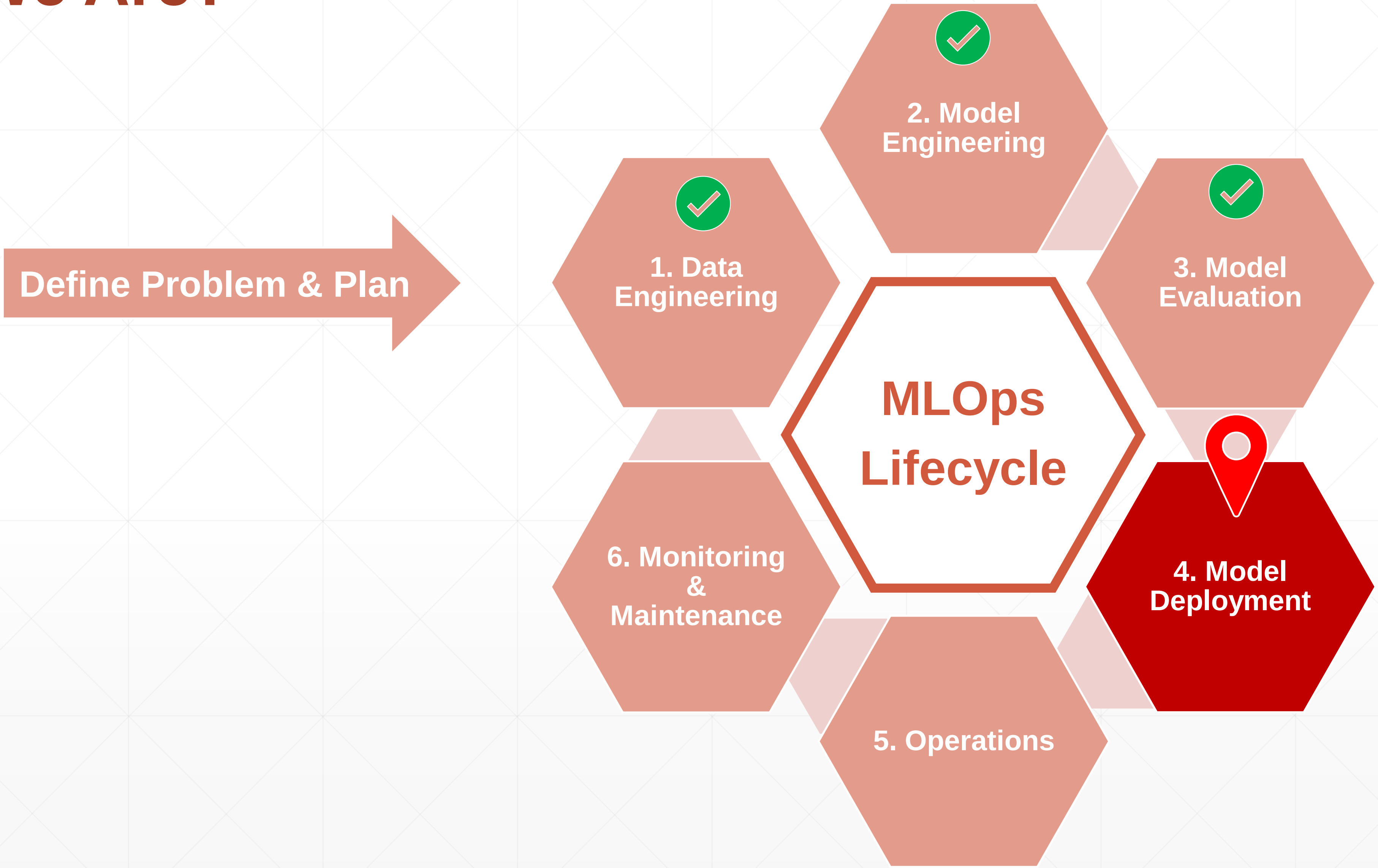
```python
# Training Data
X_train, X_test, y_train, y_test = \
    train_test_split(X, y,
    train_size=0.7,random_state=0)

# Linear Regression model
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```python
# Dataset
eval_data = X_test
eval_data["test_label"] = y_test

# Evaluate model with Dataset
mlflow.evaluate(
    "runs:/run_id/model",
    eval_data,
    targets="test_label",
    model_type="regressor"
)
```

Where We Are?



Define Problem & Plan
1. Data Engineering
2. Model Engineering
3. Model Evaluation
4. Model Deployment
5. Operations
6. Monitoring & Maintenance
MLOps Lifecycle

Model Severing - Command

```
# MLflow serve command
mlflow models serve --help
Usage: mlflow models serve [OPTIONS]
```

Model Severing - Command

```
# Local Filesystem
mlflow models serve -m relative/path/to/local/model
```

```
# Run ID
mlflow models serve -m runs:/<mlflow_run_id>/artifacts/model
```

```
# AWS S3
mlflow models serve -m s3://my_bucket/path/to/model
```

Model Severing - APIs

MLflow allows serving models via REST APIs using MLflow Model Serving.

- `/ping` - for health checks
- `/health` - for health checks
- `/version` - for getting the version of MLflow
- `/invocations` - for model scoring
- Port 5000

Model Severing - Invocations endpoint

`/invocations`

```
No,Name,Subject
1,Bill Johnson,English
2,Gary Valentine,Mathematics
```

`Content-Type` : `application/json` or `application/csv`

```
{
    "1": {
        "No": "1",
        "Name": "Bill Johnson",
        "Subject": "English"
    },
    "2": {
        "No": "2",
        "Name": "Gary Valentine",
        "Subject": "Mathematics"
    }
}
```

Model Severing - Invocations input data

CSV format

- Pandas Dataframe
- `pandas_df.to_csv()`

JSON format

- `dataframe_split` - pandas DataFrame in split orientation
- `dataframe_records` - pandas DataFrame in records orientation

Model Severing - Invocations input data

When sending data to an MLflow model serving invocation endpoint, the data needs to be formatted correctly in JSON. One of the supported formats is the Pandas DataFrame split format.

What is Pandas DataFrame Split Format?
The split format is a way of structuring tabular data where:

- Columns are explicitly defined.
- Index is included (optional).
- Data is provided as a list of lists (each inner list represents a row).

This format is useful for structured data, such as tabular datasets used in ML models.

Model Severing - Invocations input data

Example – DataFrame Split

```python
import pandas as pd
import json

# Create a sample DataFrame
df = pd.DataFrame({
    "feature1": [5.1, 4.9, 6.2],
    "feature2": [3.5, 3.0, 2.8],
    "feature3": [1.4, 1.4, 4.8],
    "feature4": [0.2, 0.2, 1.8]
})

# Convert DataFrame to Pandas Split format JSON
data_json = df.to_json(orient="split")

# Print the JSON data
print(json.dumps(json.loads(data_json), indent=4))
```


```json
{
    "columns": ["feature1", "feature2", "feature3", "feature4"],
    "index": [0, 1, 2],
    "data": [
        [5.1, 3.5, 1.4, 0.2],
        [4.9, 3.0, 1.4, 0.2],
        [6.2, 2.8, 4.8, 1.8]
    ]
}
```

Model Severing - Invocations input data

Example – DataFrame Split

```python
import requests
import json

# Define the MLflow model serving endpoint
model_url = "http://127.0.0.1:5001/invocations"

# Prepare input data in Pandas Split format
input_data = {
    "columns": ["feature1", "feature2", "feature3", "feature4"],
    "data": [
        [5.1, 3.5, 1.4, 0.2],  # First sample
        [4.9, 3.0, 1.4, 0.2]   # Second sample
    ]
}

# Send POST request
response = requests.post(model_url, json=input_data)

# Print response (model predictions)
print(response.json())
```

Model Severing - Invocations input data

Pandas DataFrame in Records Orientation
The records orientation is another way of formatting a Pandas DataFrame when converting it to JSON.

Definition
- The records format represents each row as a dictionary (JSON object).
- The keys in each dictionary correspond to the column names.
- The result is a list of dictionaries, where each dictionary represents a row.

Model Severing - Invocations input data

Example – DataFrame Records

```python
import pandas as pd
import json

# Create a sample DataFrame
df = pd.DataFrame({
    "feature1": [5.1, 4.9, 6.2],
    "feature2": [3.5, 3.0, 2.8],
    "feature3": [1.4, 1.4, 4.8],
    "feature4": [0.2, 0.2, 1.8]
})

# Convert DataFrame to JSON with 'records' orientation
data_json = df.to_json(orient="records")

# Print the JSON data
print(json.dumps(json.loads(data_json), indent=4))
```



```json
[
    {
        "feature1": 5.1,
        "feature2": 3.5,
        "feature3": 1.4,
        "feature4": 0.2
    },
    {
        "feature1": 4.9,
        "feature2": 3.0,
        "feature3": 1.4,
        "feature4": 0.2
    },
    {
        "feature1": 6.2,
        "feature2": 2.8,
        "feature3": 4.8,
        "feature4": 1.8
    }
]
```

Let’s Practice – MLflow Models