Azure ML Studio Deployment Guide

Track Clustering Model with Retraining Pipeline

📋 Table of Contents

1️⃣ Prerequisites

A. Install Required Tools

```bash
# Install Azure CLI
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# Install Azure ML SDK
pip install azure-ai-ml azure-identity azureml-core

# Login to Azure
az login
```

B. Required Files Structure

```
project/
├── src/
│   ├── train.py              # Training script
```

```
│   ├── score.py              # Inference script
│   └── track_analyzer.py     # Your refactored code
├── environments/
│   └── conda.yml             # Dependencies
├── data/
│   ├── track_with_elevation.csv
│   ├── device_track_1.csv
│   └── emergency_events.csv
└── pipelines/
    ├── training_pipeline.py  # Training pipeline
    └── inference_pipeline.py # Inference pipeline
```

2️⃣ Setup Azure ML Workspace

A. Via Azure Portal

1. Login ke Azure Portal
2. Klik **Create a resource** → Search "Machine Learning"
3. Klik **Create** → **New workspace**
4. Isi form:
   - **Subscription**: Pilih subscription
   - **Resource group**: Create new atau pilih existing
   - **Workspace name**: `track-clustering-ml`
   - **Region**: `Southeast Asia` (terdekat dengan Indonesia)
   - **Storage account**: Auto-create
   - **Key vault**: Auto-create
   - **Application insights**: Auto-create
   - **Container registry**: Auto-create
5. Klik **Review + Create** → **Create**
6. Wait 5-10 menit untuk provisioning

B. Via Azure CLI (Alternative)

```
bash
```

```
# Set variables
RG_NAME="track-clustering-rg"
WORKSPACE_NAME="track-clustering-ml"
LOCATION="southeastasia"

# Create resource group
az group create --name $RG_NAME --location $LOCATION

# Create ML workspace
az ml workspace create \
  --name $WORKSPACE_NAME \
  --resource-group $RG_NAME \
  --location $LOCATION
```

C. Connect to Workspace

```python
from azure.ai.ml import MLClient
from azure.identity import DefaultAzureCredential

# Connect to workspace
ml_client = MLClient(
    credential=DefaultAzureCredential(),
    subscription_id="YOUR_SUBSCRIPTION_ID",
    resource_group_name="track-clustering-rg",
    workspace_name="track-clustering-ml"
)
```

---

3 Prepare Training Script

A. Create `environments/conda.yml`

```yaml
```

```yaml
name: track_clustering_env
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.9
  - pip
  - pip:
      - pandas==2.0.3
      - numpy==1.24.3
      - scikit-learn==1.3.0
      - joblib==1.3.2
      - azureml-core==1.54.0
      - azure-ai-ml==1.11.0
```

B. Create `src/train.py`

```python
```

```
"""
Azure ML Training Script
This script wraps TrackAnalyzer for Azure ML integration
"""
import argparse
import os
import joblib
import mlflow
import mlflow.sklearn

# Import your TrackAnalyzer class
from track_analyzer import TrackAnalyzer


def parse_args():
    """Parse command line arguments from Azure ML"""
    parser = argparse.ArgumentParser(description="Train track clustering model in Azure ML")

    # Data paths
    parser.add_argument("--data-path", type=str, required=True,
                        help="Path to input data folder")
    parser.add_argument("--output-path", type=str, required=True,
                        help="Path to output model folder")

    # Model parameters
    parser.add_argument("--n-clusters", type=int, default=3,
                        help="Number of clusters (if not tuning)")
    parser.add_argument("--max-iter", type=int, default=300,
                        help="Maximum iterations (if not tuning)")

    # Tuning flag
    parser.add_argument("--enable-tuning", type=str, default="false",
                        help="Enable hyperparameter tuning (true/false)")

    return parser.parse_args()


def main():
    """Main training function for Azure ML"""
    args = parse_args()

    # Convert string to boolean
    enable_tuning = args.enable_tuning.lower() == "true"
```

```
print("="*60)
print("AZURE ML TRAINING - TRACK CLUSTERING MODEL")
print("="*60)
print(f"Data path: {args.data_path}")
print(f"Output path: {args.output_path}")
print(f"Enable tuning: {enable_tuning}")
print("="*60)

# Start MLflow run for tracking
mlflow.start_run()

# Log input parameters
mlflow.log_param("enable_tuning", enable_tuning)
if not enable_tuning:
    mlflow.log_param("n_clusters", args.n_clusters)
    mlflow.log_param("max_iter", args.max_iter)

try:
    # Initialize analyzer (uses your refactored code!)
    analyzer = TrackAnalyzer()

    # Load data from Azure storage
    print("\n[1/6] Loading data...")
    analyzer.load_data(
        f"{args.data_path}/track_with_elevation.csv",
        f"{args.data_path}/device_track_1.csv",
        f"{args.data_path}/emergency_events.csv"
    )

    # Generate segments
    print("\n[2/6] Generating segments...")
    analyzer.generate_segments()

    # Map data to segments
    print("\n[3/6] Mapping data to segments...")
    analyzer.map_data_to_segments()

    # Engineer features
    print("\n[4/6] Engineering features...")
    analyzer.engineer_features()

    # Training
    print("\n[5/6] Training model...")
```

```
if enable_tuning:
    print("Running hyperparameter tuning...")

    # Define parameter grid for Azure (smaller for faster execution)
    param_grid = {
        'n_clusters': [3, 4, 5],
        'max_iter': [100, 300],
        'n_init': [10, 20],
        'algorithm': ['lloyd'],
        'scaler': ['standard', 'robust']
    }

    df_results, best_params = analyzer.tune_hyperparameters(param_grid=param_grid)
    analyzer.apply_best_params(best_params)

    # Log best parameters to MLflow
    for key, value in best_params.items():
        if key == 'features':
            mlflow.log_param(key, ','.join(value))
        else:
            mlflow.log_param(key, value)

    # Save tuning results
    df_results.to_csv(f"{args.output_path}/tuning_results.csv", index=False)

else:
    print("Training with specified parameters...")
    analyzer.train_clustering_model(
        n_clusters=args.n_clusters,
        max_iter=args.max_iter
    )

# Evaluate model
print("\n[6/6] Evaluating model...")
metrics = analyzer.evaluate_clustering()

# Log metrics to MLflow
mlflow.log_metric("calinski_harabasz", metrics['calinski_harabasz'])
mlflow.log_metric("davies_bouldin", metrics['davies_bouldin'])
mlflow.log_metric("silhouette", metrics['silhouette'])

# Create output directory
os.makedirs(args.output_path, exist_ok=True)
```

```
    # Save model artifacts using TrackAnalyzer's save method
    # But adjust paths for Azure ML
    model_path = f"{args.output_path}/model.joblib"
    scaler_path = f"{args.output_path}/scaler.joblib"
    features_path = f"{args.output_path}/features.joblib"

    joblib.dump(analyzer.kmeans, model_path)
    joblib.dump(analyzer.scaler, scaler_path)

    feature_info = {
        'feature_cols': analyzer.feature_cols,
        'best_params': analyzer.best_params
    }
    joblib.dump(feature_info, features_path)

    # Save segmented data
    analyzer.df_seg.to_csv(f"{args.output_path}/df_seg.csv", index=False)

    # Log model to MLflow (for versioning)
    mlflow.sklearn.log_model(
        analyzer.kmeans,
        "model",
        registered_model_name="track-clustering-model"
    )

    # Log artifacts
    mlflow.log_artifact(scaler_path)
    mlflow.log_artifact(features_path)

    print("\n" + "="*60)
    print("TRAINING COMPLETED SUCCESSFULLY!")
    print("="*60)
    print(f"Model saved to: {model_path}")
    print(f"Scaler saved to: {scaler_path}")
    print(f"Features saved to: {features_path}")
    print(f"Silhouette Score: {metrics['silhouette']:.4f}")

except Exception as e:
    print(f"\n❌ Training failed: {str(e)}")
    mlflow.log_param("status", "failed")
    mlflow.log_param("error", str(e))
    raise

finally:
```

```
    mlflow.end_run()


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

Key Differences Summary:

Aspect	`track_analyzer.py`	`train.py`
Purpose	Reusable library	Azure ML wrapper
Dependencies	Only sklearn, pandas	+ Azure ML, MLflow
Input	Local file paths	Azure storage paths
Output	Local files	Azure registry + local
Tracking	Print statements	MLflow tracking
Usage	`analyzer.run_pipeline()`	`python train.py --args`
Environment	Anywhere	Azure ML compute

Best Practice: `train.py` should be a **thin wrapper** that:

1. Handles Azure-specific I/O
2. Parses command-line arguments
3. Logs to MLflow
4. Calls `TrackAnalyzer` methods
5. Saves to Azure storage

This way:

- ✅ No code duplication
- ✅ Easy to test locally with `TrackAnalyzer`
- ✅ Easy to deploy to Azure with `train.py`
- ✅ Single source of truth for training logic

C. Create `src/score.py` (Inference Script)

```
python
```

```
"""
Azure ML Inference Script
"""
import json
import joblib
import numpy as np
import pandas as pd
import os


def init():
    """Initialize model and scaler"""
    global model, scaler, feature_cols

    # Get model path
    model_path = os.path.join(os.getenv("AZUREML_MODEL_DIR"), "model.joblib")
    scaler_path = os.path.join(os.getenv("AZUREML_MODEL_DIR"), "scaler.joblib")

    # Load model and scaler
    model = joblib.load(model_path)
    scaler = joblib.load(scaler_path)

    # Define feature columns (should match training)
    feature_cols = ['slope', 'curvature', 'offtrack_rate',
                    'density', 'stuck_rate', 'sos_rate']

    print("Model and scaler loaded successfully")


def run(raw_data):
    """
    Make predictions on input data

    Input format:
    {
        "data": [
            {
                "slope": 0.05,
                "curvature": 0.2,
                "offtrack_rate": 0.1,
                "density": 50,
                "stuck_rate": 0.02,
                "sos_rate": 0.01
```

```
        }
    ]
}
"""
try:
    # Parse input
    data = json.loads(raw_data)
    df = pd.DataFrame(data['data'])

    # Validate features
    missing_cols = set(feature_cols) - set(df.columns)
    if missing_cols:
        return json.dumps({
            "error": f"Missing columns: {missing_cols}"
        })

    # Select and order features
    X = df[feature_cols]

    # Scale features
    X_scaled = scaler.transform(X)

    # Predict
    predictions = model.predict(X_scaled)

    # Get cluster centers distance (for confidence)
    distances = model.transform(X_scaled)
    min_distances = np.min(distances, axis=1)

    # Prepare response
    results = []
    for i, pred in enumerate(predictions):
        results.append({
            "cluster": int(pred),
            "distance_to_center": float(min_distances[i]),
            "input": df.iloc[i].to_dict()
        })

    return json.dumps({
        "predictions": results
    })

except Exception as e:
    return json.dumps({
```

```
        "error": str(e)
    })
```

---

4 Create Azure ML Pipeline

A. Upload Data to Azure ML Datastore

```python
from azure.ai.ml.entities import Data
from azure.ai.ml.constants import AssetTypes

# Create data asset
data_path = "./data"

data_asset = Data(
    name="track-clustering-data",
    description="Track clustering training data",
    path=data_path,
    type=AssetTypes.URI_FOLDER
)

ml_client.data.create_or_update(data_asset)
```

B. Register Environment

```python
from azure.ai.ml.entities import Environment

env = Environment(
    name="track-clustering-env",
    description="Environment for track clustering",
    conda_file="./environments/conda.yml",
    image="mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest"
)

ml_client.environments.create_or_update(env)
```

C. Create Training Pipeline `pipelines/training_pipeline.py`

```python
```

```
from azure.ai.ml import MLClient, command, Input, Output
from azure.ai.ml.entities import Environment
from azure.ai.ml.dsl import pipeline
from azure.identity import DefaultAzureCredential

# Connect to workspace
ml_client = MLClient(
    credential=DefaultAzureCredential(),
    subscription_id="YOUR_SUBSCRIPTION_ID",
    resource_group_name="track-clustering-rg",
    workspace_name="track-clustering-ml"
)

# Define training component
training_component = command(
    name="train_clustering_model",
    display_name="Train Track Clustering Model",
    description="Train K-Means clustering model on track segments",
    code="./src",
    command="""
        python train.py \
        --data-path ${{inputs.data_path}} \
        --output-path ${{outputs.model_output}} \
        --n-clusters ${{inputs.n_clusters}} \
        --max-iter ${{inputs.max_iter}} \
        --enable-tuning ${{inputs.enable_tuning}}
    """,
    environment="track-clustering-env@latest",
    inputs={
        "data_path": Input(type="uri_folder"),
        "n_clusters": Input(type="integer", default=3),
        "max_iter": Input(type="integer", default=300),
        "enable_tuning": Input(type="boolean", default=False)
    },
    outputs={
        "model_output": Output(type="uri_folder")
    },
    compute="cpu-cluster"  # Will create later
)

# Define pipeline
@pipeline(
    name="track_clustering_training_pipeline",
```

```
    description="End-to-end pipeline for training track clustering model",
    default_compute="cpu-cluster"
)
def training_pipeline(
    pipeline_data,
    n_clusters: int = 3,
    max_iter: int = 300,
    enable_tuning: bool = False
):
    """Training pipeline"""

    train_job = training_component(
        data_path=pipeline_data,
        n_clusters=n_clusters,
        max_iter=max_iter,
        enable_tuning=enable_tuning
    )

    return {
        "model_output": train_job.outputs.model_output
    }

# Create pipeline instance
pipeline_job = training_pipeline(
    pipeline_data=Input(
        type="uri_folder",
        path="azureml://datastores/workspaceblobstore/paths/track-clustering-data"
    ),
    n_clusters=3,
    max_iter=300,
    enable_tuning=True
)

# Submit pipeline
pipeline_job = ml_client.jobs.create_or_update(
    pipeline_job,
    experiment_name="track_clustering_experiment"
)

print(f"Pipeline submitted: {pipeline_job.name}")
```

D. Create Compute Cluster

```
bash
```

```
# Via Azure CLI
az ml compute create \
  --name cpu-cluster \
  --type AmlCompute \
  --min-instances 0 \
  --max-instances 4 \
  --size Standard_DS3_v2 \
  --resource-group track-clustering-rg \
  --workspace-name track-clustering-ml
```

Or via Python:

```python
from azure.ai.ml.entities import AmlCompute

cluster = AmlCompute(
    name="cpu-cluster",
    type="amlcompute",
    size="Standard_DS3_v2",
    min_instances=0,
    max_instances=4,
    idle_time_before_scale_down=120
)

ml_client.compute.begin_create_or_update(cluster)
```

---

5 Register Model

A. Register Model After Training

```python
```

```python
from azure.ai.ml.entities import Model
from azure.ai.ml.constants import AssetTypes

# Register model
model = Model(
    name="track-clustering-model",
    path="azureml://jobs/<JOB_ID>/outputs/model_output",
    type=AssetTypes.CUSTOM_MODEL,
    description="K-Means clustering model for track segmentation",
    tags={
        "framework": "sklearn",
        "task": "clustering"
    }
)

registered_model = ml_client.models.create_or_update(model)
print(f"Model registered: {registered_model.name}:{registered_model.version}")
```

6️⃣ Deploy Inference Endpoint

A. Create Managed Online Endpoint

```python
from azure.ai.ml.entities import ManagedOnlineEndpoint

# Create endpoint
endpoint = ManagedOnlineEndpoint(
    name="track-clustering-endpoint",
    description="Endpoint for track clustering predictions",
    auth_mode="key"  # or "aml_token"
)

ml_client.online_endpoints.begin_create_or_update(endpoint).result()
```

B. Create Deployment

```python
```

```python
from azure.ai.ml.entities import ManagedOnlineDeployment, CodeConfiguration

# Create deployment
deployment = ManagedOnlineDeployment(
    name="blue",
    endpoint_name="track-clustering-endpoint",
    model=registered_model,
    code_configuration=CodeConfiguration(
        code="./src",
        scoring_script="score.py"
    ),
    environment="track-clustering-env@latest",
    instance_type="Standard_DS2_v2",
    instance_count=1
)

ml_client.online_deployments.begin_create_or_update(deployment).result()

# Set traffic to 100%
endpoint.traffic = {"blue": 100}
ml_client.online_endpoints.begin_create_or_update(endpoint).result()
```

C. Test Endpoint

```python
```

```python
import json

# Prepare test data
test_data = {
    "data": [
        {
            "slope": 0.05,
            "curvature": 0.2,
            "offtrack_rate": 0.1,
            "density": 50,
            "stuck_rate": 0.02,
            "sos_rate": 0.01
        }
    ]
}

# Invoke endpoint
response = ml_client.online_endpoints.invoke(
    endpoint_name="track-clustering-endpoint",
    request_file=json.dumps(test_data)
)

print(response)
```

7️⃣ Setup Retraining Pipeline

A. Create Retraining Schedule

```python
```

```python
from azure.ai.ml.entities import JobSchedule, RecurrenceTrigger, RecurrencePattern

# Create schedule for weekly retraining
schedule = JobSchedule(
    name="weekly_retraining",
    trigger=RecurrenceTrigger(
        frequency="week",
        interval=1,
        schedule=RecurrencePattern(
            week_days=["Monday"],
            hours=[2],
            minutes=[0]
        )
    ),
    create_job=pipeline_job
)

ml_client.schedules.begin_create_or_update(schedule).result()
```

B. Create Event-Based Retraining (When New Data Arrives)

```python
from azure.ai.ml.entities import CronTrigger

# Monitor data storage for new files
# This requires Azure Event Grid setup

# Create event-triggered pipeline
# 1. Setup Event Grid on your storage account
# 2. Configure webhook to trigger pipeline

# Example using Azure Functions as intermediary:
# Storage Blob Trigger → Azure Function → Trigger ML Pipeline
```

C. Complete Retraining Script **pipelines/retrain_pipeline.py**

```python
```

```
"""
Automated Retraining Pipeline
Monitors for new data and triggers retraining
"""
from azure.ai.ml import MLClient, Input
from azure.identity import DefaultAzureCredential
from datetime import datetime
import os

def trigger_retraining(data_path: str, enable_tuning: bool = False):
    """
    Trigger retraining pipeline

    Args:
        data_path: Path to new data
        enable_tuning: Whether to run hyperparameter tuning
    """

    # Connect to workspace
    ml_client = MLClient(
        credential=DefaultAzureCredential(),
        subscription_id=os.getenv("AZURE_SUBSCRIPTION_ID"),
        resource_group_name="track-clustering-rg",
        workspace_name="track-clustering-ml"
    )

    # Get pipeline
    pipeline = ml_client.jobs.get("track_clustering_training_pipeline")

    # Create new run with timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    pipeline_job = training_pipeline(
        pipeline_data=Input(type="uri_folder", path=data_path),
        n_clusters=3,
        max_iter=300,
        enable_tuning=enable_tuning
    )

    # Submit pipeline
    job = ml_client.jobs.create_or_update(
        pipeline_job,
        experiment_name=f"track_clustering_retrain_{timestamp}"
```

```
    )

    print(f"Retraining pipeline submitted: {job.name}")

    return job

if __name__ == "__main__":
    # Trigger retraining
    trigger_retraining(
        data_path="azureml://datastores/workspaceblobstore/paths/new-data",
        enable_tuning=True
    )
```

D. Setup Model Comparison & Auto-Deploy

python

```
"""
Compare new model with production model
Auto-deploy if better
"""
from azure.ai.ml import MLClient
from azure.identity import DefaultAzureCredential

def compare_and_deploy(new_model_name: str, new_model_version: int):
    """
    Compare new model with production model
    Deploy if metrics are better
    """
    ml_client = MLClient(
        credential=DefaultAzureCredential(),
        subscription_id="YOUR_SUBSCRIPTION_ID",
        resource_group_name="track-clustering-rg",
        workspace_name="track-clustering-ml"
    )

    # Get production model metrics
    prod_model = ml_client.models.get(
        name="track-clustering-model",
        label="production"
    )

    # Get new model metrics
    new_model = ml_client.models.get(
        name=new_model_name,
        version=new_model_version
    )

    # Compare metrics (example: silhouette score)
    prod_silhouette = float(prod_model.tags.get("silhouette", 0))
    new_silhouette = float(new_model.tags.get("silhouette", 0))

    print(f"Production model silhouette: {prod_silhouette}")
    print(f"New model silhouette: {new_silhouette}")

    # Deploy if better
    if new_silhouette > prod_silhouette:
        print("New model is better! Deploying...")

        # Update deployment
```

```
    deployment = ml_client.online_deployments.get(
        name="blue",
        endpoint_name="track-clustering-endpoint"
    )

    deployment.model = f"{new_model_name}:{new_model_version}"

    ml_client.online_deployments.begin_create_or_update(deployment).result()

    # Update production label
    new_model.tags["environment"] = "production"
    ml_client.models.create_or_update(new_model)

    print("Deployment complete!")
else:
    print("New model is not better. Keeping production model.")
```

8️⃣ Monitoring & Maintenance

A. Enable Application Insights

```python
from azure.ai.ml.entities import ManagedOnlineEndpoint

# Update endpoint with App Insights
endpoint = ml_client.online_endpoints.get("track-clustering-endpoint")
endpoint.enable_app_insights = True

ml_client.online_endpoints.begin_create_or_update(endpoint).result()
```

B. Setup Alerts

Via Azure Portal:

1. Go to your ML Workspace
2. **Monitoring → Alerts → Create alert rule**
3. Set conditions:
   - **Metric**: Request latency > 1000ms
   - **Metric**: Failed requests > 5%

- **Metric**: CPU utilization > 80%

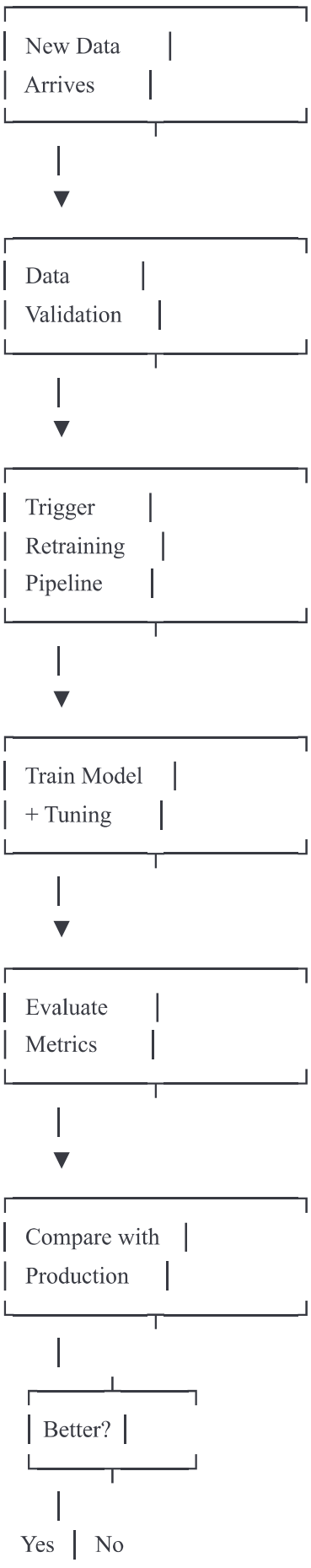
4. **Actions**: Email notification

C. Monitor Model Performance

```python
```

```
"""
Monitor deployed model performance
"""
from azure.ai.ml import MLClient
from azure.monitor.query import LogsQueryClient, LogsQueryStatus
from datetime import datetime, timedelta

def get_endpoint_metrics():
    """Get endpoint performance metrics"""

    ml_client = MLClient(...)
    logs_client = LogsQueryClient(credential=DefaultAzureCredential())

    # Query Application Insights
    query = """
    requests
    | where timestamp > ago(7d)
    | summarize
        RequestCount = count(),
        AvgDuration = avg(duration),
        FailureRate = countif(success == false) * 100.0 / count()
    | project RequestCount, AvgDuration, FailureRate
    """

    workspace_id = "YOUR_APP_INSIGHTS_WORKSPACE_ID"
    response = logs_client.query_workspace(
        workspace_id=workspace_id,
        query=query,
        timespan=timedelta(days=7)
    )

    if response.status == LogsQueryStatus.SUCCESS:
        for table in response.tables:
            for row in table.rows:
                print(f"Requests: {row[0]}")
                print(f"Avg Duration: {row[1]}ms")
                print(f"Failure Rate: {row[2]}%")
```

Complete MLOps Workflow Summary



📝 Quick Start Checklist

- [ ] Create Azure ML Workspace
- [ ] Upload training data
- [ ] Create conda environment
- [ ] Create compute cluster
- [ ] Prepare train.py and score.py
- [ ] Test training pipeline locally
- [ ] Submit training pipeline to Azure
- [ ] Register trained model
- [ ] Create online endpoint
- [ ] Deploy model to endpoint
- [ ] Test endpoint with sample data
- [ ] Setup retraining schedule
- [ ] Enable monitoring & alerts
- [ ] Document API usage

🔗 Useful Resources

- Azure ML Documentation
- Azure ML Python SDK v2
- MLOps Best Practices
- Azure ML Examples

Total Setup Time: 2-4 hours (first time)
Monthly Cost Estimate: $50-200 (depending on usage)
Deployment Type: Production-ready with auto-scaling