DSBA Platform MLOps Project: Enhanced Specification and Implementation Report

Dorian Boucher, Léo Lebuhotel, Karim Nabhan, Yasmina Nasr

Friday 11th April, 2025

Executive Summary

This document details the implementation of an enhanced MLOps workflow for the `dsba-platform`. The project successfully established a pipeline enabling model training, evaluation, persistent storage, API serving, containerization, and automated cloud deployment to Azure Container Instances (ACI). Key MLOps principles such as automation, reproducibility, and cloud-native deployment were demonstrably implemented, providing a practical foundation for managing the machine learning lifecycle, albeit with certain simplifications appropriate for the project's scope. The core achievement is a functional, cloud-deployed API capable of training models on user-provided data and serving predictions.

Project Goal and MLOps Context

The overarching goal was to evolve the `dsba-platform` into a system embodying modern MLOps practices. Referencing concepts from the course slides, the project focused on creating a reusable and automated infrastructure for managing ML models in a production-like context. Specifically, it aimed to showcase:

- **Automation:** Minimizing manual steps in training, packaging, and deployment.
- **Reproducibility:** Ensuring consistent environments and traceable artifacts through containerization and model/metadata storage.
- **Modularity:** Maintaining a structured codebase with distinct components (`src/dsba`, `src/api`, `src/cli`).
- **Scalability & Cloud Deployment:** Utilizing cloud services (ACI, Azure Storage) for deployment and persistence.

The implementation prioritized delivering a working end-to-end system quickly, aligning with agile principles of iterating towards value.

Implemented Workflow: From Data to Deployed API

The implemented workflow provides a streamlined path for users to train, manage, and utilize classification models:

1. **Model Training and Evaluation:** Users can trigger training via the API (`POST /train/`) or CLI (`dsba_cli train`), supplying a data source (URL or local path) and target column. The `run_training_pipeline` function in `src/dsba/model_training.py` handles

data ingestion, preprocessing (using functions from `src/dsba/preprocessing.py`), train-test splitting, XGBoost model fitting, and subsequent evaluation using `src/dsba/model_evaluation.py`. Performance metrics (accuracy, precision, recall, F1) are automatically calculated.

2. **Model Registry and Persistence:** Upon successful training and evaluation, the model artifact (`.pkl`) and detailed metadata (`.json`, including metrics and hyperparameters) are saved using `src/dsba/model_registry.py`. This module interacts with a designated storage location. For the deployed Azure environment, this location is an Azure File Share (`modelshare`) mounted into the container at `/app/models/registry`. This crucial step ensures model persistence across container restarts and decouples storage from compute. The decision to hardcode this path in `src/dsba/config.py` streamlined the deployment script by avoiding config file mounting complexities in ACI for this iteration.
3. **API Serving with FastAPI:** The core logic is exposed via a RESTful API (`src/api/api.py`). Key endpoints include:
   - `GET /models/`: List available model IDs from the registry.
   - `GET /models/{model_id}/metadata`: Retrieve detailed metadata and performance metrics.
   - `POST /train/`: Initiate the training/evaluation/saving workflow.
   - `POST /predict/`: Serve predictions for provided feature sets using a specified model.

   Pydantic models are used for robust request/response validation.
4. **Containerization via Docker:** The API application, along with its dependencies listed in `pyproject.toml`, is packaged into a Docker image using the instructions in `src/api/Dockerfile`. This standardizes the runtime environment, isolates dependencies, and ensures that the application runs identically locally and in the cloud, embodying the principle of reproducibility. The Dockerfile efficiently copies the source code, installs packages, and configures Uvicorn to serve the FastAPI application on port 8000.
5. **Automated Azure Deployment:** Deployment to Azure is handled by a comprehensive PowerShell script. This script automates several critical MLOps tasks:
   - Docker image build and push to Azure Container Registry (ACR).
   - Provisioning of necessary Azure infrastructure: Resource Group, Storage Account, and the vital `modelshare` Azure File Share for model persistence.
   - Deployment to Azure Container Instances (ACI), specifying compute resources (CPU, memory).
   - Secure retrieval and injection of ACR and Storage Account credentials into the ACI configuration.
   - Configuration of the volume mount, linking the Azure File Share (`modelshare`) directly to the container's internal model registry path (`/app/models/registry`). This is the core mechanism enabling stateful model management in the otherwise stateless ACI environment.
   - Assignment of a public IP address and DNS label for external accessibility.

   This script effectively acts as an Infrastructure-as-Code (IaC) component for this specific deployment target, ensuring consistency and reducing manual deployment errors.

Key Functionalities and Code Structure

The project delivers a cohesive set of functionalities accessible through different interfaces:

- **Core Library (`src/dsba/`)** Contains modules for:

  - Data Ingestion (`data_ingestion/`) supporting local files and URLs.
  - Preprocessing (`preprocessing.py`) including categorical encoding.
  - Model Training (`model_training.py`) orchestrating the main pipeline.
  - Model Evaluation (`model_evaluation.py`) providing performance metrics.
  - Model Registry (`model_registry.py`) for saving/loading models and metadata.
  - Configuration (`config.py`) defining the (hardcoded) model storage path.
- **API Layer (`src/api/`)** Defines the FastAPI app (`api.py`) and its Dockerfile (`Dockerfile`).
- **CLI Layer (`src/cli/`)** Provides a command-line tool (`dsba_cli.py`) for local interaction.
- **Testing (`tests/`)** Includes unit and integration tests (using pytest and pytest-mock) for API, CLI, registry, and training modules, ensuring code quality and correctness.
- **Deployment Artifacts:** The PowerShell deployment script serves as the primary deployment mechanism.

Limitations and Future Enhancements

While successfully meeting the core objectives, this implementation has scope for future improvement:

- **Configuration Management:** The model path is currently hardcoded. Implementing dynamic configuration loading (e.g., via mounted config files or environment variables populated from Azure Key Vault) would enhance flexibility.
- **Infrastructure as Code (IaC):** While the PowerShell script provides automation, migrating the Azure resource definitions to Terraform would offer more robust, declarative, and stateful infrastructure management.
- **CI/CD Integration:** Automating the build, test, and deployment process using a CI/CD platform (like GitHub Actions or Azure DevOps) would further streamline the workflow and reduce manual intervention.
- **Monitoring and Logging:** Current logging is basic. Integrating structured logging and dedicated monitoring tools (e.g., Azure Monitor, Application Insights) would provide deeper insights into application performance and model behavior in production.
- **Advanced Deployment Strategies:** Implementing strategies like canary or blue-green deployments would allow for safer model updates.
- **Client Library:** Developing the Python client library (`src/dsba/client.py`) would simplify interaction with the deployed API.

Conclusion

This project successfully demonstrates a practical MLOps workflow, integrating model training, evaluation, containerization, and automated cloud deployment. It provides a solid foundation adhering to key MLOps principles, offering a valuable toolset for managing simple classification models. The implemented solution highlights the benefits of automation and cloud infrastructure in streamlining the path from model development to production use. Future work can build upon this foundation to incorporate more advanced configuration management, IaC, CI/CD, and monitoring practices for a fully mature MLOps system.