Milestone 1: Web & Serverless Model Serving

MLOps Course - Module 2

Aligned Learning Objectives

- **CG1.LO1:** Describe lifecycle stages relevant to deployment and monitoring.
- **CG1.LO2:** Apply reproducibility and maintainability practices in deployment.
- **CG1.LO3:** Explain model-artifact interaction with serving APIs.
- **CG2.LO1:** Construct containerized ML environments and configure registries.

Why This Matters

Model deployment is where your ML work transitions from experimentation to real-world impact. Understanding different serving patterns-from traditional web services to serverless functions-is essential for making informed architectural decisions in production MLOps environments. This milestone bridges the gap between model development and operational deployment, teaching you to navigate trade-offs in latency, scalability, cost, and maintainability.

Assignment Overview

You will build and deploy a complete model serving solution that demonstrates your understanding of the ML lifecycle, artifact management, and deployment architecture trade-offs. This assignment includes:

1. **Local FastAPI Service:** You will create a reproducible FastAPI microservice that loads a trained scikit-learn model and exposes a prediction endpoint with proper schema validation.
2. **Cloud Run Deployment:** You will deploy your FastAPI service to Google Cloud Run with automatic HTTPS, demonstrating containerized deployment and serverless scaling.
3. **Serverless Function:** You will implement the same inference logic as a Google Cloud Function, comparing the architectural differences and lifecycle implications between web-based and pure serverless deployment.

This assignment builds foundational understanding of artifact loading, schema validation, lifecycle awareness, and reproducibility-all critical for production ML systems.

Deliverables

1. FastAPI Service (Local)

- `main.py` with FastAPI app exposing `/predict` endpoint
- Pydantic request/response models for schema validation
- Model artifact (`model.pkl`) with deterministic loading
- Reproducible environment specification (`requirements.txt`, `pyproject.toml`, or `poetry.lock`)
- README describing lifecycle position (input → model → API → consumer)

2. Cloud Run Deployment

- Deployed Cloud Run service URL (publicly accessible with HTTPS)
- GCP Artifact Registry image reference
- Evidence of successful HTTPS inference (screenshots or curl output)
- Brief analysis of cold start behavior and lifecycle implications

3. Serverless Function (GCP Functions)

- Cloud Function code implementing the same prediction logic
- Deployment configuration and logs
- Comparative report: **FastAPI container vs Cloud Function**
  - Lifecycle differences (stateful vs stateless)
  - Artifact loading strategies
  - Latency characteristics (cold starts, warm instances)
  - Reproducibility considerations

4. Documentation

- Comprehensive README with:
  - Setup and deployment instructions
  - API usage examples
  - Lifecycle stage explanations
  - Model-API interaction description
  - Comparison of deployment patterns

Rubric

Total Points: 10 (10% of final grade per course map)

Component	Points	Criteria
FastAPI Endpoint	2	Correct API implementation with Pydantic schemas, deterministic artifact loading, and reproducible environment
Cloud Run Deployment	2	Successful HTTPS deployment with proper registry workflow and working inference
Serverless Function	2	Correct GCP Function deployment with successful invocation
Lifecycle Understanding	2	Clear explanation of deployment stages, artifact management, and model-API interaction across both patterns
Comparative Analysis	1	Accurate comparison of FastAPI container vs Cloud Function (latency, statelessness, cold starts, reproducibility)
Documentation & Reproducibility	1	Clear instructions, reproducible setup, and well-organized code structure

---

Requirements & Constraints

- **Language:** Python only
- **Framework:** FastAPI for web service
- **Cloud Platform:** Google Cloud Platform (Cloud Run and Cloud Functions)
- **Model:** Scikit-learn or similar lightweight model
- **Registry:** GCP Artifact Registry required for Cloud Run

What Success Looks Like

A high-quality submission demonstrates:

1. **Correctness:** All endpoints work as specified with proper error handling
2. **Reproducibility:** Environment can be recreated from provided specifications
3. **Lifecycle Awareness:** Demonstrates understanding of deployment stages and monitoring touchpoints
4. **Architectural Understanding:** Clear articulation of trade-offs between deployment patterns
5. **Code Quality:** Clean, well-documented, and maintainable code

---

Getting Started

Follow these steps to complete this milestone:

1. **Train or obtain a scikit-learn model** (e.g., classifier or regressor) and save it as `model.pkl`
2. **Create a FastAPI application** with a `/predict` endpoint and Pydantic schemas
3. **Test your API locally** using `uvicorn main:app --reload` and curl/Postman
4. **Containerize your application** with a Dockerfile for Cloud Run deployment
5. **Push your container** to GCP Artifact Registry
6. **Deploy to Cloud Run** and verify HTTPS inference works
7. **Create a Cloud Function** implementing the same prediction logic
8. **Benchmark both deployments** measuring cold start and warm latency
9. **Write your comparative analysis** documenting the differences you observed

---

Challenge Extensions (Optional)

Consider these enhancements for deeper learning (not required for full credit):

1. Add typed error handling and lifecycle-stage annotations (e.g., where monitoring hooks would fit)
2. Implement lightweight caching in the Cloud Function to reduce cold start time
3. Configure Cloud Run concurrency settings and analyze cost/latency trade-offs
4. Add request validation and rate limiting to the FastAPI service
5. Include basic observability (structured logging, latency tracking)

Submission Requirements

Deadline: [To be announced]

Submit a GitHub repository containing:

- All source code (FastAPI app, Cloud Function code)
- Model artifact or instructions to generate it
- Environment specifications
- Deployment configurations
- Comprehensive README with deployment URLs and analysis

- Screenshots or logs demonstrating successful deployments

Tips for Success

- **Start Local:** Get your FastAPI service working locally before attempting cloud deployments
- **Test Incrementally:** Validate each deployment step before moving to the next-don't try to debug everything at once
- **Document as You Go:** Capture your observations about cold starts, latency, and behavior differences in real-time
- **Use Version Pinning:** Ensure reproducibility with exact dependency versions in your requirements file
- **Consider the Lifecycle:** Think about where monitoring, logging, and error handling would fit in each deployment pattern
- **Common Pitfall:** Don't hardcode credentials or API keys-use environment variables and GCP IAM
- **Common Pitfall:** Ensure your model artifact is properly versioned and deterministically loadable
- **Common Pitfall:** Test both cold start and warm instance behavior for accurate latency comparisons

---

Resources

- FastAPI Documentation
- Google Cloud Run Quickstart
- Google Cloud Functions Python Guide
- GCP Artifact Registry Documentation