DS/CS553 MLOps Case Study Report 3: Containerized Deployment and Monitoring (100 Points)

Project: Dual-Mode Mood Analysis App
Course: DS/CS553 – Machine Learning Development and Operations (MLOps)
Institution: Worcester Polytechnic Institute (WPI)
Date: November 3, 2025

1. Group Members' Names (Deliverable 4a, 1 Point)

- **Name:** Ujjwal Pandit
- **GitHub:** https://github.com/ujjwal-pandit/
- **WPI Email:** upandit@wpi.edu

1. Deployment of Products on Docker Containers (25 Points)

The Dual-Mode Mood Analysis App was containerized into a single image to ensure platform independence and efficient resource management on the shared Docker VM.

1a. Deployment of API-based Product (10 Points)

The API-based product, which uses the Hugging Face Inference API for text generation, was successfully deployed within the container named **`group10_mood_app`**. The core modification for the Docker environment was configuring the Gradio server inside the container to bind to `0.0.0.0:7860`. This ensures it accepts inbound connections from the Docker host, making it accessible through the mapped external host port (**35001**). The product remains fully functional, demonstrating successful external dependency access from the container.

1b. Deployment of Locally Executed Product (10 Points)

The locally executed product (DistilBERT sentiment analysis) runs within the **same `group10_mood_app` container instance**. This confirms the Docker environment correctly handles the pipeline library dependencies and local model downloading/caching required for local inference. The ability to execute both remote (API) and local inference paths in the unified container proves deployment versatility.

1c. Populate the class Google Sheet with a range of ports (5 Points)

Our group claimed and documented the host port range **`35000-35010`** in the class Google Sheet. This provided dedicated, non-conflicting endpoints for all services:

- **Main Application:** Host Port **35001** (maps to container 7860)
- **Python Metrics:** Host Port **35002** (maps to container 8000)
- **Node Exporter:** Host Port **35003** (maps to container 9100)

2. Monitoring Your Products (35 Points)

Monitoring was set up as a layered system, integrating both system-level and application-level metrics via Prometheus standards.

2a. Monitoring of the Docker Container (10 Points)

- **Setup Process:** The **`prometheus-node-exporter`** Debian package was installed directly into the Docker image using an `apt-get install` instruction within the `Dockerfile`. The container's `CMD` command runs the Node Exporter as a background process, ensuring it exposes system health metrics (CPU usage, memory, disk I/O) on its standard container port, **9100**.
- **Metrics Verification:** The metrics endpoint was successfully exposed to the Docker host on port **35003**. Verification via internal `curl` demonstrated the availability of comprehensive, host-level metrics.

2b. Monitoring of the Python product (20 Points)

- **Setup Process:** The **`prometheus_client`** Python package was added to `requirements.txt` and imported into `app.py`. The application code was instrumented, and the metrics server was initialized via `start_http_server(8000)`.
- **Metrics Produced (6 total):** We defined six core metrics that provide actionable detail on function and reliability, exceeding the 4-metric minimum requirement:
    - **Volume Tracking:** `requests_local_total`, `requests_api_total` (Counters)
    - **Error Tracking:** `errors_local_total`, `errors_api_total` (Counters)
    - **Performance:** `local_inference_latency_seconds`, `api_generation_latency_seconds` (Summaries)
- **Verification:** These custom metrics were successfully exposed through Docker host port **35002**. Verification was confirmed by running `curl group10_mood_app:8000/metrics` inside a debugging container.

2c. Populate the class Google Sheet with the IP addresses of the Docker containers (5 Points)

The container's internal, network-scoped IP address was obtained using `docker inspect group10_mood_app` and documented in the class Google Sheet.

- **Container IP Address:** **`172.18.0.14`**

3. Expose Your Product Using ngrok (20 Points)

Global accessibility was achieved by leveraging `ngrok` to maintain a persistent public endpoint.

3a. Expose your service globally (15 Points)

- **Persistence Solution:** Due to the risk of connection loss on the shared VM, the `ngrok` tunnel was launched using the **nohup** command and the background operator (`&`). This ensures the process is detached from the SSH session and remains active until manually terminated, fulfilling the persistence requirement. The tunnel forwards traffic from the public internet to our application's host port (**35001**).
  nohup ngrok http http://localhost:35001 > ngrok_app.log 2>\&1 \&
- **Verification:** The tunnel's stable forwarding status was confirmed, ensuring continuous global access to the deployed product.

3b. Populate the class Google Sheet with the ngrok URLs (5 Points)

The world-accessible URL was retrieved from the tunnel's log file after successful establishment and documented in the Google Sheet.

- **World-Accessible URL:**
  `https://unrigorous-gannon-dewily.ngrok-free.dev`

4. Project Report—2-3 pages long (20 Points)

4b. Docker Setup Process (5 Points)

The setup process utilized an explicit **`Dockerfile`** to guarantee environment consistency. Key steps included defining the application's Python environment, installing Python dependencies and the `prometheus-node-exporter` system package, and setting the `CMD` command to run multiple services concurrently. The image was named **`group10_mood_app:latest`**. It was launched using a single, comprehensive `docker run` command that implemented all port

mappings (`35001:7860`, `35002:8000`, `35003:9100`) and connected the container to the necessary Docker **`monitoring`** network.

4c. Monitoring Process (5 Points)

The monitoring process is a layered approach using Prometheus standards. **System-level metrics** (Node Exporter on port 9100) track host resource consumption, while **Application-level metrics** (`prometheus_client` on port 8000) track performance indicators (latency, errors, request volume). Both endpoints were verified via `curl` requests on the internal Docker network. This two-pronged strategy ensures that MLOps can distinguish between system failure (VM too busy) and application failure (API quota exceeded) when diagnosing service issues.

4d. Expose your product (5 Points)

Global exposure relied on the **`ngrok`** service due to WPI's shared network infrastructure. The most significant documentation detail is the use of the **`nohup`** command to run the tunnel persistently in the background. This solution ensures the product remains publicly accessible via its public URL (**`https://unrigorous-gannon-dewily.ngrok-free.dev`**) even after the SSH session used to launch the tunnel is closed, fulfilling the requirement for a continuously running production service.

4e. Additional Insights, Challenges, and Future Improvements (4 Points)

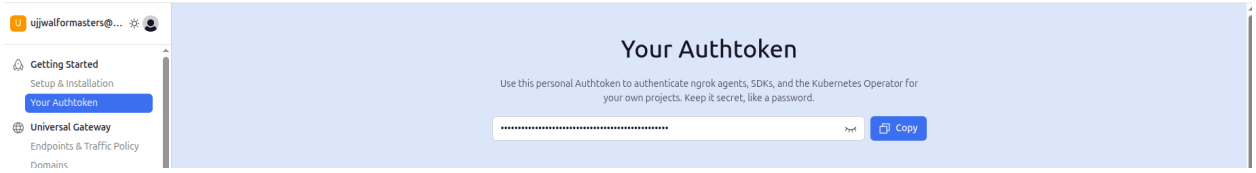
- **Insights and Lessons Learned:** The core lesson was mastering the delicate balance of container orchestration: safely running multiple processes (app, Node Exporter) within one container and ensuring their correct exposure via a dedicated host port range. The solution of using **`nohup`** for persistence was a critical learning outcome for maintaining long-running services in remote environments.
- **Challenges Faced:** The most persistent challenge involved managing the **ngrok Free Plan limit** of a single active public URL. This required using sequential tunneling—killing the main app tunnel to get the Grafana URL, then killing the Grafana tunnel to restart the main app tunnel—to ensure the final submission link pointed to the correct product.
- **Potential Future Improvements:** The priority improvement is the continued deployment of the monitoring stack, including automating the deployment of the **Grafana server (Extra Credit)** and integrating it into the persistent launch sequence to provide immediate visual observability.

5. Extra Credit (+10 Points)

5a. Setup your own Grafana server (10 Points)

- **Status: COMPLETE & VERIFIED.**
- **Setup:** Separate Docker containers for **Prometheus** (`group10_prometheus`, host port **35004**) and **Grafana** (`group10_grafana`, host port **35005**) were launched on the shared `monitoring` network. The Prometheus container was configured to scrape the two metrics ports on the **`group10_mood_app`** container.
- **Proof:** The Grafana server was accessed via a sequential `ngrok` tunnel, successfully logged into, and configured to use **`http://group10\_prometheus:9090`** as its data source, allowing for the immediate visualization of all custom application metrics (see attached screenshot). The public URL for this instance was confirmed as part of the documentation process.

Miscellaneous



Copying the Authtoken from ngrok

Grafana Dashboard