Machine Learning Engineering & AI

Bootcamp Capstone

Step 10: Design Your Deployment Solution Architecture

17 April 2025

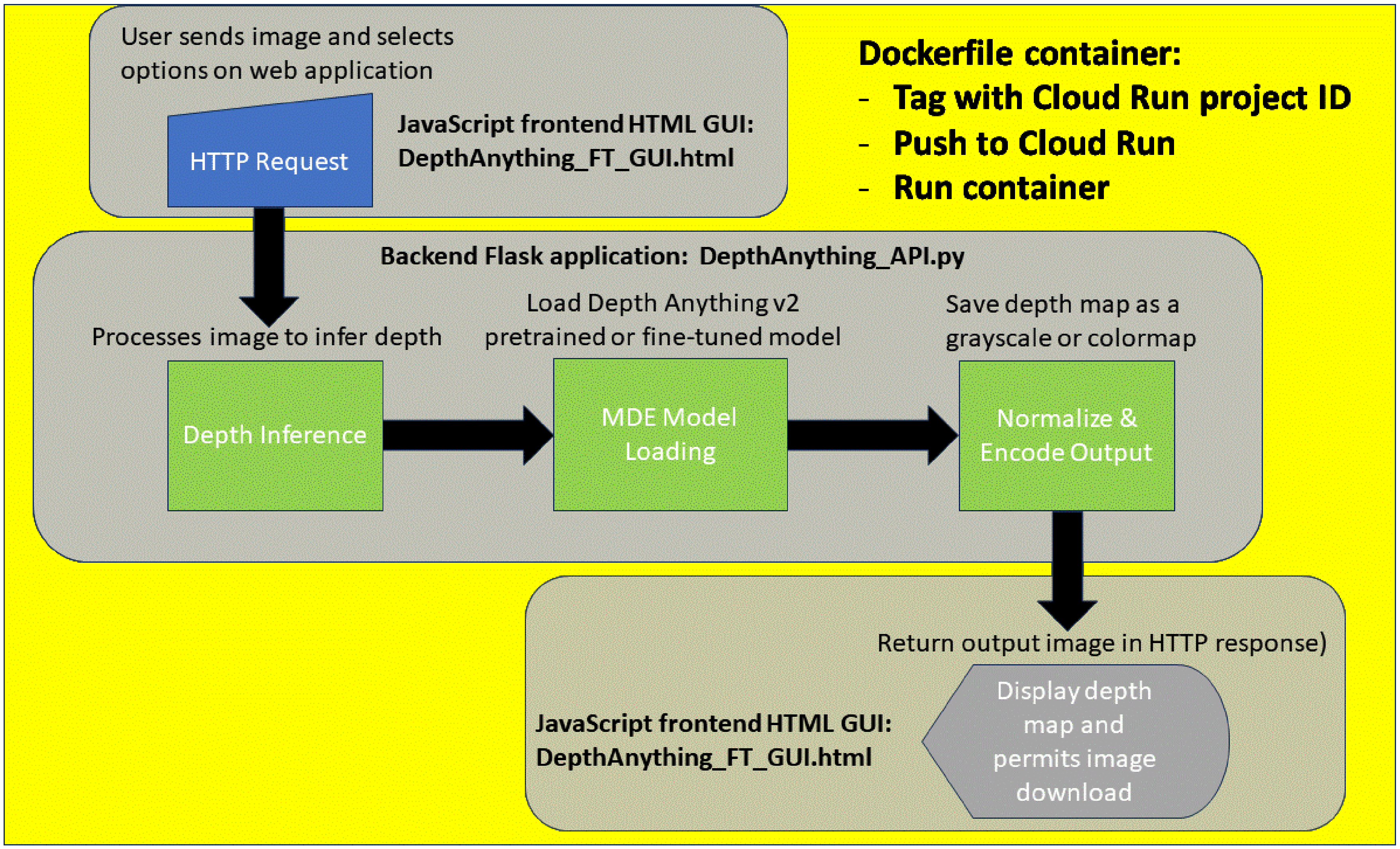
Deployment Architecture for a Web-Based Depth Estimation Application

1. Architecture Diagram

Below is a diagram representing the major components and their interactions:

```
+----------------------+
|  User’s Browser/GUI  |  <-- HTML/JS Interface
+----------+-----------+
           | HTTP(S) Requests
           v
+----------------------+
|  Flask API Backend   |  <-- Deployed as a Docker container on Cloud Run
|  (Inference Service) |
+----------+-----------+
           |
+----------v-----------+
|  Docker Container    |  <-- Contains application code, dependencies, model, etc.
|   (Cloud Run)        |
+----------+-----------+
           |
           v
+----------------------+
| Google Container     |
| Registry & Cloud Run |  <-- Manages deployment, scaling, and resource allocation
+----------------------+
           |
           v
+----------------------+
|  Cloud Storage /     |
| Artifact Registry    |  <-- Stores training data, logs, model artifacts
+----------------------+
```

Figure 1: Diagram of deployment architecture


User sends image and selects options on web application
HTTP Request
JavaScript frontend HTML GUI: DepthAnything_FT_GUI.html
Dockerfile container:
- Tag with Cloud Run project ID
- Push to Cloud Run
- Run container
Backend Flask application: DepthAnything_API.py
Processes image to infer depth
Depth Inference
Load Depth Anything v2 pretrained or fine-tuned model
MDE Model Loading
Save depth map as a grayscale or colormap
Normalize & Encode Output
Return output image in HTTP response)
JavaScript frontend HTML GUI: DepthAnything_FT_GUI.html
Display depth map and permits image download

Below is a diagram representing the end-to-end application flow:

```
User's Browser
      |
      v
[HTTP GET] / --> Flask serves DepthAnything_FT_GUI.html
      |
      v
User Interacts with GUI:
   - Uploads Image
   - Selects Feature & Representation
      |
      v
[HTTP POST] /infer  --> Flask API receives image and options
      |
      v
Flask API:
   - Loads Model
   - Processes Image to generate Depth Map
   - Returns Depth Map as PNG
      |
      v
User's Browser displays Depth Map and offers Download Option
```

---

2. Major Components of the System

- **Frontend GUI (HTML/JavaScript):**
  Provides an interface for users to upload images, select options (e.g., mirror, indoor art, outdoor art, grayscale vs. colormap), and view/download depth maps.
- **Flask API Backend:**
  Implements an inference endpoint (e.g., /infer) using Flask. It processes incoming HTTP requests, runs the MDE model, and returns the depth map as an image.
- **Docker Container:**
  The entire application (backend, frontend assets, and dependencies) is containerized using Docker. This container is what gets deployed on Cloud Run.

- **Google Cloud Run Service:**
  A managed serverless container environment that automatically scales based on incoming traffic and assigns dynamic port values via environment variables.
- **Google Container Registry (or Artifact Registry):**
  Stores the Docker image used by Cloud Run.
- **Cloud Storage / Artifact Repository:**
  Optionally used for persistent storage of training data, model artifacts, logs, and retraining outputs.
- **Monitoring and Logging Tools:**
  Cloud Monitoring and Cloud Logging are used to track performance, memory/CPU usage, errors, and other service metrics.

---

3. Inputs and Outputs

- **Inputs:**
  - **User-Uploaded Images:**
    Images submitted by users through the GUI for depth inference.
  - **Configuration Options:**
    User choices (for example, image representation type, feature option) are sent along with the image.
- **Outputs:**
  - **Depth Map:**
    A depth map image (grayscale or colormap) is returned to the user.
  - **Downloadable File:**
    The API sends the depth map as an attachment so that users can download it.
  - **Logs and Metrics:**
    Operational logs and performance metrics are generated for monitoring the service.

---

4. Data Storage and Flow

- **Data Storage:**

    - **Ephemeral Request Data:**
      Images sent for inference are processed on the fly within the Docker container.
    - **Persistent Artifacts:**
      Training data, retrained model checkpoints, and logs are stored in Google Cloud Storage and/or an artifact repository.

- **Data Flow:**

1. **User Interaction:**
The user's browser sends an HTTP POST (with the image and options) to the Flask API.

2. **Flask API Processing:**
The Flask API retrieves the image from the request, processes it using the MDE model, and generates a depth map.

3. **Response Streaming/Download:**
The depth map is either streamed as a response (using Flask's streaming techniques) or sent as a complete file download.

4. **Monitoring and Logging:**
Logs from inference and performance metrics are pushed to Cloud Logging and Cloud Monitoring.

- **Inter-Component Communication:**
  All communication is done using standard HTTP/HTTPS:
    - The GUI sends API requests to Cloud Run.
    - Cloud Run manages incoming requests via load balancers.
    - Data for retraining (if applicable) is moved between Cloud Storage and the training pipeline using services like Cloud Build, AI Platform, or custom scripts.

---

5. ML/DL Model Lifecycle

A. Retraining Frequency and Triggers

- **Retraining Schedule:**
  The model could be retrained at fixed intervals (e.g., monthly, quarterly) or retraining could be triggered when performance metrics (e.g., error rates or accuracy) indicate that the model's predictions have degraded.

B. Data for Retraining

- **Data Requirements:**
  Retraining a depth estimation model requires high-quality, annotated images with corresponding ground truth depth maps. Additional data—such as images from new environments or feedback from users—can further enhance the model.
- **Data Storage:**
  The data can be stored in Cloud Storage buckets. Use versioning and access controls to manage and secure the training dataset.

C. Evaluation of the Retrained Model

- **Validation Metrics:**
  Use evaluation metrics such as Absolute Relative Error (Abs_Rel), Squared Relative Error (Sq_Rel), RMSE, RMSE_log, log10 error, and SILog error to quantify performance.
- **Testing Environment:**
  Deploy the retrained model to a staging environment (or run batch tests) before promoting it to production. Use automated tests and A/B testing if possible.

D. Deployment of the Retrained Model

- **CI/CD Pipeline:**
  Automate the process using Cloud Build or GitHub Actions to build a new Docker image with the retrained model.
- **Rolling Updates:**
  Deploy the new image to Cloud Run as a new revision. Initially, an option may be to route only a fraction of the traffic to the new revision before switching over completely.

E. Storage of Model Artifacts

- **Checkpoint Management:**
  Store retrained model checkpoints (e.g., .pth files) in a dedicated Cloud Storage bucket or an artifact repository.
- **Versioning:**
  Use clear naming and versioning practices to enable identification and rolling back to previous versions if needed.

---

6. Monitoring and Debugging

- **Cloud Monitoring:**
  Use Google Cloud Monitoring to view CPU, memory, request latency, and error metrics. Set up custom dashboards to track specific metrics (e.g., inference time, error rates).
- **Cloud Logging:**
  Cloud Logging captures detailed logs from the Flask application. Structured logging can be used to flag errors and performance issues.
- **Alerting:**
  Configure alerts in Cloud Monitoring for out-of-memory errors, high latency, and other critical conditions.
- **Debugging Tools:**
  Use Cloud Trace or Cloud Debugger for deeper insights into performance bottlenecks or errors in the application.

---

7. Tools and Technologies

The following tools and technologies are involved in this system:

- **Programming Languages & Libraries:**
  - **Python:** Flask, PyTorch
  - **JavaScript/HTML:** for the frontend GUI
  - **OpenCV and Matplotlib:** for image processing and visualization
- **Containerization:**
  - **Docker:** Package the application, ensuring consistency across environments.
- **Cloud Platforms:**
  - **Google Cloud Run:** Managed container service for deploying the application
  - **Google Container Registry:** Repository for Docker images
  - **Google Cloud Storage:** Storage for training data, model artifacts, and logs
  - **Cloud Monitoring and Logging:** Tools for performance and error tracking
- **CI/CD:**
  - **Google Cloud Build / GitHub Actions:** Automate image building and deployment when models are retrained.

- **Version Control and Collaboration:**
  - **GitHub:** Source code hosting and versioning
  - **Artifact Registry:** (Optional) for model artifacts

---

8. Estimated Implementation Cost

A. Resource Costs

- **Cloud Run:**
  Costs depend on usage. For low-traffic applications, it might cost a few dollars a month. At constant, heavy usage with multiple CPUs (e.g., 2–4 vCPUs) and higher memory (e.g., 4–8 GiB), costs can range from $50 to $200 per month.
- **Storage:**
  Cloud Storage costs for model artifacts and training data are generally low unless there are very large datasets (often a few dollars per month). Since the application processes only one image per request and is expected to be used only for demonstration purposes, extremely large datasets are not expected.
- **CI/CD and Build Costs:**
  Cloud Build and other CI/CD tools typically fall under free tiers for modest usage, with costs increasing only with high-frequency or long-duration builds.

B. Time and Money Costs

- **Development Time:**
  The project might require several weeks (2–4 weeks) of initial development, including building the Flask backend, designing the GUI, containerization, and deployment pipelines.
- **Operational Overhead:**
  Cloud Run's managed environment significantly reduces ongoing maintenance tasks compared to managing own servers.

C. Cost Comparison

- For a start-up or proof-of-concept, the pay-per-use model of Cloud Run can be very cost-effective.
- As the user base scales, Cloud Run's auto-scaling can handle the load efficiently, and it may be necessary to monitor and adjust resource allocations (CPU, memory) as needed.

9. Conclusion

The deployment architecture of my fine-tuned Depth Anything v2 MDE model involves several integrated components—from the Flask API and JavaScript frontend to containerization with Docker and deployment on Google Cloud Run. The system not only handles inference (taking an image as input and delivering a depth map as output) but also ties into the broader ML lifecycle that includes data storage, model retraining, evaluation, and redeployment.

Key Takeaways:

- **Architecture:** A containerized Flask API on Cloud Run, triggered by a user-friendly web interface.
- **Data Flow and Storage:** Input images are processed on the fly, while training data and models are archived in Cloud Storage.
- **ML Model Lifecycle:** Retraining occurs based on scheduled intervals or performance triggers with corresponding evaluation and automated redeployment pipelines.
- **Monitoring and Costs:** Integrated logging and monitoring tools help manage performance and resource usage, while the serverless, on-demand nature of Cloud Run enables cost efficiency for variable workloads.
- **Toolset:** Docker, Flask, PyTorch, Google Cloud Run, Cloud Build, Cloud Storage, and Cloud Monitoring.

This architecture balances ease of deployment and scalability with ongoing maintenance, making it ideal for modern ML pipelines that require rapid iteration and robust production deployment.