# 🎨 Neural Style Transfer Web Application: DevOps-Driven MLOps Platform

> ***CSE 816: Software Production Engineering – Final Project***
> *A showcase of real-world DevOps and MLOps automation, modularity, and innovation.*

---

### TEAM MEMBERS

- **Deepanshu Saini MT2024039**
- **Aryan Rastogi MT2024026**

---

## 🚩 Project at a Glance

This project presents a **production-grade, end-to-end MLOps platform** for Neural Style Transfer (NST), where users can transform their photos into digital artwork using deep learning. The system is engineered for **modularity, automation, scalability, and continuous improvement**, leveraging a robust DevOps toolchain to automate the entire Software Development Life Cycle (SDLC).

---

### 🛠️ DevOps Toolchain & Workflow

- **Version Control:** Git & GitHub for all code, configuration, and infrastructure-as-code.
- **CI/CD Automation:** Jenkins pipelines triggered by GitHub webhooks for build, test, Docker image creation, and deployment.
- **Containerization:** Docker & Docker Compose for consistent, portable microservices.
- **Configuration Management:** Ansible playbooks and modular roles for Kubernetes and ELK stack provisioning.
- **Orchestration & Scaling:** Kubernetes (K8s) Deployments, Persistent Volumes, and (optionally) Horizontal Pod Autoscaler (HPA).
- **Monitoring & Logging:** ELK Stack (Elasticsearch, Logstash, Kibana) for centralized, real-time log analytics.

---

### 🧩 Application Architecture

- **Frontend:** Modern HTML/CSS/JS UI (Nginx) for model selection, image upload, live preview, stylization, and feedback.
- **Routing Service:** FastAPI API gateway for `/stylize` and `/feedback`, dynamic model routing, and feedback logging.
- **Inference Services:** Four independent FastAPI microservices for model-specific NST inference, supporting dynamic checkpoint loading.
- **Fine-Tuning Service:** Automated model retraining and versioning, driven by user feedback and curated data.
- **Persistent Storage:** Shared PVC for all images, models, weights, and feedback logs.

---

### 🔄 Automated DevOps Workflow

1. **Code Commit:** Developer pushes code to GitHub.
2. **Jenkins Pipeline:** Fetches code, runs tests, builds & pushes Docker images.
3. **Automated Deployment:** Jenkins/Ansible deploys to Kubernetes with rolling updates.
4. **Monitoring:** All logs shipped to ELK, visualized in Kibana dashboards.

---

### 🔒 Security & Advanced Features

- **Secure Storage:** (Recommended) Vault/Kubernetes Secrets for credentials.
- **Modular Ansible Roles:** Maintainable, reusable automation.
- **Kubernetes HPA:** (If implemented) Dynamic autoscaling of inference services.
- **Live Patching:** Zero-downtime updates via K8s rolling deployments.

---

### 💡 Innovation & Domain Focus

- **MLOps:** ML model deployment, feedback loop, and continuous learning with DevOps automation.
- **Feedback-Driven Retraining:** User feedback closes the ML loop for ongoing model improvement.
- **Dynamic Model Routing:** Seamless model upgrades and A/B testing via auto-discovered checkpoints.

---

### 🎯 Evaluation Criteria Mapping

| Criteria | Implementation Highlights |
|---|---|
| **Working Project** | Fully functional, multi-service app with automated CI/CD, containerization, and K8s deployment. |
| **Advanced** | Modular Ansible roles, (optional) Vault, K8s HPA, live patching. |
| **Innovation** | Feedback-driven retraining, dynamic routing/versioning, MLOps focus. |
| **Domain-Specific** | MLOps: ML deployment, feedback loop, continuous learning, and DevOps automation. |

---

### 🚀 Getting Started

- **Local Dev:**
  `docker-compose up --build` → http://localhost:8080
- **Kubernetes:**
  Deploy via Ansible playbooks, monitor with `kubectl` and Kibana.

---

**This project demonstrates a real-world, production-ready MLOps platform, showcasing the power of DevOps automation, modular microservices, and continuous ML improvement.**

---

# 🎨 Neural Style Transfer Web Application

**End-to-End DevOps, ML, and Architecture Report**

---

## 1. Application Overview

### 🚀 Executive Summary

The Neural Style Transfer (NST) platform empowers users to transform their photos into works of art by applying the styles of famous paintings or custom styles. The system is designed for **scalability, modularity, and continuous improvement**—leveraging user feedback to fine-tune models and deliver ever-better results.

**Key Features:**

- Upload any image and stylize it with one of several deep learning models.
- Download the stylized result instantly.
- Submit feedback to help improve the models.
- Robust, production-grade backend with CI/CD, monitoring, and logging.

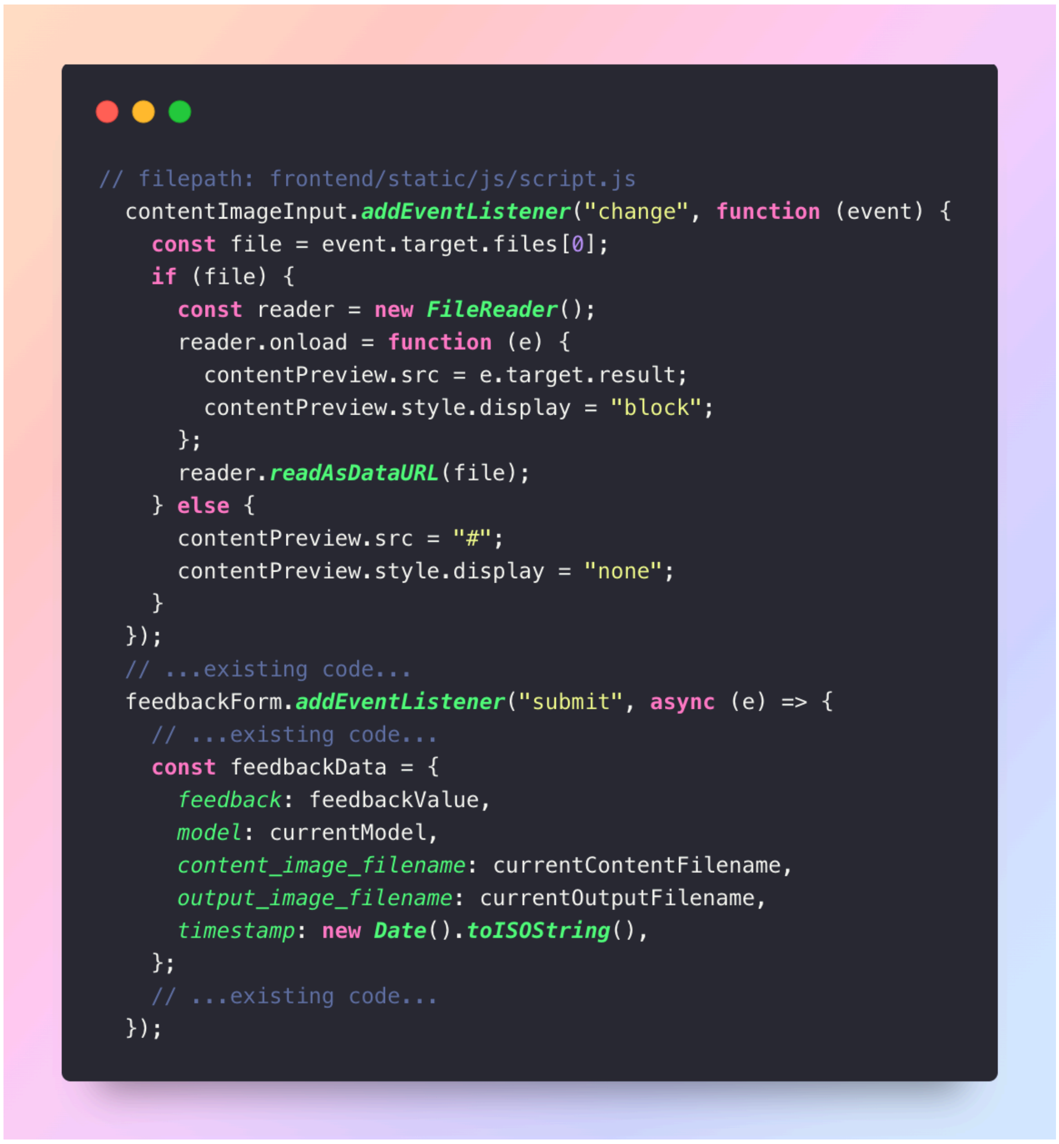
---

## 🏗️ System Architecture

### 🖼️ Front-end ( `frontend/` )

- **UI:** `frontend/index.html`
    - Modern, responsive HTML5 interface.
    - Model selector, image upload, live preview, stylized result, and feedback form. **Screenshot: NST Web Application Frontend**
- **Styling:** `frontend/static/css/style.css`
    - Clean, mobile-friendly design with subtle shadows, rounded corners, and intuitive layout.
- **Client-side Logic:** `frontend/static/js/script.js`
    - Handles image preview, AJAX requests for stylization and feedback, result display, and download.
    - Smart error handling and user guidance.

**Innovative Snippet: Live Image Preview and Feedback Integration**

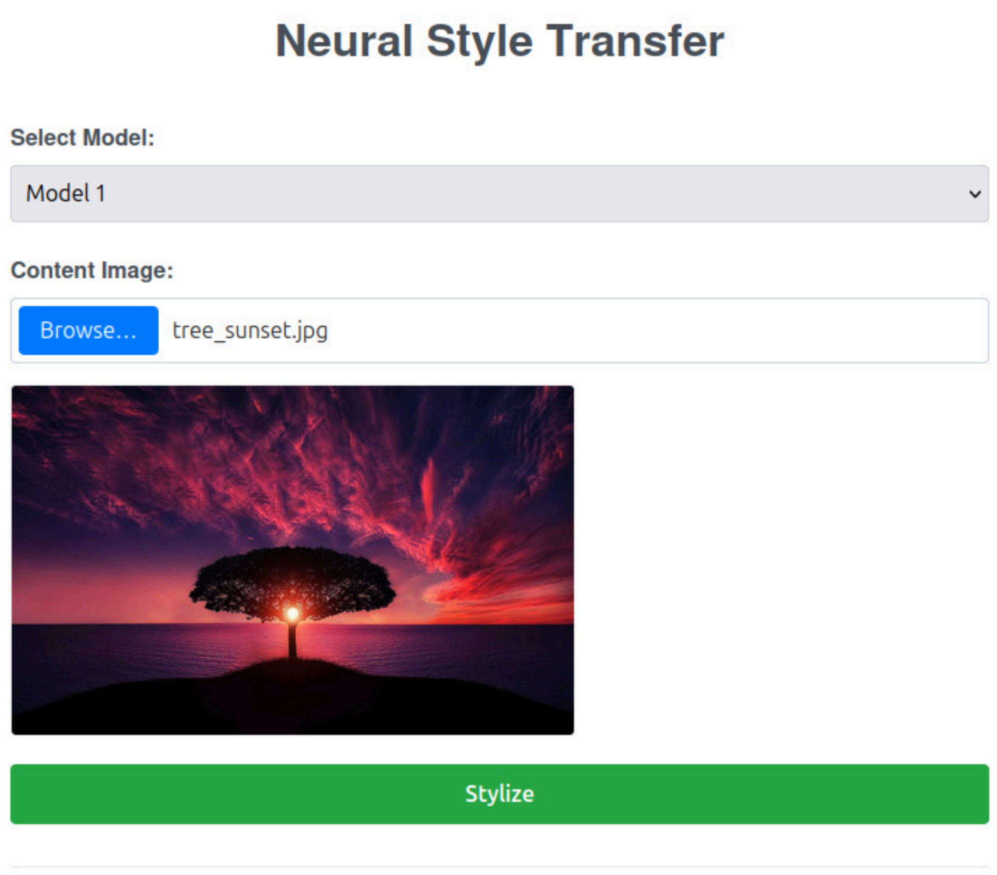


```
// filepath: frontend/static/js/script.js
  contentImageInput.addEventListener("change", function (event) {
    const file = event.target.files[0];
    if (file) {
      const reader = new FileReader();
      reader.onload = function (e) {
        contentPreview.src = e.target.result;
        contentPreview.style.display = "block";
      };
      reader.readAsDataURL(file);
    } else {
      contentPreview.src = "#";
      contentPreview.style.display = "none";
    }
  });
  // ...existing code...
  feedbackForm.addEventListener("submit", async (e) => {
    // ...existing code...
    const feedbackData = {
      feedback: feedbackValue,
      model: currentModel,
      content_image_filename: currentContentFilename,
      output_image_filename: currentOutputFilename,
      timestamp: new Date().toISOString(),
    };
    // ...existing code...
  });
```

**General Concept: Front-End Web Development**

- The front-end is the user-facing part of the application, responsible for providing an intuitive and interactive experience.
- Technologies like HTML, CSS, and JavaScript are used to structure content, style the presentation, and handle user interactions.
- Modern front-end development emphasizes responsiveness, accessibility, and performance.

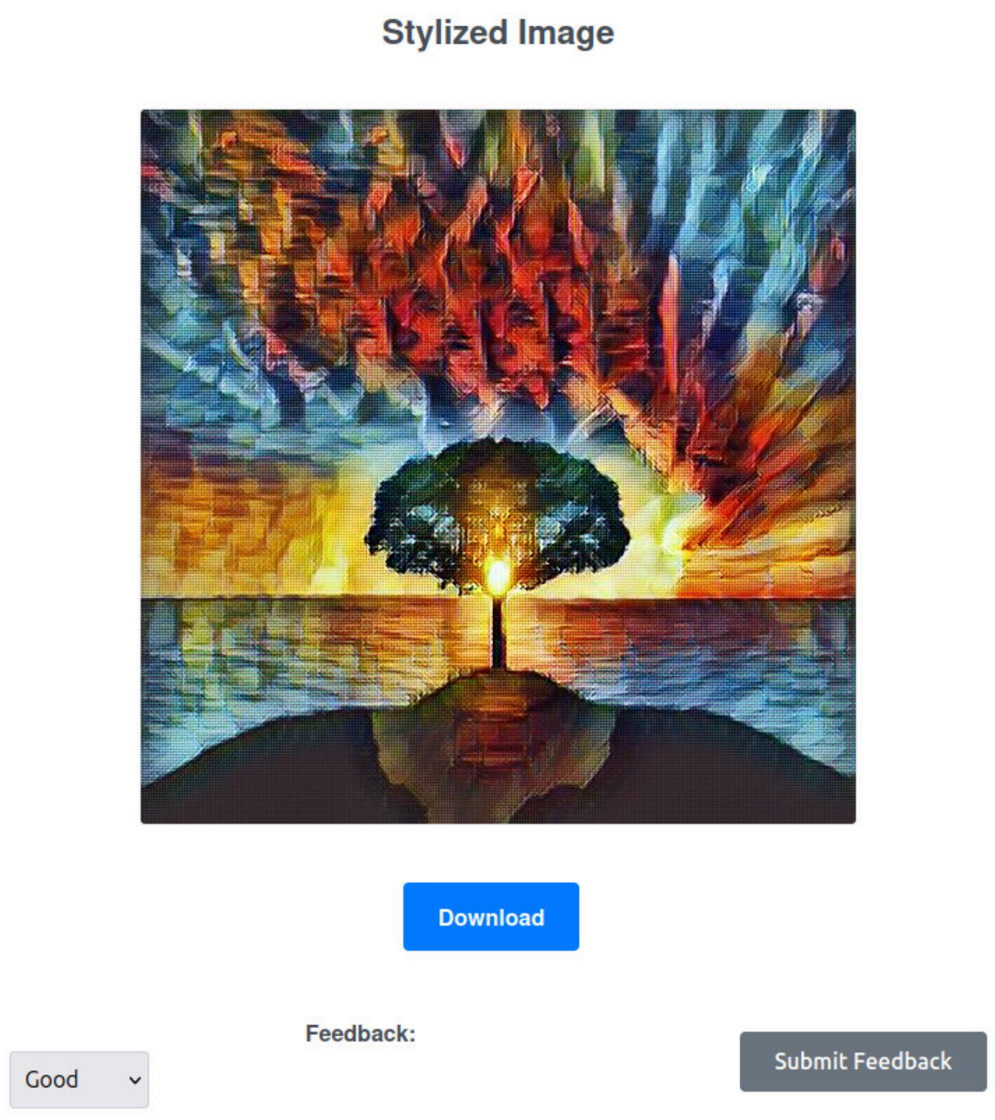
**Project Application: Front-End**



Neural Style Transfer

Select Model: Model 1

Content Image: tree_sunset.jpg



- The front-end of the NST application is built using HTML, CSS, and JavaScript.
- Nginx serves these static assets, providing a fast and efficient delivery mechanism.
- JavaScript handles asynchronous communication with the backend services, enabling a dynamic user experience.

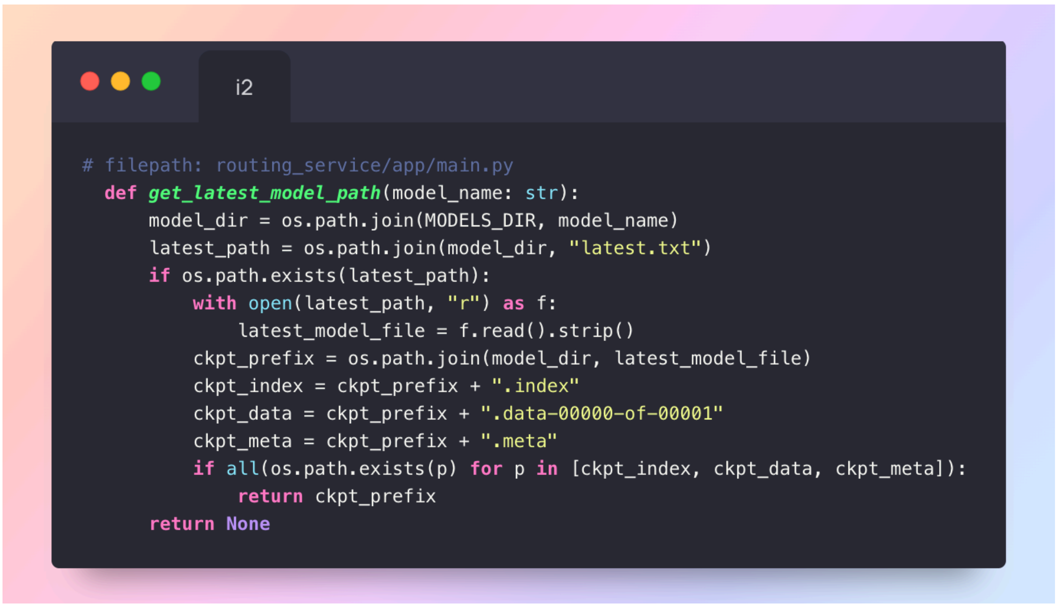
- **Web Server:** `frontend/nginx.conf`
  - Nginx serves static assets, handles routing, and can be extended for caching and compression.

🔀 **Routing Service (** `routing_service/app/main.py` **)**

- **API Gateway:**
  - Receives `/stylize` POST: saves input, resolves latest model checkpoint, forwards to the correct inference microservice, saves and returns stylized image.

- Receives `/feedback` POST: appends feedback JSON to `persistent_storage/feedback.jsonl` .
- Handles errors gracefully, logs all steps, and manages persistent storage directories.

**Innovative Snippet: Dynamic Model Routing and Checkpoint Resolution**



```python
# filepath: routing_service/app/main.py
  def get_latest_model_path(model_name: str):
      model_dir = os.path.join(MODELS_DIR, model_name)
      latest_path = os.path.join(model_dir, "latest.txt")
      if os.path.exists(latest_path):
          with open(latest_path, "r") as f:
              latest_model_file = f.read().strip()
          ckpt_prefix = os.path.join(model_dir, latest_model_file)
          ckpt_index = ckpt_prefix + ".index"
          ckpt_data = ckpt_prefix + ".data-00000-of-00001"
          ckpt_meta = ckpt_prefix + ".meta"
          if all(os.path.exists(p) for p in [ckpt_index, ckpt_data, ckpt_meta]):
              return ckpt_prefix
      return None
```

**General Concept: API Gateways**

- An API gateway acts as a single entry point for all client requests, decoupling the front-end from the backend services.
- It handles routing, authentication, authorization, rate limiting, and other cross-cutting concerns.
- API gateways improve security, simplify client development, and enable easier evolution of backend services.

**Project Application: Routing Service**

- The Routing Service acts as an API gateway for the NST application.
- It receives requests from the front-end, routes them to the appropriate inference service, and returns the results.
- This design allows for easy addition or removal of inference services without affecting the front-end.

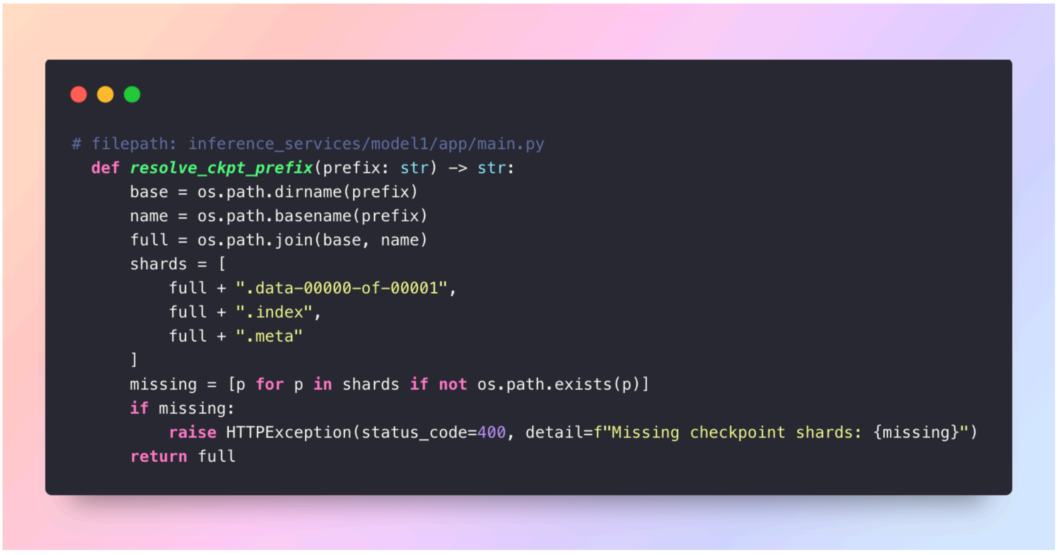
- **Smart Model Routing:**
  - Dynamically discovers the latest model checkpoint for each style model.
  - Forwards requests to the correct inference service using Kubernetes service DNS.

🤖 **Inference Microservices ( `inference_services/model*/` )**

- **Each Model (1–4):**
  - FastAPI app ( `main.py` , etc.).
  - `/infer` POST: receives content image and model checkpoint path, runs style transfer, returns base64-encoded stylized image and filename.
  - Uses model-specific inference logic (e.g., `model1_inference.py` ).

- Containerized for isolated, scalable deployment.

**Innovative Snippet: Robust TensorFlow Checkpoint Loading and Inference**



```python
# filepath: inference_services/model1/app/main.py
  def resolve_ckpt_prefix(prefix: str) -> str:
      base = os.path.dirname(prefix)
      name = os.path.basename(prefix)
      full = os.path.join(base, name)
      shards = [
          full + ".data-00000-of-00001",
          full + ".index",
          full + ".meta"
      ]
      missing = [p for p in shards if not os.path.exists(p)]
      if missing:
          raise HTTPException(status_code=400, detail=f"Missing checkpoint shards: {missing}")
      return full
```

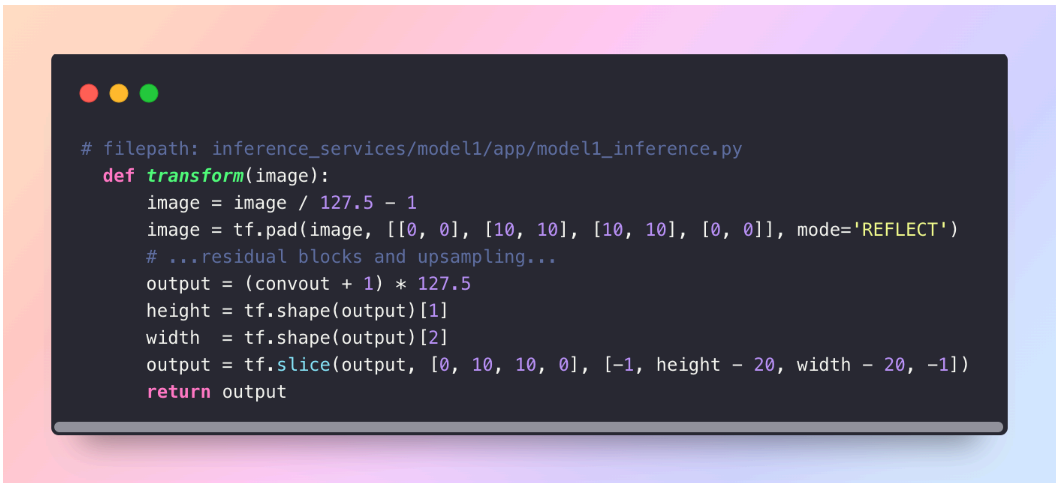
**General Concept: Microservices**

- Microservices are a software development approach where an application is structured as a collection of small, autonomous services, modeled around a business domain.
- Each service is independently deployable, scalable, and maintainable.
- Microservices enable faster development cycles, improved fault isolation, and greater technology diversity.

**Project Application: Inference Microservices**

- The NST application uses microservices for each style model.
- This allows each model to be deployed and scaled independently, optimizing resource utilization.
- It also enables easier experimentation with new models without affecting the entire application.
- **Model Architecture:**
    - Deep residual convolutional neural networks, trained for fast style transfer.
    - TensorFlow 1.x compatible, with custom layers and instance normalization.

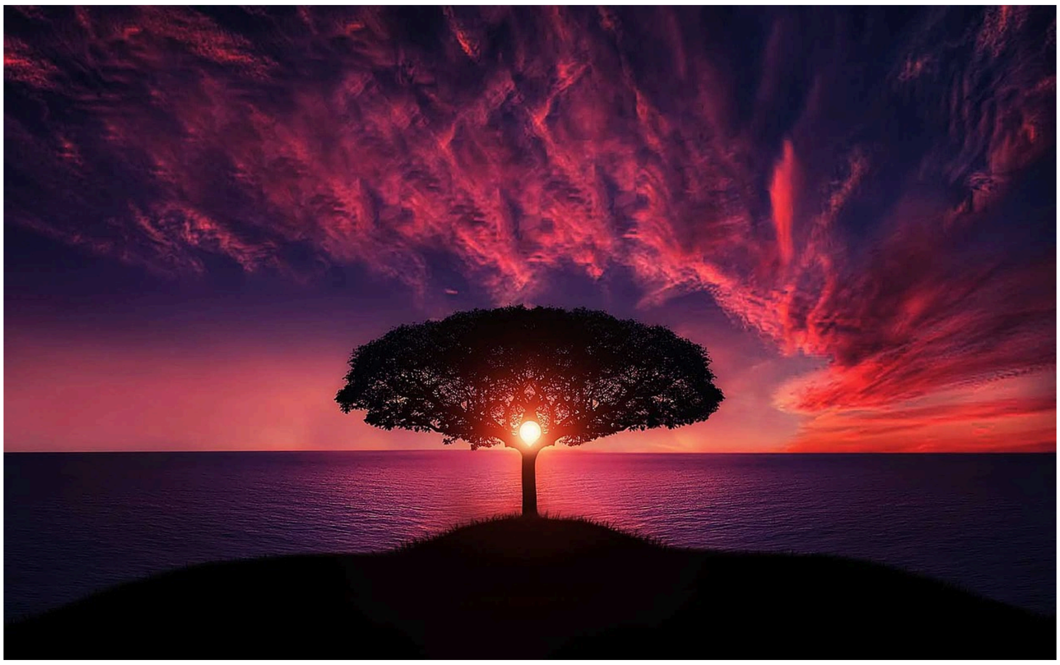
**Innovative Snippet: Efficient Image Transformation Network**



```python
# filepath: inference_services/model1/app/model1_inference.py
  def transform(image):
      image = image / 127.5 - 1
      image = tf.pad(image, [[0, 0], [10, 10], [10, 10], [0, 0]], mode='REFLECT')
      # ...residual blocks and upsampling...
      output = (convout + 1) * 127.5
      height = tf.shape(output)[1]
      width  = tf.shape(output)[2]
      output = tf.slice(output, [0, 10, 10, 0], [-1, height - 20, width - 20, -1])
      return output
```

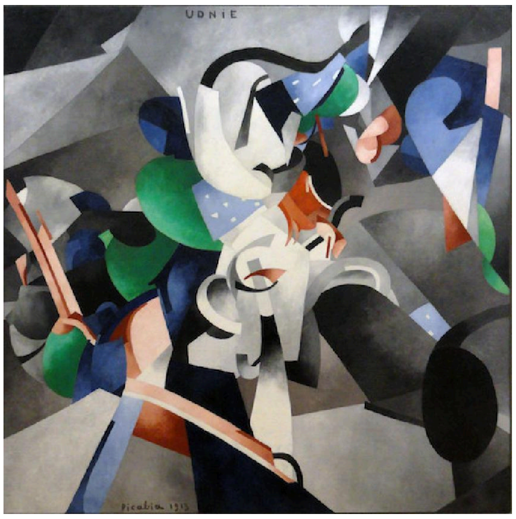
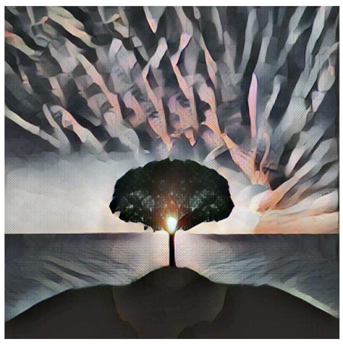
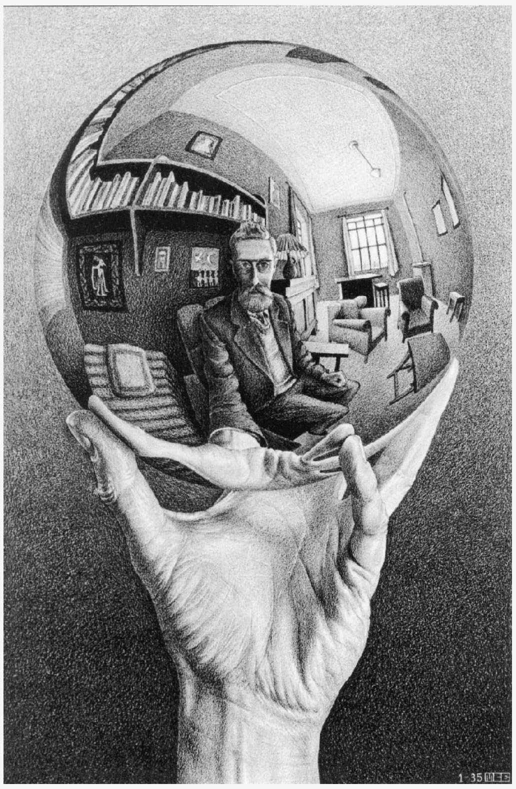
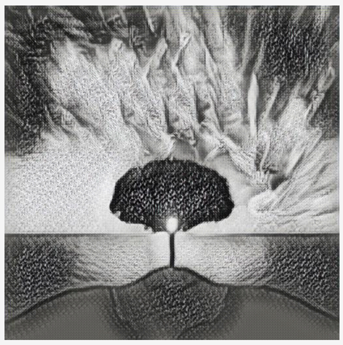
**Inference and Results**

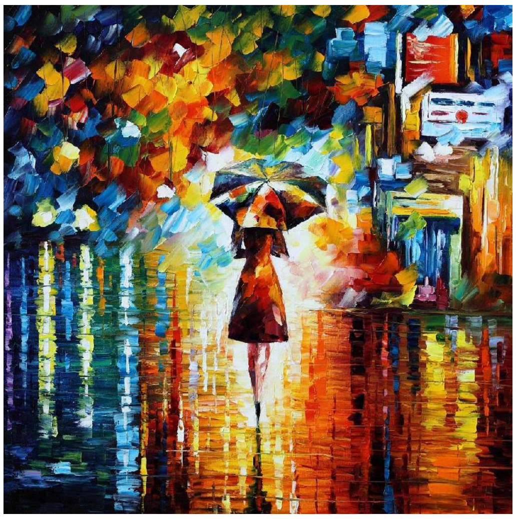
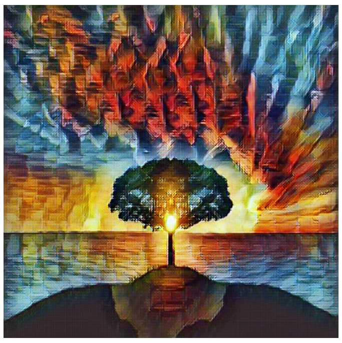
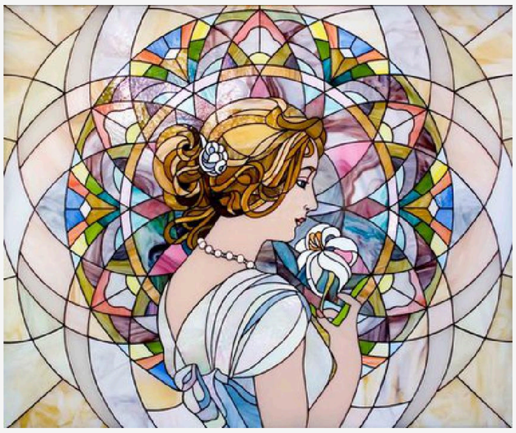
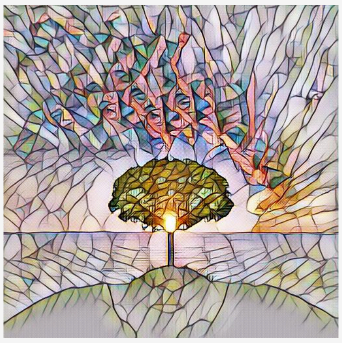
**Input Image**



**Stylized outputs by the models**

| Model | Style Image | Output Image |
|---|---|---|

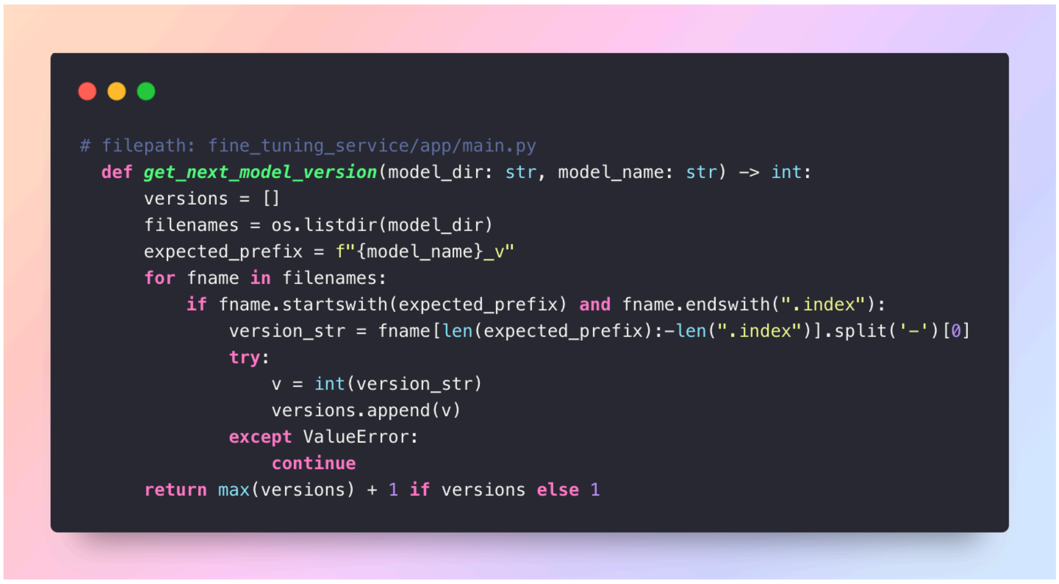
|  |  |  |
|---|---|---|
| Model 1 |  |  |
| Model 2 |  |  |

| | | |
|---|---|---|
| Model 3 |  |  |
| Model 4 |  |  |

🛠 **Fine-Tuning Service ( `fine_tuning_service/app/` )**

- **Feedback Loop:**
    - Consumes `persistent_storage/feedback.jsonl` for user feedback.
    - Training logic in `main.py` and `training_utils.py` .
    - Accepts fine-tuning requests (model, epochs, batch size, weights, etc.), validates data, launches background training, manages model versioning and checkpointing.
    - Updates `latest.txt` in model directories for routing/inference services.

**Innovative Snippet: Automated Model Versioning and Checkpoint Management**



```python
# filepath: fine_tuning_service/app/main.py
  def get_next_model_version(model_dir: str, model_name: str) -> int:
      versions = []
      filenames = os.listdir(model_dir)
      expected_prefix = f"{model_name}_v"
      for fname in filenames:
          if fname.startswith(expected_prefix) and fname.endswith(".index"):
              version_str = fname[len(expected_prefix):-len(".index")].split('-')[0]
              try:
                  v = int(version_str)
                  versions.append(v)
              except ValueError:
                  continue
      return max(versions) + 1 if versions else 1
```

**General Concept: Continuous Learning and Fine-Tuning**

- Machine learning models often require continuous learning and fine-tuning to maintain accuracy and adapt to new data.
- Fine-tuning involves training a pre-trained model on a smaller, task-specific dataset to improve its performance on that specific task.
- This approach is more efficient than training a model from scratch and can lead to better results.

**Project Application: Fine-Tuning Service**

- The Fine-Tuning Service enables continuous improvement of the NST models.
- User feedback is used to curate training data, which is then used to fine-tune the models.
- This feedback loop ensures that the models are constantly adapting to user preferences and producing high-quality stylized images.

- **Robust Validation:**
  - Ensures all images are valid RGB, style image is present, and VGG weights are available before training.

### 🗄️ Data Storage

- **Image Volumes:**
  - `persistent_storage/input_images/` : uploaded content images.
  - `persistent_storage/output_images/` : stylized outputs.
- **Model Checkpoints:**
  - `persistent_storage/models/model*/` : versioned TensorFlow checkpoints, `latest.txt` , and style images.
- **VGG Weights:**
  - `persistent_storage/vgg19.npz` : required for perceptual loss in training.
- **Feedback Log:**

- `persistent_storage/feedback.jsonl` : newline-delimited JSON, append-only, for auditability and ML curation.

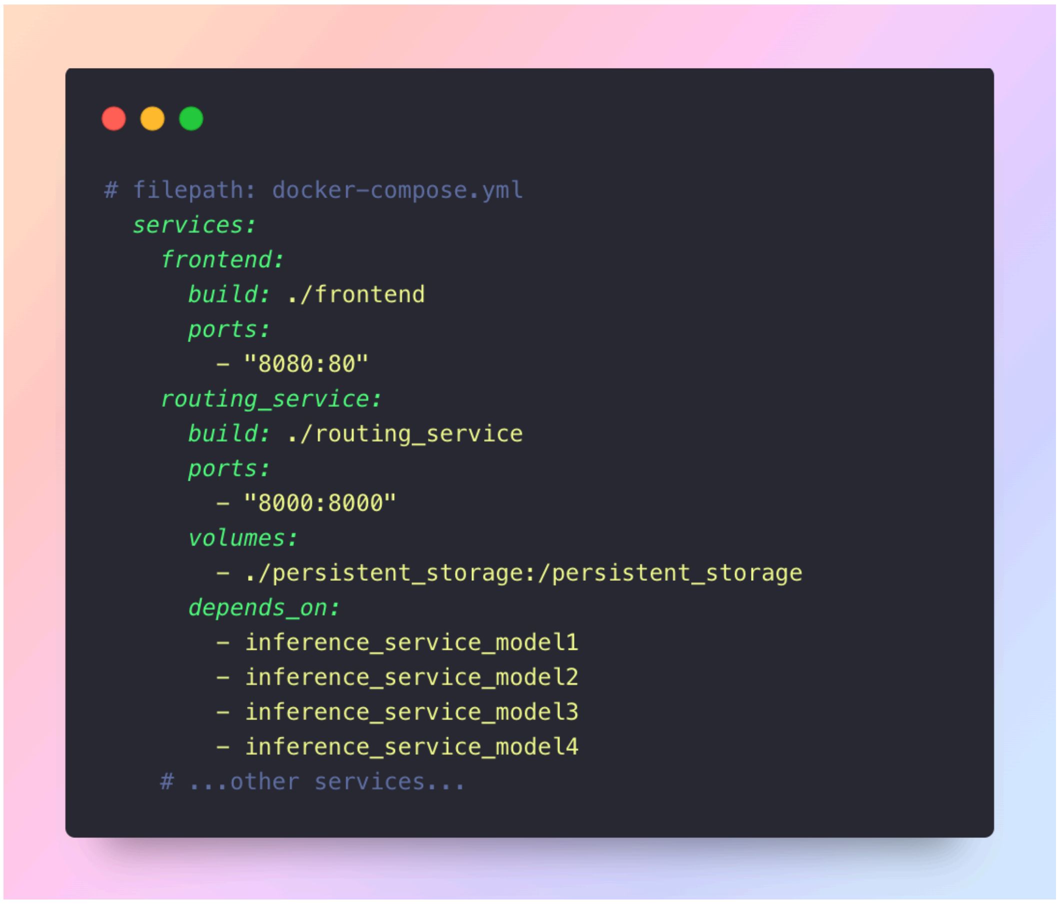
---

## 2. DevOps & CI/CD

### 🐳 Docker

- **General Concept: Containerization with Docker**
  - Imagine you're packing up all the ingredients and instructions needed to bake a specific cake. Docker is like creating that perfect, self-contained package. It bundles your application, its code, runtime, system tools, libraries, and settings into a single unit called a container. This container will run the same way, no matter where you put it – your computer, a friend's laptop, or a big cloud server. This eliminates the "it works on my machine" problem.
- **Base ML Image:** `docker/base-ml.Dockerfile`
  - Python 3.9, installs ML dependencies, used as a base for inference/fine-tuning services.
- **Service Containers:**
  - Each service has its own Dockerfile (e.g., `inference_services/model1/Dockerfile` , `fine_tuning_service/Dockerfile` , `frontend/Dockerfile` ).
- **Compose for Local Dev:** `docker-compose.yml`
  - Defines all services, networks, and volumes for local multi-container orchestration.
  - Hot-reload and persistent storage for rapid iteration.

**Innovative Snippet: Multi-Service Docker Compose for Local ML Development**



```yaml
# filepath: docker-compose.yml
  services:
    frontend:
      build: ./frontend
      ports:
        - "8080:80"
    routing_service:
      build: ./routing_service
      ports:
        - "8000:8000"
      volumes:
        - ./persistent_storage:/persistent_storage
      depends_on:
        - inference_service_model1
        - inference_service_model2
        - inference_service_model3
        - inference_service_model4
    # ...other services...
```

**Running the Services via Docker Compose (Step-by-Step)**

1. **Install Docker and Docker Compose:**
   - Think of Docker as the engine that runs your application packages (containers). Docker Compose is a tool that lets you define and manage multiple containers as a single application.
   - **On macOS:** Download Docker Desktop from the official Docker website and follow the installation instructions. Docker Compose is included.
   - **On Windows:** Download Docker Desktop from the official Docker website and follow the installation instructions. Docker Compose is included. Ensure that you enable WSL 2 during installation if prompted.
   - **On Linux:** Follow the instructions on the Docker website for your specific distribution to install Docker Engine and Docker Compose separately.
2. **Navigate to the Project Root:**
   - Open a terminal (command prompt or PowerShell on Windows) and use the `cd` command to navigate to the root directory of the NST project. This is the folder that contains the `docker-compose.yml` file.
   - Example: `cd /Users/aryanrastogi/college/spe/NST_Major/neural-style-transfer`
3. **Start the Services:**

    - Run the following command to start all the services defined in `docker-compose.yml`:

      ```
      docker-compose up --build
      ```

        - `docker-compose up`: This command tells Docker Compose to start the services defined in the `docker-compose.yml` file.
        - `--build`: This flag tells Docker Compose to build the Docker images for any services that have a `build` section in the `docker-compose.yml` file. This is necessary the first time you run the application or when you make changes to the Dockerfiles.

4. **Monitor the Startup:**

    - Observe the terminal output to monitor the startup process of each service. Docker Compose will display logs from each container as they start.
    - Wait until you see messages indicating that each service has started successfully (e.g., "Uvicorn running on ...").

5. **Access the Application:**

    - Once all services are running, access the application by opening your web browser and navigating to `http://localhost:8080`. This will direct you to the frontend service.
    - `localhost` refers to your own computer. `8080` is the port number that the frontend service is configured to listen on (as defined in `docker-compose.yml`).

6. **Access the Services Directly (for Debugging):**

    - You can access individual services directly for debugging or testing purposes:

        - Frontend: `http://localhost:8080`
        - Routing Service: `http://localhost:8000`
        - Inference Model 1: `http://localhost:8001`
        - Inference Model 2: `http://localhost:8002`
        - Inference Model 3: `http://localhost:8003`
        - Inference Model 4: `http://localhost:8004`
        - Fine-Tuning Service: `http://localhost:8005`

7. **Stop the Services:**

    - To stop the services, run the following command in the same terminal:

      ```
      docker-compose down
      ```

        - This command will stop and remove the containers, networks, and volumes created by `docker-compose up`.
        - It's a clean way to shut down the application and free up resources.

## Kubernetes

- **General Concept: Container Orchestration with Kubernetes**

  - Kubernetes (often shortened to K8s) is like a conductor for an orchestra of containers. It automates the deployment, scaling, and management of your containerized applications. Think of it as a platform that makes sure your application is always running as you intended, even if individual containers fail or need to be scaled up to handle more traffic. Kubernetes handles tasks like:
    - **Scheduling:** Deciding where to run your containers.
    - **Scaling:** Automatically increasing or decreasing the number of containers based on demand.
    - **Self-healing:** Restarting failed containers and replacing them automatically.
    - **Service discovery:** Allowing containers to find and communicate with each other.
    - **Rolling updates:** Updating your application without downtime.

- **Deployments:**

  - `kubernetes/deployments/` : Deploys each service with resource requests/limits, volume mounts, and health checks.
  - **In Detail:** Kubernetes Deployments are declarative specifications for how to create and update instances of your application. They ensure that a specified number of pod replicas are running at any given time. If a pod fails, the Deployment automatically replaces it. In our NST app, each microservice (Frontend, Routing, Inference Models, Fine-Tuning) is deployed as a separate Deployment.

- **Services:**

  - `kubernetes/services/` : Exposes each deployment as a ClusterIP service for internal communication.
  - **In Detail:** Kubernetes Services provide a stable IP address and DNS name for accessing a set of Pods. They act as a load balancer, distributing traffic across the healthy Pods in a Deployment. We use ClusterIP Services for internal communication between our microservices.

- **Ingress:**

  - `kubernetes/ingress/ingress.yaml` : (empty, but intended for HTTP(S) routing, TLS termination, and path-based routing).
  - **In Detail:** An Ingress exposes HTTP and HTTPS routes from outside the cluster to Services within the cluster. It acts as a reverse proxy and load balancer. While the file is currently empty, it's intended to handle TLS termination (HTTPS) and route traffic to the Frontend and Routing services based on URL paths.

- **Persistent Volumes:**

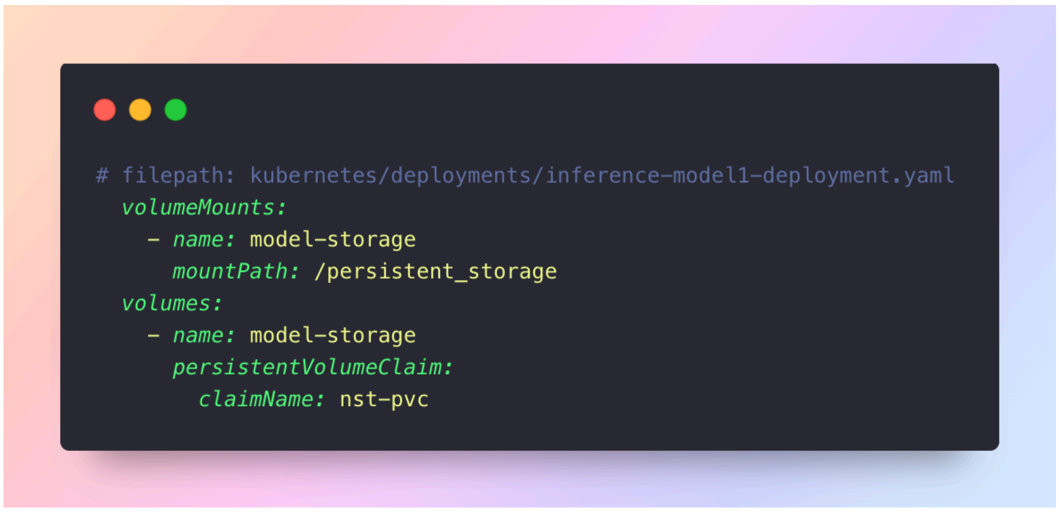
  - `kubernetes/persistent-volumes/pv.yaml` , `pvc.yaml` : Shared storage for models, images, and feedback.
  - **In Detail:** Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) provide a way to manage persistent storage in a Kubernetes cluster. PVs are cluster-wide resources that represent physical storage, while PVCs are requests for storage by users. Our NST app uses a PVC ( `nst-pvc` ) to request storage for models, images, and feedback data, ensuring that this data persists even if pods are restarted or rescheduled.

- **NetworkPolicies:**

  - **In Detail:** Kubernetes NetworkPolicies control traffic flow at the IP address or port level (OSI layer 3 or 4). They allow you to specify rules for which pods can communicate with each other. Implementing NetworkPolicies is a crucial security best practice for limiting the blast radius of potential security breaches.

- **Monitoring & Logging:**
  - **ELK:** `kubernetes/elk/` : Elasticsearch, Logstash, Kibana, Filebeat manifests for centralized logging.
  - **Prometheus/Grafana:** `kubernetes/monitoring/` : Placeholders for metrics stack and dashboards.

**Innovative Snippet: Persistent Volume Claim for Model and Data Sharing**



```yaml
# filepath: kubernetes/deployments/inference-model1-deployment.yaml
  volumeMounts:
    - name: model-storage
      mountPath: /persistent_storage
  volumes:
    - name: model-storage
      persistentVolumeClaim:
        claimName: nst-pvc
```

**Running the Application using Kubernetes (Step-by-Step)**

1. **Install kubectl:**
   - `kubectl` is the command-line tool for interacting with your Kubernetes cluster.
   - **On macOS:** `brew install kubectl` (if you have Homebrew installed)
   - **On Windows:** Download the `kubectl` binary from the Kubernetes website and add it to your PATH.
   - **On Linux:** Follow the instructions on the Kubernetes website for your specific distribution.
2. **Configure kubectl:**
   - `kubectl` needs to be configured to connect to your Kubernetes cluster. This typically involves setting the `KUBECONFIG` environment variable or placing a configuration file in `~/.kube/config` .
   - The specific configuration steps depend on your Kubernetes provider (e.g., Minikube, Google Kubernetes Engine, Amazon EKS, Azure Kubernetes Service). Follow the instructions provided by your provider.
3. **Install Ansible:**
   - Ansible is an automation tool that we'll use to deploy the application to Kubernetes.
   - **On macOS:** `brew install ansible` (if you have Homebrew installed)
   - **On Windows:** Install Python and pip, then run `pip install ansible` .

- **On Linux:** Use your distribution's package manager (e.g., `apt install ansible` on Debian/Ubuntu).

4. **Configure Ansible:**

   - Ansible needs to be configured to connect to your Kubernetes cluster. This typically involves setting up SSH access to a node in the cluster or using a Kubernetes API token.
   - The specific configuration steps depend on your Kubernetes provider.

5. **Navigate to the Ansible Playbooks Directory:**

   - Open a terminal and navigate to the `ansible/playbooks` directory in the NST project.

6. **Run the Umbrella Playbook:**

   - Execute the `umbrella-playbook.yml` playbook to deploy the entire application stack:

     ```
     ansible-playbook umbrella-playbook.yml -i inventory.ini
     ```

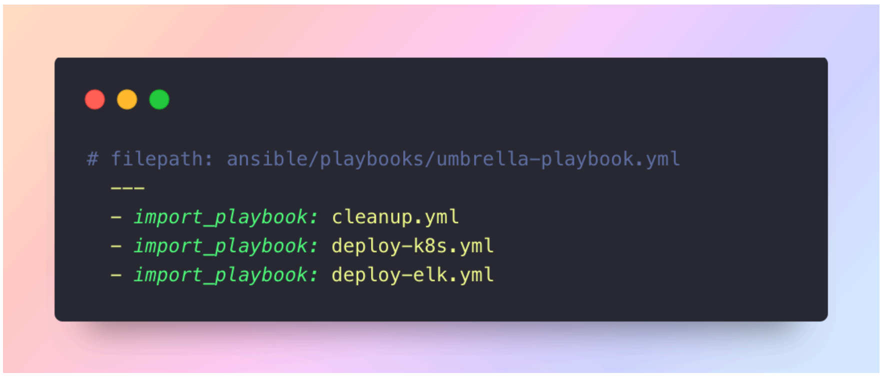
     - `ansible-playbook` : This command tells Ansible to run the specified playbook.
     - `umbrella-playbook.yml` : This is the main playbook that orchestrates the deployment of the application.
     - `-i inventory.ini` : This flag specifies the inventory file, which contains information about the target Kubernetes cluster. You may need to create or modify this file based on your cluster setup.

7. **Monitor the Deployment:**

   - Observe the terminal output to monitor the deployment process. Ansible will display the status of each task as it executes.
   - Pay attention to any errors or warnings that may occur.

8. **Verify the Deployment:**

   - Once the playbook has completed, verify that all the services are running correctly by checking the status of the deployments and pods:



```yaml
# filepath: ansible/playbooks/umbrella-playbook.yml
  ---
  - import_playbook: cleanup.yml
  - import_playbook: deploy-k8s.yml
  - import_playbook: deploy-elk.yml
```

        - `kubectl get deployments` : This command lists all the deployments in the Kubernetes cluster.
        - `kubectl get pods` : This command lists all the pods in the Kubernetes cluster.
        - Ensure that all deployments have the desired number of replicas and that all pods are in the `Running` state.

9. **Access the Application:**
    - To access the application, you need to determine the external IP address or hostname of the Ingress controller. The exact steps depend on your Kubernetes provider.
    - **Minikube:** `minikube service list`
    - **Cloud Providers (GKE, EKS, AKS):** Check the Ingress resource in your cloud provider's console or use `kubectl get ingress` .
    - Once you have the external IP address or hostname, open it in your web browser to access the application.
10. **Access Kibana:**
    - Similarly, determine the external IP address or hostname of the Kibana service and access the Kibana dashboard in your web browser.

## ⚙️ Ansible

- **General Concept: Infrastructure as Code (IaC) with Ansible**
    - Infrastructure as Code (IaC) is the practice of managing and provisioning infrastructure through machine-readable definition files, rather than manual configuration.
    - Ansible is an automation tool that simplifies IaC by using a declarative language to define the desired state of the infrastructure.
    - IaC enables repeatable, consistent, and auditable infrastructure deployments.
- **Inventory & Group Vars:**
    - `ansible/inventory.ini` (not shown, but implied), `ansible/group_vars/all.yml` : Cluster and registry config.
- **Playbooks:**
    - `ansible/playbooks/cleanup.yml` : Removes all K8s resources for a clean slate.
    - `ansible/playbooks/deploy-k8s.yml` : Deploys core app to K8s.
    - `ansible/playbooks/deploy-elk.yml` : Deploys ELK stack.
    - `ansible/playbooks/umbrella-playbook.yml` : Orchestrates full deployment.

    **Innovative Snippet: One-Click Full Stack Deployment**

    ```
    # filepath: ansible/playbooks/umbrella-playbook.yml
    ---
    - import_playbook: cleanup.yml
    - import_playbook: deploy-k8s.yml
    - import_playbook: deploy-elk.yml
    ```

## 🔁 CI/CD Pipeline

- **General Concept: Continuous Integration and Continuous Delivery (CI/CD)**
  - CI/CD is a set of practices that automate the build, test, and deployment of software changes.
  - Continuous Integration (CI) ensures that every code change is automatically built and tested, catching issues early.
  - Continuous Delivery (CD) automates the release of tested code to production-like environments, enabling rapid and reliable deployments.

**Main Pipeline: NST_App**

The main Jenkins pipeline orchestrates the entire DevOps lifecycle for the NST platform. Here's how it works, step by step:

1. **Checkout:**
   The pipeline begins by securely fetching the latest code from the GitHub repository. This ensures that every build operates on the most recent version of the codebase, enabling traceability and reproducibility.

2. **Lint & Tests (Parallelized):**
   Code quality is paramount. The pipeline runs Python linting ( `flake8` ) and frontend linting ( `npm run lint` ) to enforce style and catch common errors. Simultaneously, it executes unit tests for all Python services ( `pytest` ) and validates all Kubernetes YAML manifests ( `yamllint` ). Running these checks in parallel speeds up feedback for developers and helps maintain a healthy codebase.

3. **Build & Push Images:**
   Once the code passes all checks, Docker images are built for each microservice (frontend, routing, inference, fine-tuning). These images are then pushed to DockerHub using secure credentials, ensuring that deployments always use the latest, tested containers.

4. **Deploy to Staging:**
   The pipeline applies all Kubernetes manifests to the staging cluster using `kubectl` . This step leverages Jenkins credentials for secure access and ensures that the application is deployed in a production-like environment for further validation.

5. **Smoke Test:**
   To verify that the deployment was successful, the pipeline performs a health check by querying the `/health` endpoint via the Kubernetes Ingress. This quick test ensures that the core services are up and running before proceeding.

6. **Feedback-Driven Fine-Tuning:**
   The pipeline automatically monitors user feedback by counting the number of `"bad"` feedback entries in `feedback.jsonl` . If the count exceeds a configurable threshold (e.g., 5), it triggers the dedicated FineTuneModels pipeline to retrain the models. This closes the ML loop, ensuring that the models continuously improve based on real user input.

7. **Deploy to Production (Optional):**
   For maximum safety, production deployment is gated behind a parameter. When enabled, the pipeline runs an Ansible playbook to deploy the latest images to the production environment, ensuring a controlled and auditable release process.

8. **Cleanup:**
   After every run, the pipeline prunes unused Docker images to keep the build agents clean and

efficient.

**Why this design?**
This pipeline is crafted for speed, safety, and continuous improvement. By parallelizing checks, automating deployments, and integrating feedback-driven retraining, it enables rapid iteration while maintaining high reliability. The use of secure credentials, parameterized production deploys, and automated cleanups reflects best practices in modern DevOps.

---

**Fine-Tuning Pipeline: FineTuneModels**

The FineTuneModels Jenkins job is a specialized, parameterized pipeline designed to trigger model fine-tuning on demand—either automatically (from the main pipeline) or manually by developers. Here's how it works:

1. **Checkout:**
   The pipeline starts by fetching the latest code from GitHub, ensuring that the fine-tuning process uses the most up-to-date training logic and configurations.

2. **Trigger Fine-Tuning Service:**
   The heart of this pipeline is a POST request to the fine-tuning service's `/finetune` endpoint. All relevant parameters—such as model name, epochs, batch size, image size, learning rate, style/content weights, and the service URL—are passed dynamically. This enables flexible, reproducible, and auditable fine-tuning runs.

3. **Publish Trigger Info:**
   For transparency and traceability, the pipeline logs which model was requested for fine-tuning and which build triggered the process. This is invaluable for audit trails and debugging.

4. **Post Actions:**
   - **On Success:**
     If fine-tuning succeeds, the pipeline can (optionally) re-trigger the main NST_App pipeline. This ensures that the newly fine-tuned model is immediately integrated into the deployment workflow, closing the loop from feedback to improved production models.
   - **On Failure:**
     If anything goes wrong, an email notification is sent to the relevant stakeholders, including a link to the build logs for rapid troubleshooting.
   - **Always:**
     Regardless of outcome, the pipeline logs completion, ensuring that every run is accounted for.

**Why this design?**
This pipeline empowers both automation and developer-driven experimentation. By parameterizing every aspect of the fine-tuning process, it supports rapid iteration, reproducibility, and easy integration with the main CI/CD flow. The ability to automatically re-trigger the main pipeline ensures that improvements are quickly propagated to users, while robust notifications and logging keep the team informed.

---

**In summary:**
These Jenkins pipelines embody the principles of modern MLOps: automation, feedback-driven improvement, traceability, and developer empowerment. They ensure that every code change, model update, and user feedback cycle is seamlessly integrated into a robust, production-grade workflow—delivering better models, faster, and with confidence.

---

## 3. Network & Security

- **TLS Termination:**
  - Should be handled at Ingress (see `kubernetes/ingress/ingress.yaml`), with Let's Encrypt or custom certificates.
- **NetworkPolicies & RBAC:**
  - Only allow routing service to talk to inference pods.

---

## 4. Scalability & Resilience

- **Horizontal Pod Autoscaler:**
  - (Not present, but recommended for inference/routing services. Example: scale up when CPU > 70%.)
- **Load Balancing:**
  - K8s Services provide internal load balancing; Ingress for external.
- **Service Discovery:**
  - K8s DNS for service-to-service communication (e.g., `http://inference-service-model1:8000`).
- **Data Persistence:**
  - All critical data (models, images, feedback) stored on shared PVC, ensuring no data loss on pod restarts.
- **Fault Tolerance:**
  - Pod restarts, readiness/liveness probes, persistent storage, and stateless service design.

---

## 5. Appendix

### 📁 Full Directory Tree (Summary)

```
.
├── ansible/
│   ├── group_vars/all.yml
│   └── playbooks/{cleanup.yml,deploy-elk.yml,deploy-k8s.yml,umbrella-playbook.yml}
├── docker/
│   └── base-ml.Dockerfile
├── docker-compose.yml
├── fine_tuning_service/
│   ├── Dockerfile
│   ├── app/{main.py,training_utils.py}
│   └── requirements.txt
├── frontend/
│   ├── Dockerfile
│   ├── index.html
│   ├── nginx.conf
│   └── static/{css/style.css,js/script.js}
├── inference_services/
│   └── model{1,2,3,4}/
│       ├── Dockerfile
│       ├── app/{main.py,model*_inference.py}
│       └── requirements.txt
├── kubernetes/
│   ├── deployments/
│   ├── elk/
│   ├── ingress/
│   ├── monitoring/
│   ├── persistent-volumes/
│   └── services/
├── persistent_storage/
│   ├── feedback.jsonl
│   ├── input_images/
│   ├── models/
│   │   └── model*/{latest.txt,*.index,*.meta,*.data-00000-of-00001,style.jpg}
│   ├── output_images/
│   └── vgg19.npz
└── routing_service/
    ├── Dockerfile
    └── app/main.py
```

### 🧩 Microservice Summary Table

| Name | Port | Dockerfile Path | Endpoints | Description |
|---|---|---|---|---|
| **Frontend** | 80 | frontend/Dockerfile | `/` | Static UI via Nginx |
| **Routing Service** | 8000 | routing_service/Dockerfile | `/stylize`, `/feedback` | API gateway, feedback logger |
| **Inference Model 1** | 8000 | inference_services/model1/Dockerfile | `/infer` | NST inference (model 1) |
| **Inference Model 2** | 8000 | inference_services/model2/Dockerfile | `/infer` | NST inference (model 2) |
| **Inference Model 3** | 8000 | inference_services/model3/Dockerfile | `/infer` | NST inference (model 3) |
| **Inference Model 4** | 8000 | inference_services/model4/Dockerfile | `/infer` | NST inference (model 4) |
| **Fine-Tuning Service** | 8000 | fine_tuning_service/Dockerfile | `/finetune`, `/health` | Model training/fine-tuning |

---

**By incorporating these elements, the report will not only be more comprehensive and visually appealing, but also more accessible and engaging for a wide range of readers—from engineers to stakeholders and newcomers to DevOps/ML.**