**Question 2**

**Question 2.1**

**Serialize & Package Your Model**

1. **Export a single file that the service can load at startup**
   * **TorchScript**
   * scripted\_model = torch.jit.script(my\_model)
   * scripted\_model.save("model.pt")
   * **Or state\_dict**
   * torch.save(my\_model.state\_dict(), "model\_state.pth")
2. **Implement the Inference Service**
   * Pick a minimal web framework (e.g., FastAPI or Flask).
   * On startup, load the serialized model into eval() mode.
   * Define the preprocessing (image → tensor) and postprocessing (logits → JSON) steps.
   * Expose a /predict endpoint that:
     1. Accepts input (e.g., file upload or JSON).
     2. Runs inference with:
     3. with torch.no\_grad():
     4. output = model(x)
     5. Returns the results as JSON.
3. **Containerize**
   * Write a Dockerfile that installs exact dependencies (torch, Uvicorn or Gunicorn, Pillow, etc.) and copies in the code + model file.
   * Build the image:
   * docker build -t my-model-api:latest .
4. **Local QA Test**
   * Run the container on a local port:
   * docker run --rm -p 8000:80 my-model-api:latest
   * Verify correctness and basic latency.
5. **Deploying the Container**
   * Push the image to a registry (Docker Hub, ECR, GCR).
   * Deploy it on the chosen infrastructure (bare VM, Kubernetes, Cloud Run, ECS/Fargate, etc.) so it’s reachable over HTTP.

**Question 2.2 - Model Versioning, Updates & Monitoring**

**1. Model Versioning**

* Use a model registry (e.g., SageMaker Model Registry or GitHub Packages).
* Record metadata for each model: evaluation metrics and hyperparameters.
* Assign explicit versions to model names to track which version is running.
* Mirror model versions in Docker image tags.
* Use versioned keys when pushing the model to cloud storage.
* Embed the model version in the API payload so you can audit exactly which version served each request.

**2. Updates**

**CI/CD for Models**

* Automated retraining triggers (e.g. new labeled data arrives, scheduled monthly) kick off the training pipeline.
* Unit & integration tests ensure that new models meet quality standards.
* Build and push a new Docker image tagged with the new model version.

**Deployment Strategies**

* **Canary Deployments:**
  + Route a small percentage of traffic (e.g. 5 %) to the new version.
  + Compare metrics, then ramping up if everything looks good.
* **Compatibility & Schema Evolution:**
  + Version the API (e.g. /v1/predict vs. /v2/predict) if outputs change.
  + Maintain backward‑compatible APIs to avoid breaking existing clients.

**3. Monitoring & Alerting**

* **Operational Metrics:** Throughput & latency (p95/p99), error rates (HTTP 5xx, timeouts), infrastructure (CPU/GPU, memory, disk I/O).
* **Model Quality Metrics:** Alerts when performance dips below SLAs; monitor prediction distributions for data drift.
* **Logging & Visualization:**
  + Emit structured JSON logs with a unique request ID, timestamp, model version, hashed/sanitized inputs, and outputs.
  + Dashboard in Grafana or Cloud Monitoring for trend and anomaly visibility.
* **Data & Concept Drift Detection:**
  + Track feature statistics (mean, variance, percentiles).
  + Monitor label changes once true labels are collected (e.g. class ratios over time).

**Question 2.3 - Performance & Latency Optimization**

* Instrument the service to track end‑to‑end latency, as well as broken‑out times for preprocessing, inference, and postprocessing.
* Set alerts when p95 or p99 latency drifts upward.
* Reduce input resolution or change aspect ratios if acceptable.
* Train a smaller “student” network to mimic the outputs of the original “teacher” model.
* Simulate quantization during training to recover accuracy lost by post‑training methods.
* Convert the model to a static graph via TorchScript to eliminate Python overhead and unlock backend optimizations.
* Use tiny micro‑batches to boost GPU utilization with minimal extra latency.
* Implement an adaptive batching layer or queueing mechanism to collect requests within a tight time window before sending them to the GPU.
* Offload heavier preprocessing (image decoding, feature extraction) onto separate worker threads.
* Run multiple container replicas behind a load balancer to avoid noisy‑neighbor effects.
* Integrate a synthetic load tester (e.g. Locust or k6) into the CI pipeline to catch regressions in latency or throughput.
* Dynamically adjust batch size: if GPU isn’t fully busy, batch more requests; if traffic spikes cause slow “tail” responses, add more instances or shorten queueing time.

**Question 2.4 - Monitoring Model Performance & Data Drift**

1. **Capture Predictions & Labels**
   * Store each prediction alongside its eventual true label in a “results” table or time‑series database.
2. **Key Metrics**
   * **Classification:** accuracy, error rate, precision, recall, F1 (on critical slices, e.g. rare classes).
   * **Regression:** RMSE, MAE.
   * Plot these metrics at regular intervals (hourly/daily) and alert on sustained drops below SLA.
3. **Data Drift Monitoring**
   * Quantify shifts in feature distributions relative to the training baseline.
   * Monitor joint distributions or embeddings (e.g. via Maximum Mean Discrepancy).
   * Once labels are available, track shifts in conditional input‑output distributions.
   * Set drift thresholds and trigger alerts when exceeded.
4. **Feature Tracking**
   * **Numeric features:** mean, std, percentiles over time.
   * **Categorical features:** category frequencies.
   * Visualize as time series or run statistical tests.

**Question 2.5 - Continuous Integration & Delivery**

**Continuous Integration (CI)**

* **Version Control:**
  + Keep training code, serving code, and infra-as-code in sync (together or in well‑linked repos).
  + Adopt consistent Git branching strategies.
* **QA Testing:**
  + Unit tests for data-processing functions, model-building code, API handlers.
  + Static analysis: mypy/Pyright for type checks; Bandit for security scans.

**Continuous Delivery (CD)**

1. **Container Build & Push**
   * Build the inference service Docker image, copying in the versioned model artifact.
   * Push the image to your container registry (Docker Hub, ECR, GCR).
2. **Environment Promotion**
   * **Dev Environment:**
     + Automatically deploy every new image to a “dev” namespace/cluster.
     + Run integration tests (known inputs → expected outputs).
   * **Staging Environment:**
     + On merge to main, deploy to “staging.”
     + Run performance/load tests (throughput/latency).
     + Re‑validate model quality on a held‑out validation set.
   * **Approval Gates:**
     + Require manual approval (via GitHub/GitLab UI or Slack) before promoting towards production.

**Continuous Deployment**

* **Canary Deployment:**
  1. Deploy the new version to a small subset of traffic.
  2. Monitor error rates, latency, and model metrics against SLOs/SLA thresholds.
  3. Ramp to 100 % traffic if no red flags; roll back otherwise.