ChatGPT

# Project 3 – Part 4: CI/CD, Monitoring & Final Notes

This final part covers the operational aspects of your summarisation and research system: continuous integration/deployment, monitoring and alerting, security hardening and final deliverables. These practices ensure your project not only works but remains reliable, scalable and maintainable.

## 1. Multi-Environment CI/CD

### 1.1 Namespaces, Domains & Secrets

Organise each environment in its own Kubernetes namespace with distinct API hostnames and Netlify sites. Use a summary table to manage settings:

| Env | Namespace | API Host | Netlify Site | Secrets Source |
|---|---|---|---|---|
| **dev** | `sum-dev` | `api-dev.yourdomain.com` | `dev–yourapp.netlify.app` | K8s Secrets or External Secrets |
| **test** | `sum-test` | `api-pr-<PR#>.yourdomain.com` | Netlify deploy previews | PR-specific secrets |
| **staging** | `sum-staging` | `api-staging.yourdomain.com` | `staging–yourapp.netlify.app` | K8s Secrets or External Secrets |
| **prod** | `sum-prod` | `api.yourdomain.com` | `yourapp.com` | Cloud secret manager / Vault |

Each environment uses a separate PostgreSQL database (`sum_dev`, `sum_test`, etc.). Secrets (database credentials, API keys) are injected via Helm values and managed using an external secrets operator if available.

### 1.2 Helm Charts & Values

Create a Helm chart under `infra/helm/` with templated resources:

- **Deployments** for the API and vLLM server, with configurable replicas and resource requests. In prod, enable a `HorizontalPodAutoscaler` and `PodDisruptionBudget`.
- **Services** and **Ingress** definitions to expose the API; use `ingress.host` values specific to each environment.
- **ServiceMonitor** to configure Prometheus scraping.

- **Secrets** and **ConfigMaps** for environment variables (e.g. `MODEL_ID`, `PGHOST`, `PGUSER`).

Use environment-specific values files (`values-dev.yaml`, `values-test.yaml`, `values-staging.yaml`, `values-prod.yaml`) to override defaults. For example, `values-prod.yaml` might set:

```
api:
  replicas: 3
  resources:
    requests: { cpu: "500m", memory: "1Gi" }
    limits:   { cpu: "2",    memory: "3Gi" }
  env:
    MODEL_ID: mistralai/Mistral-7B-Instruct-v0.3
    VLLM_URL: http://sum-prod-vllm:8000/v1/chat/completions
prometheus:
  scrapeInterval: 15s
alerts:
  errorRate: { threshold: 0.01 }
  latencyP95: { thresholdSeconds: 0.8 }
```

### 1.3 GitHub Actions Workflows

Define the following workflows in `.github/workflows/`:

1. **preview.yml** – triggered on PR creation/updates. Builds the API image, pushes it to your registry, deploys `sum-pr-<PR#>` using Helm and builds a Netlify deploy preview. Cleans up on PR close.
2. **ci.yml** – runs static analysis (ruff), unit tests (pytest) and small integration tests. Fails the build on errors.
3. **deploy-env.yml** – triggered on pushes to `develop`, `staging` or release tags. Builds the image and deploys to the appropriate namespace using Helm. For production tags, require manual approval and check metrics (error rate, p95 latency, ROUGE regression) before promoting.
4. **nightly-train-eval.yml** – runs RLHF training on the latest feedback dataset, evaluates models on hold-out data (ROUGE, BERTScore, QAFactEval) and uploads reports. Use the results to inform gating conditions for future releases.

Use docker/build-push-action for image builds and azure/setup-helm or `helmfile` for chart deployment. Store your container images in a registry (e.g. GitHub Container Registry) and sign them with `cosign`.

## 2. Monitoring, Alerts & SLOs

Reliability is critical in production. Instrument your API and inference server and set up dashboards and alerts.

### 2.1 Metric Instrumentation

- **FastAPI** – use the `prometheus_client` library to expose metrics such as request count (`api_requests_total`), latency (`api_request_latency_seconds`), error count (`api_request_errors_total`), summary length and RLHF reward.
- **vLLM** – enable the internal `/metrics` endpoint to expose inference latency and token throughput.
- **RAG Pipeline** – record ingestion times, retrieval times and hit rates (percentage of queries where relevant passages were retrieved).

### 2.2 Dashboards & Alerts

Create Grafana dashboards to visualise:

- **Latency:** p50/p95/p99 for summarisation and research endpoints. Target p95 < 0.8 s for GPU-backed prod; < 2.5 s for CPU.
- **Throughput:** requests per second; monitor spikes and ensure autoscaling keeps up.
- **Error rate:** fraction of requests returning 5xx; alert if > 1 % over 10 minutes.
- **Quality metrics:** average ROUGE-L and QAFactEval scores over time; RLHF reward. Sudden drops may indicate model drift.
- **Resource utilisation:** CPU/GPU, memory and database connections per environment.

Set up Prometheus **alert rules** and use Alertmanager to route notifications. For dev/test, send alerts to a low-priority channel; for staging/prod, page the on-call engineer when SLOs are violated.

### 2.3 Service Level Objectives (SLOs)

Define SLOs to hold the system to a measurable standard:

- **Availability:** $\geq$ 99 % of requests succeed (non-5xx) per rolling 7-day window.
- **Latency:** 95 % of summarisation requests complete within 0.8 seconds (GPU) or 2.5 seconds (CPU) for production; 1 second tolerance in dev.
- **Quality:** ROUGE-L must not regress more than 2 % relative to the previous release; QAFactEval score must stay above a baseline threshold.
- **Error rate:** 5xx error rate < 1 % over 10 minutes.

CI/CD pipelines should gate production releases if SLOs are violated. Use canary rollouts or blue-green deployments to minimise risk.

## 3. Security & Compliance

- **Least privilege:** Run containers as non-root; limit container capabilities; enforce read-only root filesystems in prod.
- **Secrets management:** Use Kubernetes secrets or an external secret manager; avoid hard-coding credentials in code or Git.
- **SBOM & scanning:** Generate a Software Bill of Materials (e.g. with `syft`) and scan images for vulnerabilities (e.g. with `grype`). Sign images with `cosign`.
- **Network policies:** Deny all traffic by default; allow only necessary ingress to the API and egress to vLLM, Postgres and external search APIs.

- **Audit & logging:** Enable request/response logging; store logs in a centralised logging system; mask sensitive data.

## 4. Runbooks & Incident Response

Prepare runbooks for common scenarios:

1. **Deployment rollback:** Use `helm rollback` to revert to a previous release; verify via smoke tests.
2. **Database migration:** Outline steps to run Alembic migrations, back up the database and verify integrity.
3. **API outage:** Check Prometheus alerts; inspect logs; scale up replicas; redeploy if necessary.
4. **Model drift:** Monitor quality metrics; if drift is detected, trigger retraining or revert to a previous model.
5. **Security incident:** Rotate credentials, revoke tokens and review audit logs. Follow your organisation's incident response plan.

## 5. Final Deliverables Checklist

✔ **Project documentation:** Four parts outlining the problem, foundations, architecture, sprints, CI/CD and monitoring.

✔ **Source code:** Monorepo with `train`, `service`, `rag`, `eval`, `web` and `infra` directories.

✔ **Models:** Baseline, LoRA-tuned and RLHF-tuned checkpoints for T5, BART, Mistral/Mixtral. Reward model checkpoint.

✔ **RAG subsystem:** Ingestion script, retrieval logic, citations, integration with API.

✔ **Deep Research feature:** Search client, ingestion of web articles, multi-step planning, RLHF training.

✔ **Multi-document summarisation:** Map–reduce and refine implementations; evaluation results.

✔ **Web UI:** Netlify-hosted React app with summarise and research tabs; feedback collection.

✔ **Helm charts & values:** Parameterised for dev/test/staging/prod; includes ServiceMonitor and secret management.

✔ **CI/CD pipelines:** PR preview, dev/staging/prod deploy, nightly training, quality gates.

✔ **Monitoring & alerts:** Grafana dashboards; Prometheus metrics; alert rules tied to SLOs.

✔ **Runbooks & documentation:** Setup instructions, troubleshooting guides, incident response playbooks.

## 6. Conclusion & Next Steps

Building an automated summarisation and deep-research system is a comprehensive journey that touches on NLP, vector search, RLHF, web development and cloud infrastructure. By breaking the work into logical parts and adopting a sprint-based approach you develop both depth and breadth. Remember to iterate—start small, collect feedback, refine your models and prompts, and monitor everything. With the foundations laid in this series you are well on your way to mastering large language models and deploying them responsibly in production.

---