MLOPS Group 114: End-to-End Local MLOps Experiment (Heart Disease UCI)

MLflow • FastAPI • Docker • Minikube (Kubernetes) • Prometheus/Grafana • GitHub Actions

Date: 28 Dec 2025
One-command evaluation: make verify
Repository: https://github.com/deepakkt/bitsaiml-mlops-group-114

Ravindra Babu Katiki	2024AB05286	100%
Deepak Kumaran	2024AB05107	100%
Degala Venkatesh	2023AC05236	100%
Hemant Dyavarkonda	2024AB05109	100%
Ashish Kumar	2024AB05110	100%

A. Architecture Diagram

This project implements an end-to-end local MLOps workflow for the Heart Disease UCI dataset. It covers the full lifecycle from data acquisition and EDA, to model training and MLflow tracking, to production-like serving with FastAPI, containerization with Docker, deployment to local Kubernetes (Minikube), and monitoring using Prometheus and Grafana.

The workflow is designed to be reproducible and evaluator-friendly: a single command (`make verify`) can run the full pipeline (setup → lint/tests → data → training/MLflow → Docker smoke test → Minikube deploy → monitoring scrape check → teardown).

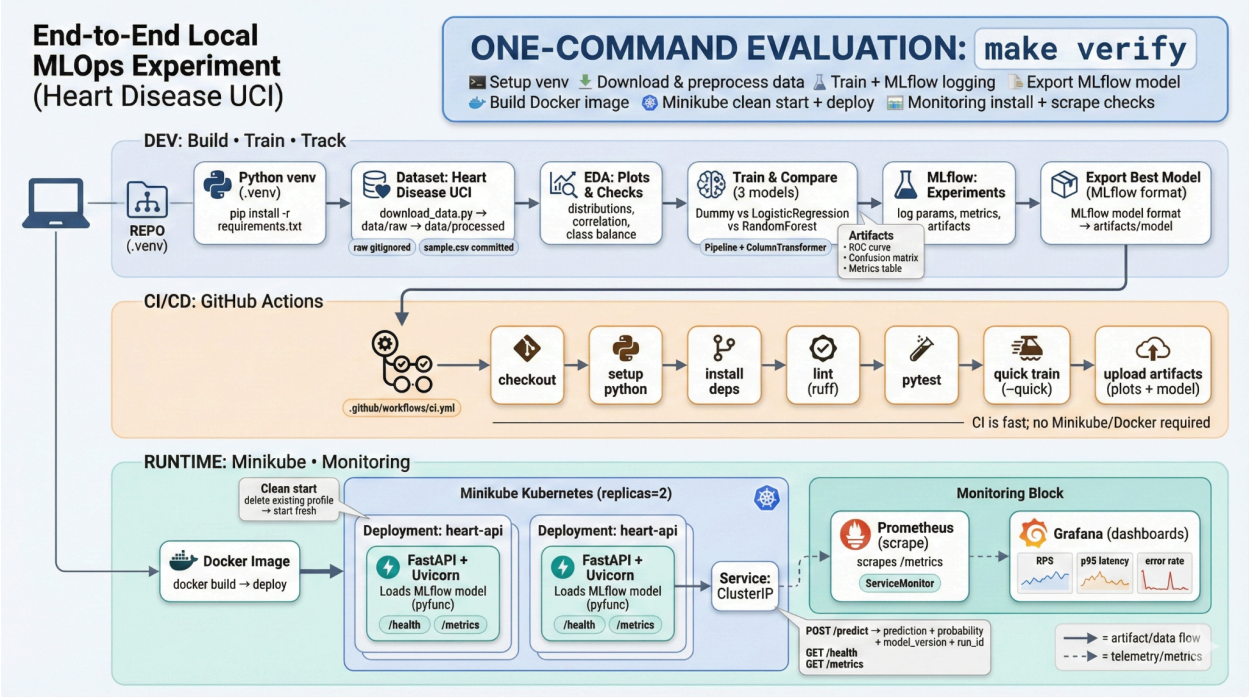


Figure 1. High-level architecture and data/telemetry flow for the local MLOps pipeline.

Key architectural blocks:

- Development (local): Python venv + Makefile orchestration; data download/preprocessing; EDA figure generation; training and model comparison.
- Experiment tracking: MLflow local file store (`./mlruns`) logging parameters, metrics, models, and plots for each run.
- Model packaging: Best model exported in MLflow model format under `./artifacts/model` (plus metadata with run_id).
- Serving: FastAPI + Uvicorn loads the exported MLflow model and exposes `/health`, `/predict`, and `/metrics`.
- Containerization: Docker image with application code and model artifact; local run via `make docker-run`.
- Kubernetes runtime: Minikube clean-start profile, Deployment with 2 replicas, ClusterIP Service, readiness/liveness probes.

- Monitoring: kube-prometheus-stack (Prometheus + Grafana) via Helm + a ServiceMonitor scraping `/metrics` from the API.

B. Setup and Install Instructions

Prerequisites (local developer machine):

- Python 3.11+ (venv + pip)
- `make`
- Docker (required for `make docker-build` / `make docker-run`)
- Minikube + kubectl (required for Kubernetes deployment)
- Helm (required for monitoring stack installation)

Recommended quickstart (from a clean clone):

```
# 1) Create/reuse the virtual environment and install dependencies
./scripts/bootstrap_venv.sh
source .venv/bin/activate

# 2) Run static checks and tests
make lint
make test

# 3) Download and preprocess data
make data

# 4) Optional: generate EDA figures (report/figures)
make eda

# 5) Train models and log to MLflow (full run)
make train

# 6) Run the API locally
make api
```

Useful supporting commands:

```
# View MLflow UI (local file store)
make mlflow-ui

# One-command evaluator run (end-to-end)
make verify

# If machine resources are tight, skip monitoring during verify
VERIFY_SKIP_MONITORING=1 make verify
```

Environment variables used by the project:

- `MODEL_PATH`: Path to an exported MLflow model directory (default: `./artifacts/model`).
- `MINIKUBE_PROFILE`: Minikube profile name (default: `mlops-assign1`).
- `VERIFY_SKIP_MONITORING`: When set to 1, `make verify` skips Prometheus/Grafana installation and scrape check.
- `GRAFANA_ADMIN_PASSWORD`: Optional override for Grafana admin password during `make monitor-up` (default: `prom-operator`).

C. EDA and Modeling Choices

Dataset and target definition

Dataset: Heart Disease UCI (binary classification). The goal is to predict presence (1) vs absence (0) of heart disease from a set of clinical features.

Dataset provenance and reproducibility metadata:

- Source: ucimlrepo id=45 (Heart Disease) (https://archive.ics.uci.edu/ml/datasets/Heart+Disease)
- Raw records: 303 | Clean records used for training: 297
- Download timestamp (UTC): 2025-12-28T11:18:50.923323+00:00
- SHA-256 checksum (raw CSV): fbd8e872af527dd3544a47b52cb3ce38f2f94b19082a40a1e20904a80d8377d6

Features used by the model (13 total) and target:

- Numeric: age, trestbps (resting blood pressure), chol (serum cholesterol), thalach (max heart rate), oldpeak, ca
- Categorical/ordinal: sex, cp, fbs, restecg, exang, slope, thal
- Target: target (binarized as `target > 0` → 1 else 0)

Data preparation and preprocessing

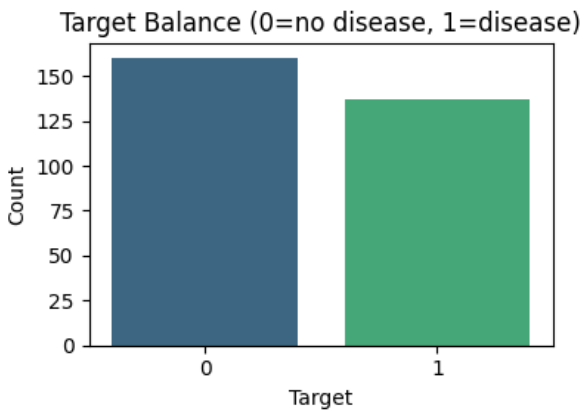
Data preparation is implemented in `src/heart/data.py` and `scripts/download_data.py` and follows a reproducible flow:

- Download via `ucimlrepo` (preferred) with a GitHub CSV fallback if needed.
- Standardize column names to lowercase; align target column naming (`num` → `target` when applicable).
- Convert values to numeric and coerce invalid tokens (e.g., '?') to missing values.
- Drop rows with missing values after coercion to ensure a clean training frame.
- Binarize target: `target = (target > 0).astype(int)` to map all disease severities to a single positive class.

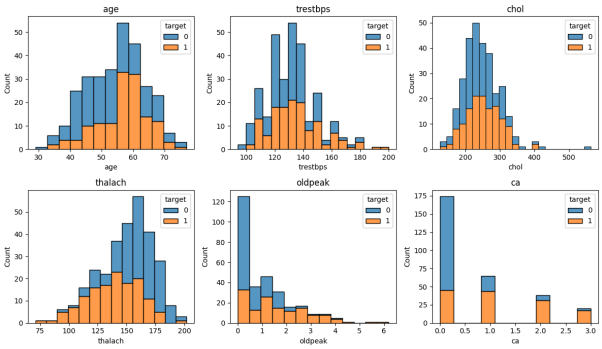
After preprocessing, the dataset contains 297 rows and 13 features (plus target). Class balance is approximately 160 negatives vs 137 positives (46.1% positive).

Exploratory Data Analysis (EDA)

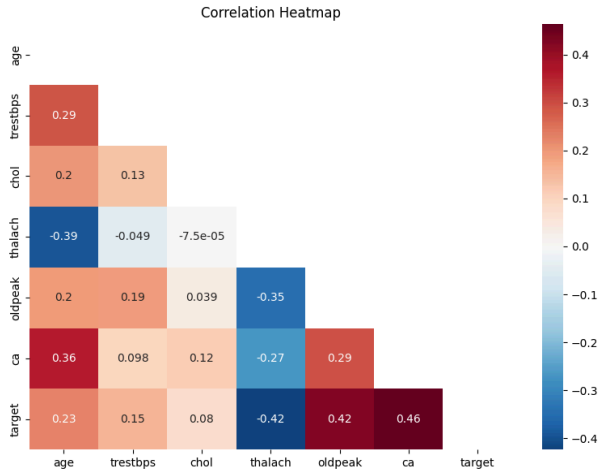
EDA is generated via `make eda` (script: `src/heart/eda.py`) and saved under `report/figures/`. The included plots cover class balance, per-feature distributions, correlations, and missingness.



Target balance (0=no disease, 1=disease).



Numeric feature distributions by class.



Correlation heatmap for numeric features and target.

Feature missingness heatmap (post-cleaning).

Selected EDA observations (from the processed dataset):

- Class balance is reasonably even (positives ~46.1%), enabling standard classification metrics without extreme imbalance handling.
- Among numeric features, the strongest absolute correlations with the target are: ca (+0.463), oldpeak (+0.424), and thalach (-0.424).
- Missing values are removed during preprocessing (the processed dataset contains no NaNs).

Modeling choices

The training pipeline is implemented in `src/heart/train.py` and compares three models using a consistent scikit-learn Pipeline + ColumnTransformer preprocessing stack:

- Baseline: DummyClassifier (most_frequent) to establish a low-effort benchmark.
- Logistic Regression: interpretable linear baseline, tuned via GridSearchCV.
- Random Forest: non-linear ensemble baseline, tuned via GridSearchCV.

Preprocessing strategy (in `src/heart/features.py`):

- Numeric features: median imputation + standard scaling.
- Categorical features: most-frequent imputation + one-hot encoding (handle_unknown=ignore).
- All preprocessing steps are packaged together with the estimator in a single Pipeline for reproducibility.

Evaluation methodology:

- Hold-out split: stratified train/test split (default test_size=0.20, random_seed=42).
- Cross-validation: StratifiedKFold CV with 5 folds for full training; 3 folds for quick/CI mode.
- Primary tuning metric: ROC-AUC (GridSearchCV scoring='roc_auc', refit=True).
- Reported metrics: accuracy, precision, recall, ROC-AUC, plus best cross-validated ROC-AUC score.

D. Experiment Tracking Summary

Experiment tracking is implemented with MLflow using a local file store. All model runs log parameters, metrics, plots, and the trained model artifact.

MLflow configuration:

- Tracking URI: file:./mlruns
- Experiment name: heart-disease-uci
- Runs logged: dummy, log_reg, random_forest (each with a unique run_id).

Where results are stored:

- `./mlruns/<experiment_id>/<run_id>/` contains MLflow run metadata, metrics, parameters, and artifacts.
- `./artifacts/training_summary.json` provides a consolidated JSON summary across all trained models.
- `./artifacts/plots/` contains locally persisted copies of ROC and confusion matrix plots for quick inspection.
- `./artifacts/model/` contains the exported best model in MLflow format for serving (plus metadata.json).

Run metrics summary

The table below summarizes the key CV and hold-out test metrics captured for each run.

Model	Run ID	Best CV ROC-AUC	Test ROC-AUC	Accuracy	Precision	Recall
dummy	06eabcfc678d471db421851b00ab4071	0.500	0.500	0.533	0.000	0.000
log_reg	ecd7b01bf3cf4e44a1241e6764766477	0.896	0.953	0.867	0.885	0.821
random_forest	79e54fdfc5fe413e952aa667963d61f7	0.894	0.936	0.817	0.840	0.750

Best model selected (by ROC-AUC, tie-breaker accuracy): log_reg (run_id=ecd7b01bf3cf4e44a1241e6764766477). Hold-out ROC-AUC=0.953, Accuracy=0.867.

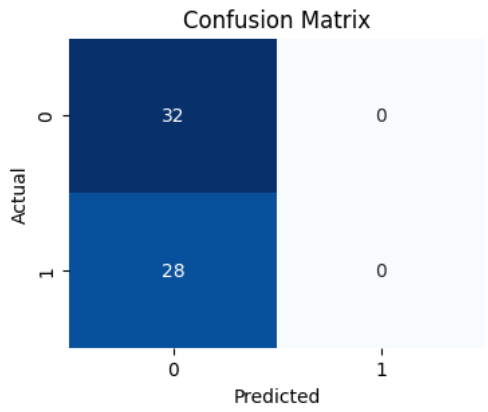
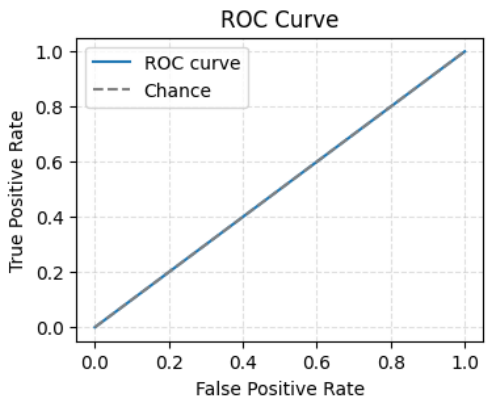
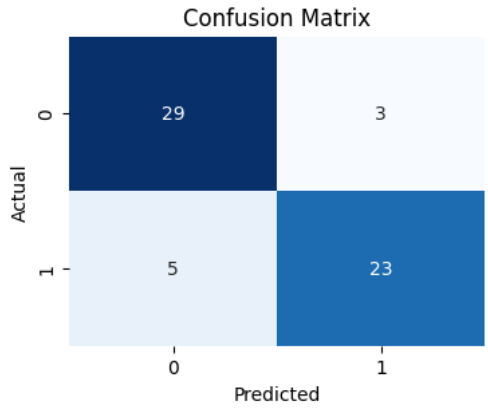
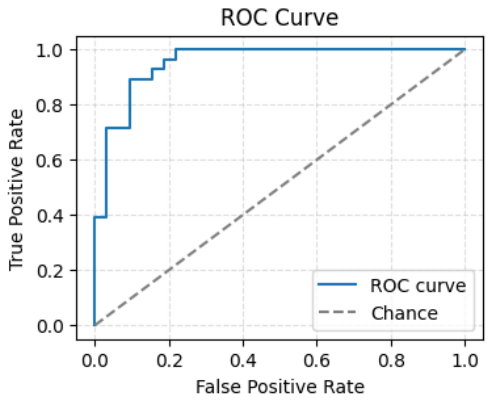
Best model hyperparameters:

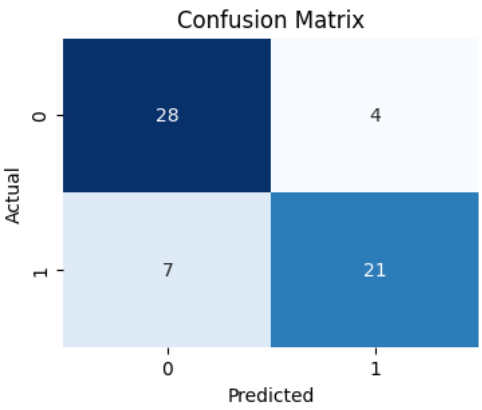
- model__C: 0.1
- model__max_iter: 500

- model__penalty: l2
- model__solver: liblinear

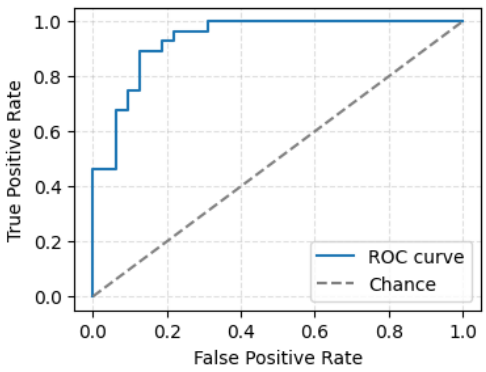
Logged plots (ROC curves and confusion matrices)

For each run, MLflow logs a confusion matrix and ROC curve plot. The same figures are also saved under `artifacts/plots/` for convenience.

Confusion Matrix	ROC Curve
 ***dummy: confusion matrix***	 ***dummy: ROC curve***
 ***log_reg: confusion matrix***	 ***log_reg: ROC curve***



random_forest: confusion matrix



random_forest: ROC curve

Model export for serving

After training, the selected best model is exported to `artifacts/model/` in MLflow model format (includes MLmodel, conda.yaml, python_env.yaml, requirements.txt, model.pkl, and metadata.json). The FastAPI service loads this exported model by default.

E. Local FastAPI Serving Summary

The project serves the trained model through a FastAPI application (`src/heart/api/main.py`) running on Uvicorn. The service is designed for production-like deployment and includes health checks, structured logging, and Prometheus-compatible metrics.

Service endpoints

- GET /health: liveness/readiness endpoint; returns model_loaded plus model_version and run_id when available.
- POST /predict: accepts a JSON payload with the 13 feature fields; returns prediction (0/1) and probability.
- GET /metrics: Prometheus exposition endpoint, including request count, latency histogram, and error counter.

Running locally

```
# Start the API (loads MLflow model from ./artifacts/model by default)
make api

# Health check
curl http://localhost:8000/health

# Prediction request (sample JSON is in data/sample/request.json)
curl -X POST -H "Content-Type: application/json" -d @data/sample/request.json
http://localhost:8000/predict

# Prometheus metrics (first few lines)
curl http://localhost:8000/metrics | head -n 20
```

Request/response contract

Sample request payload (file: `data/sample/request.json`):

```
{
  "age": 54,
  "sex": 1,
  "cp": 0,
  "trestbps": 130,
  "chol": 246,
  "fbs": 0,
  "restecg": 1,
  "thalach": 150,
  "exang": 0,
  "oldpeak": 1.2,
  "slope": 2,
  "ca": 0,
  "thal": 2
}
```

The prediction response includes: `prediction`, `probability`, and (when available) `model_version` and `run_id` to make requests traceable back to the exact MLflow run used for inference.

Observability (logging + metrics)

The service emits single-line JSON logs to stdout (suitable for container logs) with request-level fields such as `request_id`, `path`, `status_code`, `duration_ms`, and model identifiers (`model_version`, `run_id`). A request ID is also returned to the caller via the `X-Request-ID` response header.

Prometheus metrics recorded include:

- heart_api_requests_total{endpoint,method,http_status}
- heart_api_request_latency_seconds_bucket / _count / _sum (latency histogram)
- heart_api_errors_total{endpoint,method}

F. Docker Build Summary

Containerization is provided via `docker/Dockerfile`. The image packages the FastAPI application and the exported MLflow model artifact. The default local workflow mounts the model directory at runtime so the container always serves the latest exported model.

Docker image design

- Base image: python:3.11-slim
- Installs pinned dependencies from requirements.txt
- Copies application source (`src/`) and supporting scripts
- Copies sample request/data under `data/sample/`
- Exposes port 8000 and starts Uvicorn (`uvicorn src.heart.api.main:app --host 0.0.0.0 --port 8000`)

Build and run locally

```
# Prerequisite: export the model
make train

# Build image
make docker-build

# Run container (bind-mounts ./artifacts/model into /app/artifacts/model)
make docker-run

# Smoke test the running container
make smoke-test

# Stop the container
make docker-stop
```

Note: `make docker-run` sets `MODEL_PATH=/app/artifacts/model` and mounts `./artifacts/model` read-only. This keeps inference consistent with the latest locally exported model.

G. Kubernetes Deployment Process and Summary

Production-style deployment is implemented on a local Kubernetes cluster using Minikube. The deployment flow prioritizes reproducibility and a clean evaluator experience by starting from a fresh Minikube profile each time.

Cluster provisioning (clean start)

Scripts: `k8s/cluster_up.sh` and `k8s/cluster_down.sh` manage a Minikube profile (default: `mlops-assign1`). The `cluster_up.sh` script deletes any existing profile before starting a new cluster, ensuring a consistent baseline.

```
# Start Minikube with a clean profile
make k8s-up

# Tear down the Minikube profile
make k8s-down
```

Application deployment

Deployment is handled by `k8s/deploy.sh` and Kubernetes manifests under `k8s/manifests/`.

- Deployment: 2 replicas, readiness/liveness probes on /health, resource requests/limits, and MODEL_PATH env var.
- Service: ClusterIP service mapping port 80 → container port 8000.
- ServiceMonitor: applied automatically when the CRD exists (i.e., after monitoring stack install).

```
# Build image locally (or inside Minikube) then deploy
make train
make docker-build
make k8s-deploy

# Access via port-forward
kubectl -n default port-forward svc/heart-api 8000:80
curl http://localhost:8000/health
curl -X POST -H "Content-Type: application/json" -d @data/sample/request.json
http://localhost:8000/predict

# Remove resources
make k8s-undeploy
```

Kubernetes health and scaling

The Deployment uses standard Kubernetes probes:

- Readiness probe: GET /health (initialDelaySeconds=5, periodSeconds=10)
- Liveness probe: GET /health (initialDelaySeconds=10, periodSeconds=20)
- Replicas: 2 (supports basic load distribution and rolling updates)

H. Monitoring Summary

Monitoring is implemented with Prometheus and Grafana using the `kube-prometheus-stack` Helm chart. The FastAPI service exposes Prometheus metrics at `/metrics`, and a Kubernetes ServiceMonitor configures scraping.

Monitoring stack installation

Scripts: `k8s/monitoring/install.sh` installs (or upgrades) kube-prometheus-stack into the `monitoring` namespace. `k8s/monitoring/uninstall.sh` removes it.

```
# Install Prometheus + Grafana
make monitor-up

# Uninstall monitoring stack
make monitor-down
```

Prometheus scraping

- ServiceMonitor: `k8s/manifests/servicemonitor.yaml` selects the heart-api Service by label and scrapes /metrics.
- Scrape interval: 30 seconds.
- Automated check: `scripts/check_prometheus_target.sh` port-forwards Prometheus and verifies the target is UP.

Prometheus and Grafana access (port-forward):

```
# Prometheus targets UI
kubectl -n monitoring port-forward svc/kube-prometheus-stack-prometheus 9090:9090

# Grafana UI (default admin credentials: admin / prom-operator)
kubectl -n monitoring port-forward svc/kube-prometheus-stack-grafana 3000:80
```

Grafana dashboard

A ready-to-import Grafana dashboard JSON is included at `k8s/monitoring/grafana-dashboard.json`. It includes panels for request rate, p95 latency (from histogram quantiles), error rate, and pod CPU/memory.

I. Smoke Tests Summary

The repository includes smoke tests to validate the serving layer at different deployment stages (local process, Docker container, and Kubernetes service). These checks are used both standalone and as part of the `make verify` end-to-end evaluation target.

API smoke test script

`scripts/smoke_test_api.sh` validates that the API is reachable and can serve predictions using the sample request payload (`data/sample/request.json`).

- Calls GET /health and fails if HTTP status is not 200.
- Calls POST /predict (if the sample request file exists).
- Supports REQUIRE_MODEL=1 to fail if the model is not loaded (i.e., /predict returns 503).

Kubernetes smoke test

`scripts/k8s_smoke_test.sh` port-forwards the Kubernetes Service (`svc/heart-api`) and runs the same API smoke test against the forwarded localhost endpoint.

Monitoring scrape check

`scripts/check_prometheus_target.sh` confirms that Prometheus is scraping the heart-api target and that the target health is UP (using the Prometheus HTTP API /api/v1/targets).

How smoke tests are executed

```
# Local API smoke test (requires make api running in another terminal)
./scripts/smoke_test_api.sh

# Docker smoke test (build + run + validate)
make docker-build
make docker-run
make smoke-test
make docker-stop

# Kubernetes smoke test (requires cluster and deployment)
make k8s-up
make k8s-deploy
make k8s-smoke
```

J. CI/CD Summary

Continuous Integration is implemented via GitHub Actions (`.github/workflows/ci.yml`). The CI pipeline is intentionally kept fast and does not require Docker or Minikube; it focuses on code quality, unit tests, and a quick training run with MLflow logging.

CI workflow steps

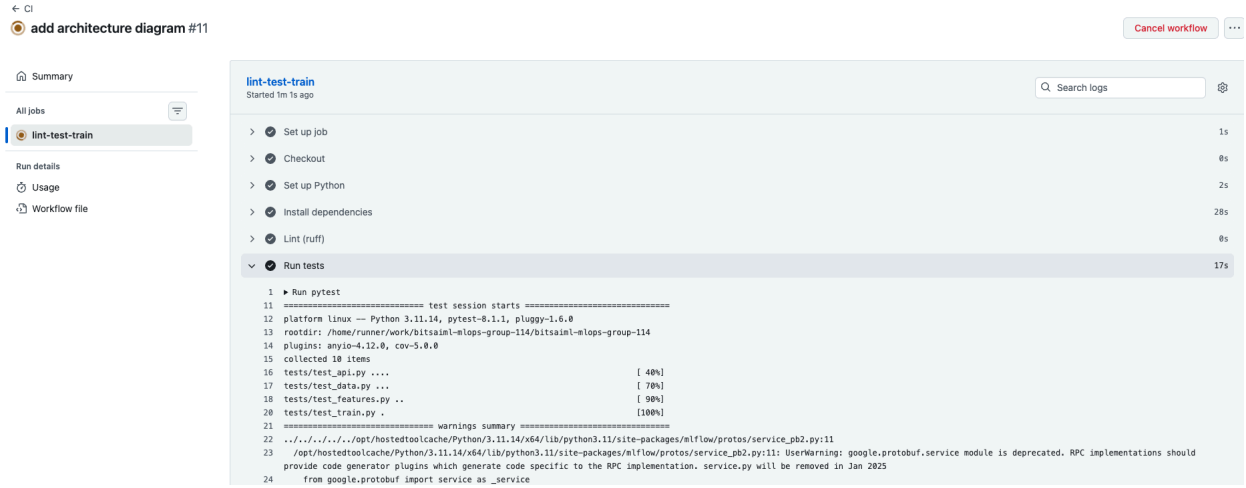
- Checkout repository source code.
- Set up Python 3.11 and cache pip dependencies.
- Install dependencies from requirements.txt.
- Lint: ruff checks on src/ and tests/.
- Unit tests: pytest.
- Quick training: python -m src.heart.train --quick --test-size 0.25 (logs to file:./mlruns).
- Artifact upload: uploads artifacts/** and mlruns/** to the workflow run for evaluator review.

Test coverage (high level)

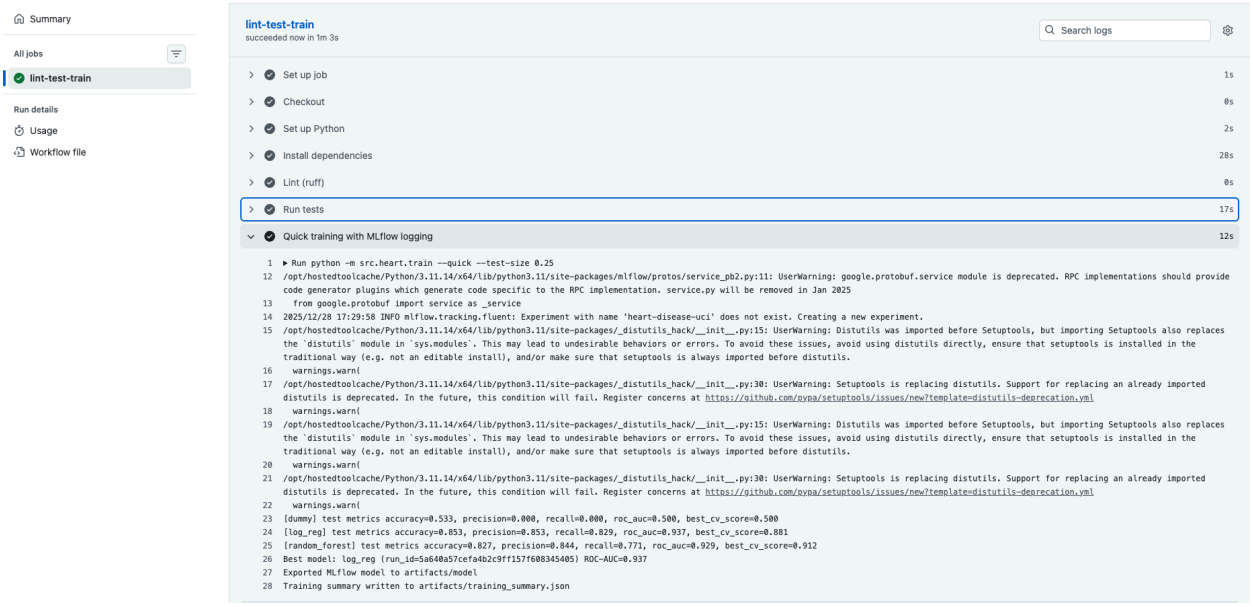
- Data: sample dataset existence, schema validation, and stratified train/test split.
- Features: pipeline fit on sample data and expected feature names.
- Training: quick training exports an MLflow model + training summary JSON.
- API: /health, /predict behavior with/without model, and /metrics output.

K. Screenshots

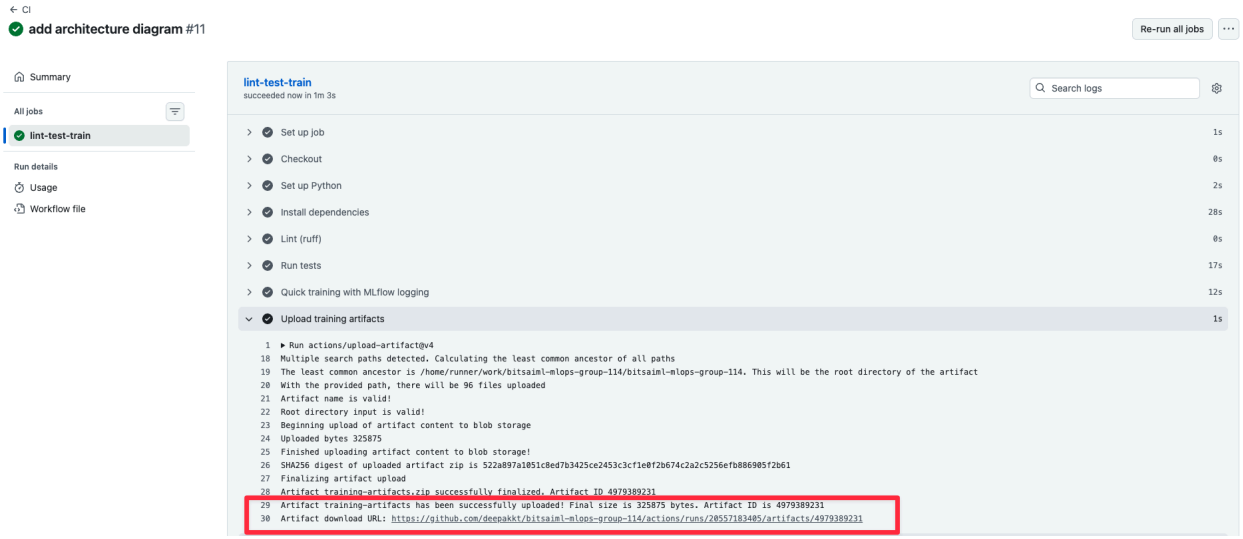
CI/CD - pytest



CI/CD - Training



CI/CD - Artifact Upload (Plots and Models)



K8s - Deploy model into local minikube cluster

```
(.venv) ➜  bits-sem-3-mlops-assignment git:(main) make k8s-deploy
bash k8s/deploy.sh mlops-assign1
[+] Loading image 'heart-api:local' into Minikube (mlops-assign1)...
[+] Applying deployment and service manifests to namespace 'default'...
deployment.apps/heart-api unchanged
service/heart-api unchanged
[+] ServiceMonitor CRD detected; applying servicemonitor.yaml...
servicemonitor.monitoring.coreos.com/heart-api unchanged
[+] Waiting for deployment rollout...
deployment "heart-api" successfully rolled out
[+] Current resources in namespace 'default':
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/heart-api    2/2     2            2           118m

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/heart-api    ClusterIP   10.105.40.38   <none>        80/TCP    118m
service/kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   169m

NAME                              READY   STATUS    RESTARTS   AGE
pod/heart-api-569ffc7df5-gh58f    1/1     Running   0          118m
pod/heart-api-569ffc7df5-pzcsd    1/1     Running   0          118m
Port-forward to reach the API:
  kubectl -n default port-forward svc/heart-api 8000:80
Then call:
  curl http://localhost:8000/health
  curl -X POST -H "Content-Type: application/json" -d @data/sample/request.json http://localhost
:8000/predict
(.venv) ➜  bits-sem-3-mlops-assignment git:(main)
```

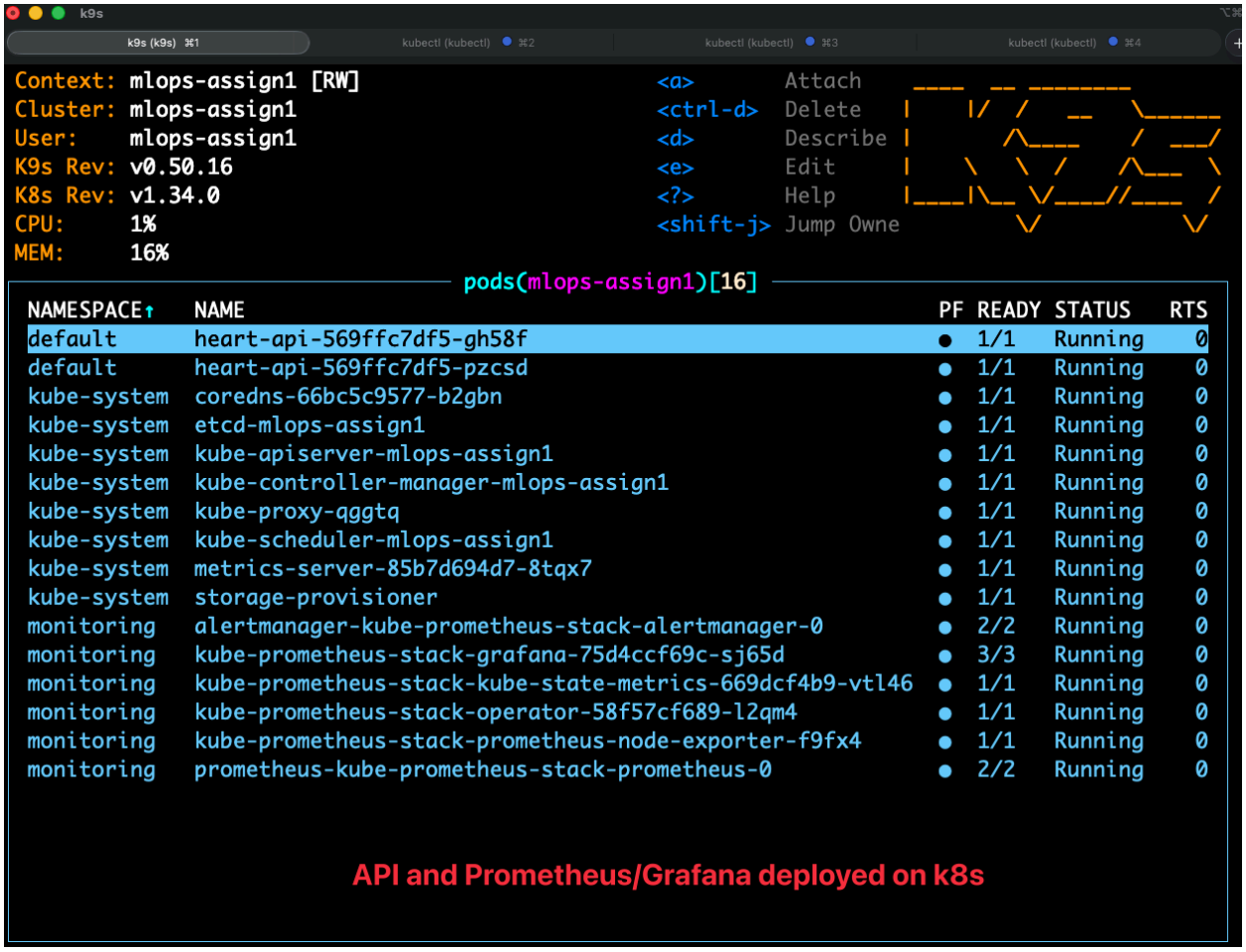
Model Deployment Into Cluster

K8s - Deploy Prometheus and Grafana into cluster

```
(.venv) ➜  bits-sem-3-mlops-assignment git:(main) clear
(.venv) ➜  bits-sem-3-mlops-assignment git:(main) make monitor-up
bash k8s/monitoring/install.sh
[+] Ensuring Minikube profile 'mlops-assign1' is running...
[+] Adding helm repo 'prometheus-community' (if needed)...
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
[+] Creating namespace 'monitoring' (if missing)...
[+] Installing/Upgrading 'kube-prometheus-stack' in namespace 'monitoring'...
I1228 23:11:00.946435   55455 warnings.go:110] "Warning: spec.SessionAffinity is ignored for hea
dless services"
I1228 23:11:00.948482   55455 warnings.go:110] "Warning: spec.SessionAffinity is ignored for hea
dless services"
I1228 23:11:00.950342   55455 warnings.go:110] "Warning: spec.SessionAffinity is ignored for hea
dless services"
I1228 23:11:00.952043   55455 warnings.go:110] "Warning: spec.SessionAffinity is ignored for hea
dless services"
I1228 23:11:00.953768   55455 warnings.go:110] "Warning: spec.SessionAffinity is ignored for hea
dless services"
Release "kube-prometheus-stack" has been upgraded. Happy Helming!
NAME: kube-prometheus-stack
LAST DEPLOYED: Sun Dec 28 23:10:54 2025
NAMESPACE: monitoring
STATUS: deployed
REVISION: 7
DESCRIPTION: Upgrade complete
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoring get pods -l "release=kube-prometheus-stack"

Get Grafana 'admin' user password by running:
```

K8s - Visual of deployed API and Monitoring stack on Minikube



L. Overall Summary

This submission delivers a complete, local-first MLOps pipeline for a binary heart disease classifier, covering data ingestion, EDA, reproducible preprocessing, model training and comparison, experiment tracking, packaging, API serving, containerization, Kubernetes deployment, monitoring, automated smoke tests, and CI.

Key outcomes:

- Data pipeline is reproducible with metadata (timestamp + checksum) and yields 297 clean rows for training.
- Model comparison across Dummy / Logistic Regression / Random Forest is tracked in MLflow; best model is 'log_reg' with ROC-AUC=0.953.
- Best model is exported in MLflow format for consistent serving across local, Docker, and Kubernetes runtimes.
- Operationalization includes health probes, JSON logging, Prometheus metrics, and a Grafana dashboard template.
- End-to-end evaluator command (`make verify`) ties together the full workflow with automated checks.

Potential next improvements (optional):

- Add an Ingress resource and domain-based routing for Kubernetes (beyond port-forward).
- Promote the best MLflow run to a model registry workflow (staging/production) for controlled rollouts.
- Expand monitoring with request payload sampling, model drift checks, and SLO-based alerts.
- Increase test coverage with contract tests for schema evolution and negative cases for input validation.

L. FAQ

(i) Why did you choose FastAPI over Flask?

1. First-class request/response validation. This project has a typed schema (PredictionRequest, PredictionResponse) and uses Pydantic under the hood. That gives strict input validation, better error messages, and fewer bad requests (e.g., missing fields, wrong types). In Flask, you typically add this via extra libraries (Marshmallow / Pydantic integration) and more boilerplate.
2. Automatic OpenAPI spec + interactive docs. FastAPI generates OpenAPI/Swagger docs out of the box from type hints. For an ML inference API, that makes it easier to verify payload type quickly and reduces friction. Flask can do this too, but usually via Flask-RESTX/Swagger installations which needs more setup
3. ASGI + modern concurrency model. FastAPI runs on ASGI servers like Uvicorn, which handle concurrent requests efficiently and support async endpoints cleanly. Even if inference is CPU-bound (like sklearn which we use here), the ASGI stack still helps with concurrent I/O (logging, metrics, health checks) and aligns with modern production Python web stacks. Flask is WSGI by default; it can scale, but typically via Gunicorn workers and isn't async-native.
4. Cleaner "production" patterns with less glue. Dependency injection patterns, middleware, structured responses, and type-driven development all tend to be smoother in FastAPI for small-but-real services.

Equivalent/valid alternatives: Flask is absolutely viable for ML inference (and many production systems use it). If you want Flask, you'd typically add: request validation (Marshmallow/Pydantic), OpenAPI generation, and robust middleware/logging/metrics—FastAPI just packages more of this into a cohesive default experience.

(ii) Why do you have two replicas in the final k8s deployment? Why not 1 or 3 or more?

Replica=1 is a single point of failure at the pod level. With one pod, any restart (crash, liveness probe failure, node pressure eviction) means the service is temporarily unavailable. Even a rolling update can cause downtime depending on rollout strategy and readiness timing.

Replica=2 is the smallest number that demonstrates "real" Kubernetes behaviors. With 2 pods you get:

- pod-level redundancy (one can restart while the other continues serving),
- service-level load balancing (ClusterIP service distributes traffic), and
- a more meaningful demonstration of readiness/liveness probes in action.

Why not 3+ in this particular setup? Once you've demonstrated horizontal scaling and redundancy, adding more replicas primarily becomes a capacity decision driven by load/SLOs. For a local cluster (often single-node), 3+ replicas can be diminishing returns: you consume more CPU/memory, increase scheduling pressure, and make the environment slightly less predictable without materially improving the "proof" of scaling concepts.

What I would do in production: pick replicas based on SLOs + load and use an HPA (Horizontal Pod Autoscaler). A common pattern is: minimum replicas ≥2 for pod redundancy, then autoscale based on CPU or request metrics.

(iii) Why did you choose a local minikube cluster instead of a cloud based k8s provider?

1. Reproducibility and clean evaluation. Minikube allows the entire system (API deployment + monitoring stack) to be brought up in a known, consistent environment without relying on external infrastructure state. This reduces variability caused by cloud networking/IAM, cluster policies, or provider-specific defaults.
2. Tight feedback loop. Local Kubernetes makes iterative testing fast: build → deploy → smoke test → adjust manifests without pushing images to a registry or dealing with remote cluster access.
3. The manifests remain portable. The Kubernetes resources used (Deployment, Service, ServiceMonitor) are standard objects. Moving to GKE/EKS/AKS is mostly operational (image registry, ingress/load balancer wiring, secrets, IAM), not a rewrite of the app.
4. Controlled cluster reset behavior. The "clean start" approach (delete profile and start fresh) prevents the common pitfall of "it worked because something was already installed in my cluster," which matters a lot for demonstrability.

Equivalent/valid alternatives: kind, k3d, or Docker Desktop Kubernetes are also valid local options. Cloud K8s is great when you need realistic networking, managed load balancers, multi-node behavior, and real IAM/secrets integration. But it adds external dependencies and operational variability.

(iv) Why did you not use a feature store in training?

- **The feature computation is fully captured in the model pipeline.** This project uses an sklearn `Pipeline` + `ColumnTransformer` for preprocessing (imputation/encoding/scaling) and logs/exports the model in MLflow format. That already

addresses the biggest practical risk feature stores often mitigate in small setups: **training-serving skew**.

- **Feature stores shine when you have "real" feature operations complexity.** A feature store is most valuable when:
    - multiple models reuse shared features,
    - features are computed from multiple upstream sources (batch + streaming),
    - you need point-in-time correctness (as-of joins) for offline training,
    - you require governance/lineage and online/offline parity at scale.
      For a single static CSV dataset and a single model service, a feature store adds infrastructure complexity without improving correctness materially beyond what a well-packaged pipeline already provides.
- **Avoiding additional moving parts in an end-to-end demonstration.** Feature stores typically introduce: storage backends, registry/config, offline store ingestion, online store serving, and additional operational concerns. For this dataset and scope, that overhead doesn't improve the core reliability of the pipeline compared to a robust pipeline artifact.

Equivalent/valid alternatives: Adding Feast (or a managed feature store) would be a logical next step if this moved from "static dataset" to "features computed from operational systems," or if multiple downstream models started consuming shared features.

(v) You expose services to localhost via port forwarding. Is there a better way to do it using a local cluster and accessing the services directly via an endpoint on the host? If yes, why wasn't that considered?

Yes. There are "more production-like" options than `kubectl port-forward`:

Better options (local cluster)

- **NodePort / `minikube service`**: expose a service via NodePort and let Minikube provide a reachable URL (`minikube service heart-api --url`).

- **Ingress (minikube ingress addon)**: create an Ingress resource and route via a hostname (often with an `/etc/hosts` entry). This looks closest to real cluster routing.

- **LoadBalancer + `minikube tunnel`**: use `Service type: LoadBalancer` and run `minikube tunnel` to get a stable IP/endpoint on the host.

Why port-forward is still a strong choice here

- **Minimal moving parts and fewer failure modes.** Port-forward works with *just* kubectl and a ClusterIP service. Requires no ingress controller setup, no DNS/hosts editing, no tunnel process that may require privileges.

- **More deterministic for scripted smoke tests.** It's easy to automate reliably: forward a known port, run a curl-based smoke test (which is what we do via the smoke tests script), and stop.

- **Reduced host exposure.** Port-forward binds to localhost by default, which is a simple, safe default for local testing.

- **Cross-platform consistency.** Ingress/tunnel workflows can behave differently across OS/network configurations; port-forward is usually the least surprising.

If you wanted to improve this today: add an Ingress manifest + enable Minikube ingress addon. That would give a stable URL without port-forward, and it's closer to how cloud clusters are typically accessed. We've already demonstrated this is possible via the port forward logic and we stop the demonstration at this point.

(vi)Three improvements you would do on this current experiment setup (write a brief paragraph for each)

Improvement 1: Add "gated" model promotion and a real model lifecycle (MLflow Registry + quality gates)

Right now, training produces metrics and an exported model, but the promotion decision is largely manual. A strong next step is to formalize **model promotion**: register each candidate in an MLflow Model Registry (or implement a registry-like flow with tags), compute metrics on a fixed holdout set, and only promote a model if it beats a baseline (e.g., ROC-AUC threshold + precision/recall constraints). You can also add simple checks like "no metric regression vs last production model."

This converts the project from “training runs exist” to a **repeatable release process** where deployment is tied to measurable quality.

Improvement 2: Strengthen reproducibility with explicit data/model lineage and versioning

The project already logs MLflow runs and exports artifacts, which is good. The next step is to ensure every deployed model is traceable to **exact code + exact data**. Concretely: log the git commit SHA, dataset checksum, and preprocessing schema as MLflow tags/artifacts; add a lightweight data versioning approach (DVC or at least strict checksums and immutable raw data references); and make the Docker build consume a specific model artifact version (e.g., by MLflow run ID) rather than “whatever is currently in `artifacts/model`.” This eliminates ambiguity and makes rollbacks and audits much easier.

Improvement 3: Production hardening for serving (Ingress + autoscaling + security + load testing)

For prod serving, I would add an Ingress (or Gateway API) with a stable hostname, configure timeouts and graceful shutdown, and introduce autoscaling (HPA) driven by CPU and/or request metrics. In parallel, I’d add basic API hardening: authentication (API key/JWT), rate limiting, and request size limits which are common issues for public endpoints. Finally, include a simple load test (Locust/k6) to measure latency and error rate under load, and validate that scaling and monitoring behave as expected. This turns the deployment into a very robust and stable output.

M. Code Repository

GitHub repository (source of truth):
https://github.com/deepakkt/bitsaiml-mlops-group-114

Repository highlights:

- `Makefile`: single-command orchestration (including `make verify`).
- `scripts/`: bootstrap, data download, smoke tests, and Prometheus scrape verification scripts.
- `src/heart/`: core Python package for data prep, features, training, evaluation, MLflow utilities, and API service.
- `docker/`: Dockerfile for containerized serving.
- `k8s/`: Minikube cluster scripts, Kubernetes manifests, and monitoring install/uninstall scripts.
- `.github/workflows/ci.yml`: GitHub Actions workflow for lint/test/quick-train and artifact upload.
- `report/figures/`: generated EDA plots (screenshots to be added separately in `report/screenshots/`).