CityAssist — Python Engineering & Data Science Detailed Guide

Project: CityAssist — Citizen-Centric AI Platform

Purpose

This document separates the responsibilities of the Python Engineering team (model serving, APIs, ETL pipelines, infra integration)
and the Data Science team (data acquisition, modelling, evaluation, explainability). Give this document to those teams so they can work concurrently
and deliver production-quality artifacts for the hackathon.

Part A — Python Team (FastAPI, Model Serving, ETL, Integration)

Objective:
The Python Engineering team converts data science outputs into reliable, secure, and scalable services. They build ETL pipelines, model serving APIs,
orchestration, logging, monitoring, and ensure the models run in production with proper error handling and performance SLAs.

Key Responsibilities

• Build lightweight inference APIs (FastAPI) for each ML model with clear request/response contracts.
• Create ETL pipelines to ingest raw city data (sensors, traffic feeds, utility logs) into the central data store (Postgres/Parquet on S3).
• Implement batch training triggers and online/near-real-time scoring as required.
• Containerize model services (Docker) and provide Helm charts for Kubernetes deployment.
• Implement caching (Redis) for frequent predictions and rate-limiting.
• Add robust logging (structured JSON), metrics (Prometheus) and distributed tracing (OpenTelemetry) for all services.
• Implement retries, circuit-breaker patterns, and graceful degradation (fallbacks) when model service is unavailable.
• Provide health-check endpoints: /health, /metrics, /ready, and readiness/liveness probes suitable for k8s.

APIs & Contracts (examples)

1) Flood / Risk prediction
POST /api/v1/predict/flood
Request JSON:
{ "zone_id": "Zone-A", "timestamp": "2025-11-06T10:00:00Z", "features": { ... } }

Response JSON:
{ "prediction_id": "pred-001", "risk_score": 0.72, "probability": 0.72, "explain": [{ "feature": "rain_mm", "contrib": 0.32 }], "meta": { "model": "flood-v1" } }

2) Image triage (report classification)
POST /api/v1/classify/report-image (multipart/form-data: file)
Response JSON:
{ "label": "pothole", "confidence": 0.91, "suggested_priority": "high" }

ETL & Data Pipelines

• Use Airflow (or Prefect) to orchestrate daily ETL jobs: ingest → validate → transform → load into a feature store (Parquet on S3 or Postgres).
• Implement data validation with Great Expectations or custom checks to catch schema drift and missing critical fields.
• Produce versioned, time-stamped feature tables for training and serving. Keep lineage metadata (who, when, source).
• Create a lightweight feature store directory or use Feast if time permits.

Model Serving & Deployment

• Wrap models with FastAPI and Uvicorn/Gunicorn with a single endpoint per model. Ensure JSON schema validation (Pydantic).
• Use Docker multi-stage builds and optimize image size. Provide sample Dockerfile and Helm chart values for deployment.
• Expose Prometheus metrics for latency, throughput, error rate, and model-specific metrics (AUROC, accuracy if available).
• Implement A/B testing or shadow mode: route a small percentage of live traffic to new model versions and compare metrics.

Observability & Reliability

• Export metrics: inference_time_seconds, requests_total, errors_total, cache_hits, model_version.

• Log traces and errors to central logging (ELK or Loki). Tag logs with request_id, user_id, model_version.
• Implement alerting rules: high error rate, model latency above threshold, data drift detection events.

Security & Ops

• Secure endpoints with API keys or JWT validation provided by Java backend (agree on auth header key).
• Do not accept arbitrary code execution. Limit file upload sizes and sanitize inputs.
• Store model artifacts in secure S3 buckets and use immutable versioning.

Deliverables — Python Team

• FastAPI repos with endpoints and Dockerfile for each model/service.
• ETL DAGs (Airflow) or Prefect flows, and data validation checks.
• Helm charts for k8s deployment and README for deployment steps.
• Postman collection for all model endpoints and example payloads.
• Monitoring dashboards (Prometheus + Grafana) & runbook for incidents.

Part B — Data Science Team (Research, Models, Evaluation, Explainability)

Objective:
The Data Science team focuses on reliable model development, evaluation, documentation, and long-term reproducibility. They produce models that the Python Engineering team will serve.

Key Responsibilities

• Data acquisition: gather datasets (Kaggle, NOAA, IMD, city sensors) and register each dataset with metadata (source, license, size, schema).
• Exploratory Data Analysis (EDA): produce notebooks with visualizations, seasonality checks, missingness, correlation matrices, and geospatial plots.
• Feature engineering: create temporal, spatial, and contextual features. Document transformation logic and tests.
• Model training & selection: baseline models (XGBoost, RandomForest), deep-learning models for images (U-Net), sequence models (LSTM/Transformer) for time-series.
• Model validation: strict train/validation/test splits using time-aware and spatial cross-validation; record metrics and confidence intervals.
• Explainability: use SHAP/Integrated Gradients for tabular and DeepSHAP/Grad-CAM for

image models.
• Model governance: track experiments (MLflow), store artifacts, and register finalized models with version and metadata.

Datasets & Sources (suggested heavy datasets)

• SEN12FLOOD (Sentinel-1/2 flood imagery) — large multi-spectral imagery for segmentation tasks.
• Air Quality timeseries (India) — sensor-level hourly data for forecasting and alerting.
• US Accidents dataset — traffic incidents with weather & severity for incident prediction.
• Local city sensor dumps (simulated) — include power consumption, water outages, event logs.

Modeling Tasks & Hard Challenges

• Build a hybrid model for flood risk: combine tabular rainfall/radar features with satellite image segmentation outputs.
• Build time-series forecasting pipelines (LSTM/TCN/Transformer) for short-term AQI and energy demand predictions.
• Create an image classifier for citizen reports (pothole vs garbage vs fallen tree) with explainability output.
• Implement model robustness tests: adversarial/noisy input checks, input distribution shift testing, and backtest over historical windows.

Experiment Tracking & Reproducibility

• Use MLflow or Weights & Biases to log experiments, parameters, and metrics. Save reproducible environment specs (conda env or requirements.txt).
• Provide training scripts that can be run via CLI (train.py) and accept config files for hyperparameters.
• Provide model card for each final model describing dataset, limitations, expected performance, and usage guidance.

Integration with Python Engineering

• Deliver model artifact (serialized weights + pre-processing code) with a small README explaining input feature schema and expected JSON request shape.
• Provide a lightweight wrapper (predict.py) that the Python team can import/use to build the FastAPI endpoint quickly.
• Collaborate on health checks and define acceptable ranges for prediction outputs and

confidence thresholds.

Evaluation & Acceptance Criteria

• Models must include clear performance reports (ROC, Precision-Recall, RMSE, MAE as applicable) and confusion matrices for classification tasks.
• Provide qualitative examples and failure cases (10 examples where model fails and why).
• Document retraining triggers: data drift threshold, model degradation over time, or scheduled periodic retraining.

Deliverables — Data Science Team

• Cleaned datasets and EDA notebooks.
• Training scripts and best-performing model artifacts with versioning.
• Experiment logs (MLflow/W&B) and model cards.
• Explainability notebooks and sample visualizations.
• Integration README with example request/response payloads.

Joint Integration Checklist (Python + Data Science)

• Agree on JSON API contract for each model endpoint (input fields, types, output schema).
• Provide sample payloads and unit tests validating end-to-end flow (sample request -> model -> response).
• Include canary test to route 1% traffic to new model versions and compare metrics.
• Ensure the model inference latency meets the SLAs (e.g., < 300ms for small models, < 2s for heavier ones) or have clear async patterns.

Final Notes & Best Practices

• Keep security in mind: never log raw PII, sanitize inputs, enforce strict upload limits.
• Automate as much as possible: CI for unit tests, CI for training reproducibility (optional), CD for model serving images.
• Keep documentation developer-friendly with code snippets, examples, and clear contact points for cross-team questions.