HORI — Real-Time Weather & Air-Quality Risk Visualization— Technical Report (Deliverable 1)

Team Name: *Group 9*
Team Members:

1)Sai Guddeti,

2) Param Upadhyay,

3) Nakita Ramachandra Purohit

4) Priyanka Tentu

Course: CSC 495/581 – Topics in Computer Science(Cloud Engineering)
Date: 09/28/2025

Summary

What we are building?

HORI is a map-based system that brings together live weather and air-quality observations, blends them with near-term forecasts, and turns them into a single explainable risk score called the Health-Oriented Risk Index (HORI) on a 0–100 scale. A user can click any point on the map to see Temperature, AQI, and HORI immediately. For trips, the user enters a start, end, and time window; we sample the route along distance and time, compute HORI at each sample, and color the route by risk bands (Low / Moderate / High / Extreme).

Why it matters?

People already check weather; air quality is now just as important. These signals live on different sites and use different units. Most users want three answers: *Is it okay to go now? If not, when is better? Which part of my route is worst?* HORI answers these questions in a way that is transparent (we show the top factor—AQI, heat, wind, or precipitation) and visual (color-coded segments you can understand at a glance).

How will we run it?

We will deploy HORI on a small CloudLab Kubernetes cluster. The system is intentionally split into focused services that mirror the data journey: Ingestion pulls observations/forecasts; a Data Store persists them; the API coordinates requests; the HORI Calculator produces a clear 0–100 score; and the Frontend shows popups and color-coded routes. The Frontend is the only public service; all other services are private inside the cluster. The API pod includes a sidecar for proxy/logging. We keep state with a persistent volume so history survives restarts, expose liveness/readiness checks and a horizontal autoscaler (HPA) for resilience and scale, protect programmatic endpoints with basic auth, emit logs/metrics to prove the system is healthy, and ship updates Our CI/CD demo will perform coordinated rolling updates across multiple connected services—first the API, then the Frontend—with smoke tests (health check + sample query) to show zero-downtime.

Chapter 1 — Vision

1.1 Problem and motivation

Weather and air quality change quickly and are reported in different units across different websites. This makes it hard to judge near-term, health-related risk—especially when planning travel. Our vision is to provide a **single, understandable number—HORI** (0–100)—**with a plain-English reason** for that number (for example, *"AQI is the main driver"*). We keep it simple (Low → Extreme bands) and **explainable** so non-experts can trust and act on it. Beyond point lookups, HORI supports **trip planning** by **showing how risk changes along a route over time**, not just at the start and end.

1.2 What the user can do

- **Point view:** Click any location to see **Temperature, AQI, HORI, Band**, plus a short note (for example, "AQI high; sensitive groups take care").
- **Trip view:** Enter **origin, destination, and a time window** (for example, depart 1:00 pm; arrive 4:00 pm). The system **samples** the route and assigns timestamps between departure and arrival. Each sample is scored; the route is color-coded by risk. A side panel summarizes **distance, duration, worst segment, average temperature, max AQI, and average HORI.**

1.3 Clean 10,000-ft architecture

There is one public entry point (the Frontend web app). All internal services are private inside the cluster. Data flows from Ingestion into a Data Store. The API decides whether to use observations or forecasts, calls the HORI Calculator, and returns JSON to the Frontend. For trips, the Routing service provides geometry and timing so we can score each segment.

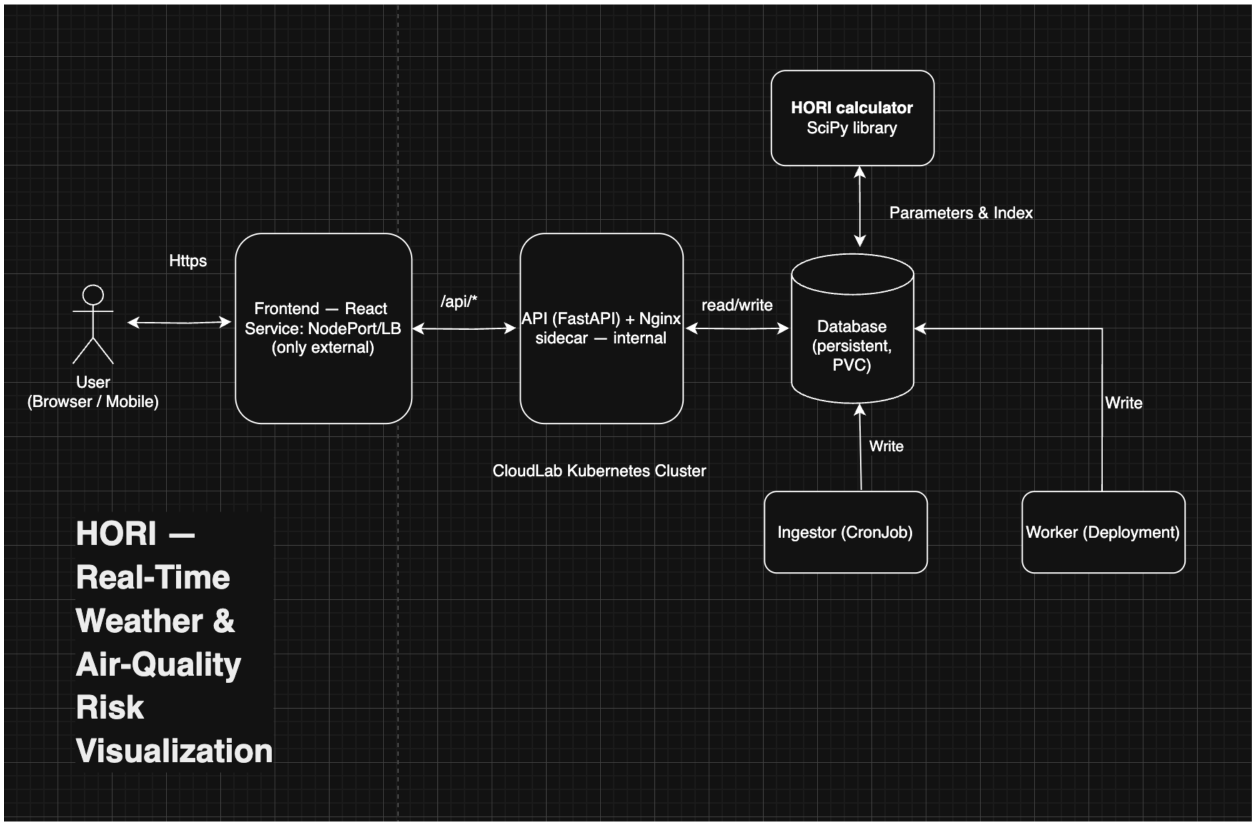

Figure 1. The Frontend is the only public service and proxies API calls. Internal services (API, HORI, Ingestion, Routing, Data Store) are private (ClusterIP). The API pod runs with a sidecar for proxy/logging. The Data Store uses a PVC so history survives restarts. Probes and HPA provide self-healing and scale; logs/metrics verify operation.

Why we designed it this way

- **One public door**. Only the **Frontend** is publicly reachable. It serves the web UI and proxies API calls. This keeps the edge simple and secure.
- **Private internals**. **API, HORI, Ingestion, Routing, Data Store run as ClusterIP services**. This clean boundary is easy to reason about and demonstrate.
- **Pipeline by responsibility**. Ingestion pulls and normalizes data → Data Store persists → API orchestrates → HORI computes → Frontend renders. Clear separation makes failures easier to isolate.
- **Sidecar pattern**. The **API pod** runs **API + sidecar** (proxy/logging) to keep request logs close to the service without mixing concerns in app code.
- **State and resilience**. The Data Store uses a **Persistent Volume Claim (PVC)** so history survives restarts. **Liveness/readiness** probes and **HPA** (on API) provide self-healing and scale-out under load.
- **Explainability**. Every HORI response includes the top factor and the inputs used (for example, AQI, Temp, Wind, Precip).

Chapter 2 — Proposed Implementation

2.1 End-to-end behavior

1. **Data refresh (background).** The Ingestion service **periodically pulls** current observations (weather + air quality) and **near-term forecasts**. It standardizes units, deduplicates, and writes to the Data Store with timestamps and coordinates.
2. **Point query.** The user clicks a point (lat, lon, time). The API selects **observations** for "now or past," and **forecasts** for future timestamps (nearest lead). It calls the HORI Calculator to compute a **0–100** score and **band** (Low/Moderate/High/Extreme), and returns **Temperature, AQI, HORI, Band**, plus a short **explanation** (for example, "Top factor: AQI 165").
3. **Trip query.** The user provides origin, destination, and a time window. The API asks the **Routing** service for a route and travel profile; it **samples points** along the polyline and assigns timestamps between departure and arrival. For each sample, it chooses obs/forecast data and computes **HORI**. The API returns **GeoJSON segments** colored by risk and a **trip summary** (distance, duration, worst segment, average HORI, max AQI, average temperature).
4. **Graceful fallback.** If a provider is slow or down, the API returns **cached last-known values** with a **data-age** flag so the UI can warn users.
5. **Security.** Programmatic v1 endpoints use **basic auth** (stored as Kubernetes Secrets). The Frontend proxies internally so end users never handle credentials.

2.2 Components and what each one contributes

- **Frontend (public; only external service)** - Renders the map and side panel, handles search and trip forms, **proxies** API calls, and displays popups and color-coded segments.
- **API (with sidecar)** - Validates requests, chooses observation vs. forecast by timestamp, calls HORI, and assembles JSON responses. The **sidecar** in the same pod provides proxy/logging (multi-container pod).
- **HORI Calculator** - Converts meteorology + AQI inputs into sub-scores and a combined **0–100 HORI** with an **explanation** ("top factor"). The initial formula is intentionally simple and adjustable; clarity is the priority.
- **Ingestion** - Pulls observations and forecasts on a schedule, normalizes units, deduplicates, and writes to the Data Store; maintains a small **cache** for hot tiles/locations.
- **Routing** - Builds a route from origin → destination and provides **timestamps** for sampled points so risk can be evaluated **over time** along the path.
- **Data Store (persistent)** - Holds time-series (observations/forecasts) and a **history** table for query logs; uses a **PVC** so **data survives restarts**.
- **Observability** — Structured logs and basic metrics: **ingestion lag, API latency, cache hit rate, and HPA replicas**. This is the evidence for correct operation, self-healing, and scaling.

2.3 Technology Choices (initial, for Deliverable 1)

- Frontend (web UI): React + Leaflet (Map rendering)
- API (gateway/service layer): Python FastAPI
- HORI Calculator (risk compute): Python (FastAPI microservice)
- Routing: OSRM / openrouteservice client
- Ingestion (data pulls): Python (requests + schedule/CronJob)
- Database (persistent store): PostgreSQL (optionally TimescaleDB)
- API Sidecar (proxy/logging): nginx or Envoy
- Kubernetes (CloudLab): Deployments/Services, ClusterIP for internals, NodePort/LB for Frontend, PVC for DB, HPA on API, liveness/readiness probes.
- CI/CD: GitHub Actions → build images → push to registry → rolling update (Helm/Kustomize or kubectl set image) → smoke tests.
- Observability: structured logs + /metrics endpoints (basic gauges/counters); verify with kubectl top/HPA status.

2.3 Implementation flow (from zero to demo)

1. **Cluster & images**
   Bring up the CloudLab cluster (one control node, two workers). Build container images for **frontend, API, HORI, ingestion, rou**ting (and DB if containerized); push to a registry the cluster can pull from.
2. **Manifests**
   Deploy each service as its own Deployment/Service. Expose **only** the frontend publicly (NodePort/LoadBalancer). Keep API/HORI/ingestion/routing/data store as **internal** services. The API pod contains **api + sidecar** containers.
3. **State & resilience**
   Bind a **persistent volume** to the data store; add **readiness/liveness** probes; enable **HPA** for the API so it scales under load.
4. **Security**
   Protect programmatic endpoints under v1 with **basic auth** (Kubernetes Secrets). The frontend proxies requests so users never handle credentials.
5. **CI/CD**
   A GitHub Actions workflow builds, pushes, and rolls out updates (Helm/Kustomize or kubectl set image). After deployment it runs a health check (/healthz) and a sample point query as a smoke test.
6. **Demo path**
   - Update a small frontend text → watch the rolling update.
   - Tweak a HORI weight → route colors change with no downtime.
   - Delete a pod → it self-heals; history remains after restart.
   - Apply light load → API scales via HPA.
   - CI/CD multi-component demo: rolling update across API → Frontend with smoke tests to prove zero-downtime.