Delhi AQI Intelligence Platform

Comprehensive Technical Report

Real-Time Air Quality Monitoring, ML Forecasting,
Dynamic Insights & RAG-Powered AI Advisory System

Author: Prachi Tewari
Repository: github.com/Prachi-Tewari/delhi-aqi-dashboard

February 2026

Table of Contents

Chapter 1: Project Overview & Architecture

1.1 Problem Statement

Delhi, India consistently ranks among the world's most polluted cities. PM2.5 concentrations routinely exceed the WHO 24-hour guideline of 15 ug/m3 by 10-20x, contributing to an estimated 6-10 year reduction in life expectancy for Delhi residents. Existing air quality information systems suffer from: (i) delayed or infrequent updates, (ii) no short-term forecasting, (iii) raw numeric outputs without health context, and (iv) no conversational interface for public engagement.

1.2 Solution: Delhi AQI Intelligence Platform

An end-to-end system with six integrated modules:

1. Data Ingestion -- Live data from 15+ CPCB/DPCC stations via OpenAQ v3 API
2. AQI Computation -- India NAQI-compliant index with per-pollutant sub-indices
3. ML Forecasting -- Regime-aware LightGBM ensemble for 6-hour recursive AQI prediction
4. Dynamic Insights -- Rule-based trend alerts, health advisories, anomaly detection
5. RAG AI Assistant -- Conversational AI (Llama 3.3 70B) with hybrid retrieval and hallucination guards
6. Streamlit Dashboard -- Professional UI with Plotly visualizations

1.3 Technology Stack

Component	Technology	Version/Detail
Language	Python	3.14
Web Framework	Streamlit	1.54.0
ML: Gradient Boosting	LightGBM	4.6.0
ML: Comparison Model	XGBoost	3.2.0
Explainability	SHAP	0.50.0
Vector Search	FAISS	CPU
Embedding Model	BAAI/bge-base-en-v1.5	768-dim
Reranker	ms-marco-MiniLM-L-6-v2	Cross-encoder
LLM	Llama 3.3 70B	via Groq API
Visualization	Plotly	Interactive charts
Data Source	OpenAQ v3 API	REST, API key auth
PDF Generation	fpdf2	Report module

1.4 System Architecture

The system follows a pipeline architecture with six layers:

OpenAQ v3 API (15+ Stations)

```
   |
   v
[1. Data Ingestion] -> Station Discovery -> Sensor Deduplication -> Hourly Aggregation
   |
   v
[2. AQI Engine] -> Sub-Index Calculation (6 pollutants) -> AQI = max(sub-indices) -> Category
   |
   +---> [3. ML Forecasting] -> 58 Features -> Regime Clustering (K=3) -> LightGBM -> 6h Forecast
   +---> [4. Dynamic Insights] -> Trend Alerts, Health Advisories, Anomaly Detection
   +---> [5. RAG Assistant] -> FAISS+BM25 -> Reranker -> Llama 3.3 70B -> Hallucination Guard
   |
   v
[6. Streamlit Dashboard] -> Live AQI | Forecast | Stations | AI Chat | Visualizations
   |
   v
End User
```

1.5 Directory Structure

```
aqi_rag_system/
  app.py                    # Main Streamlit dashboard (1204 lines)
  api/
    openaq_client.py        # OpenAQ v3 API client (592 lines)
  forecasting/
    features.py             # Feature engineering (196 lines)
    train.py                # Training pipeline (578 lines)
    inference.py            # Live inference (446 lines)
    models/                 # Saved model artifacts
  rag/
    retriever.py            # Hybrid retrieval (190 lines)
    reranker.py             # Cross-encoder reranker (142 lines)
    llm_pipeline.py         # Groq LLM interface (175 lines)
    hallucination.py        # Hallucination detection (100 lines)
    confidence.py           # Response confidence scoring (212 lines)
    query_classifier.py     # Intent classification (169 lines)
    prompt_template.py      # Prompt construction (84 lines)
    memory.py               # Conversation memory (175 lines)
    tool_calling.py         # Live data tool calls (201 lines)
  insights/
    engine.py               # Dynamic insights generator (432 lines)
  visualization/
    plots.py                # Plotly charts & AQI logic (1179 lines)
  embeddings/
    vector_store.py         # FAISS + BM25 hybrid store (226 lines)
    cache.py                # Embedding cache
  ingestion/
    load_pdfs.py            # PDF loader
    chunk_docs.py           # Text chunking
    embed_docs.py           # Document embedding
```

Chapter 2: Data Ingestion Layer (OpenAQ API Client)

2.1 OpenAQ v3 API Overview

OpenAQ is an open-source platform aggregating government air quality data from 65+ countries. Our client (api/openaq_client.py, 592 lines) interfaces with the v3 REST API using an API key passed via the X-API-Key header. Base URL: https://api.openaq.org/v3

2.2 Location Discovery

The list_locations() function performs a geo-radius search centered on India Gate (28.6139N, 77.2090E) with a 25 km radius. It filters for stations active within the last 30 days by checking each location's datetimeLast field. Up to 100 locations are returned, each with a sensors list containing {id, parameter, units}.

```
params = {
    "coordinates": "28.6139,77.209",
    "radius": 25000,
    "limit": 100,
    "iso": "IN"
}
```

2.3 Sensor Deduplication

Critical discovery: Many Delhi stations have both legacy (defunct) and current sensors for the same pollutant parameter. Legacy sensors have lower IDs and return empty results, wasting API quota. Our deduplication strategy keeps only the sensor with the highest ID per parameter per location, ensuring we query the newest, active hardware.

```
# Deduplication logic:
best_sensor = {}
for sensor in loc['sensors']:
    param = sensor['parameter']
    sid = sensor['id']
    if param not in best_sensor or sid > best_sensor[param]['id']:
        best_sensor[param] = sensor
```

2.4 Measurements Endpoint (Critical Fix)

We discovered that the /sensors/{id}/hours endpoint returns pre-computed hourly aggregates that are often stale (data from 2016!) and IGNORES date_from/date_to parameters. The fix was to switch to /sensors/{id}/measurements with datetime_from filtering, which returns raw measurements with correct date filtering. We then aggregate to hourly averages client-side.

Endpoint: /sensors/{id}/measurements?datetime_from=<ISO>&limit=1000

Client-side aggregation: group by floor(timestamp, 1h) and parameter, compute mean.

2.5 Fallback Synthesis

When the API fails or returns empty data, _demo_hourly() generates synthetic data anchored to the last known live readings. It uses a diurnal pattern with morning and evening rush-hour peaks, Gaussian noise (~8% of base), and shifts the final value to match the actual live reading. This ensures the dashboard never shows a blank state and the forecast uses consistent values.

2.6 Rate Limiting & Safety

- Maximum 50 API calls per refresh cycle
- Process up to 10 locations (most relevant by proximity)
- Skip parameters once we have 4x the needed hourly data
- 25-second timeout per request
- Safety timestamp filter: discard data older than (hours + 1) from now

2.7 Other Endpoints

get_latest_city_measurements() -- Most-recent value from each sensor. Uses /locations/{id}/latest which returns {sensorsId, value, datetime} but NO parameter name. We build a sensorId->param map from the location's sensors list.

get_historical_city() -- Daily aggregated data via /sensors/{id}/days for trend analysis.

get_station_latest() -- Per-station readings for the station detail view.

list_city_stations() -- Simplified station list for UI selectors.

Chapter 3: AQI Computation Engine (India NAQI Standard)

3.1 India National Air Quality Index

The AQI is defined by CPCB (Central Pollution Control Board) as the maximum sub-index across six criteria pollutants:

AQI = max(I_PM2.5, I_PM10, I_NO2, I_SO2, I_CO, I_O3)

Each sub-index I_p is computed via piecewise linear interpolation on CPCB breakpoint tables:

I_p = I_lo + ((I_hi - I_lo) / (C_hi - C_lo)) * (C_p - C_lo)

where C_p is the pollutant concentration and (C_lo, C_hi), (I_lo, I_hi) are the enclosing breakpoint pair.

3.2 NAQI Breakpoint Tables

AQI Range	PM2.5	PM10	NO2	SO2	CO (mg/m3)	O3
0-50	0-30	0-50	0-40	0-40	0-1.0	0-50
51-100	31-60	51-100	41-80	41-80	1.1-2.0	51-100
101-200	61-90	101-250	81-180	81-380	2.1-10	101-168
201-300	91-120	251-350	181-280	381-800	10.1-17	169-208
301-400	121-250	351-430	281-400	801-1600	17.1-34	209-748
401-500	251-380	431-510	401-520	1601-2100	34.1-46	749-940

3.3 Unit Conversion Logic

OpenAQ sensors report in various units (ug/m3, ppb, ppm, mg/m3). The _convert_to_ugm3() function handles all conversions:

- CO is unique: NAQI breakpoints are in mg/m3 (not ug/m3)
- ppb to ug/m3 conversion factors: NO2=1.88, SO2=2.62, CO=1.145e-3, O3=1.96
- Special CO handling: OpenAQ often labels CO as 'ppb' when it's actually ppm. If CO 'ppb' value < 100, we treat it as ppm (real ppb would be 500-10000 for urban sites)
- ppm to mg/m3: CO_PPM_TO_MGM3 = 1.145

3.4 24-Hour Average Computation

India NAQI uses 24-hour average concentrations. The _compute_24h_averages() function:
1. Prefers hourly data (24h median across stations) when available
2. Falls back to median of latest readings when hourly data is absent
3. Median is used instead of mean for robustness against outlier stations

3.5 AQI Categories & Health Advisories

AQI	Category	Health Guidance
0-50	Good	Safe for outdoor activities
51-100	Satisfactory	Sensitive groups limit exertion
101-200	Moderate	Reduce heavy outdoor exercise
201-300	Poor	Everyone reduce outdoor activity
301-400	Very Poor	Avoid outdoor, use air purifier
401-500	Severe	Stay indoors, N95 mask, seek help

Chapter 4: ML Forecasting System

4.1 Dataset

Delhi AQI Combined 2020-2024 dataset: 43,848 hourly observations across 42 columns from CPCB continuous ambient air quality monitoring stations.

Variables:
- 6 criteria pollutants: PM2.5, PM10, NO2, SO2, CO, O3
- 3 auxiliary pollutants: NO, NOx, NH3
- 7 meteorological parameters: temperature, humidity, wind speed, wind direction, rainfall, solar radiation, barometric pressure
- AQI and category labels

Preprocessing:
- Column standardization via RAW_COL_MAP dictionary
- Hourly resampling using median aggregation
- Linear time-interpolation for gaps <= 6 hours
- Winsorization at 0.1% and 99.9% quantiles
- Duplicate timestamp removal (keep last)

4.2 Feature Engineering (58 Features)

All features are strictly causal -- no future information leakage. The build_features() function applies transformations in a fixed order.

4.2.1 AQI Lag Features (7 features)

Lag horizons: h in {1, 2, 3, 6, 12, 24, 168} hours.
aqi_lag1 captures immediate autoregressive dynamics.
aqi_lag24 captures daily periodicity.
aqi_lag168 captures weekly patterns.

4.2.2 Pollutant Lag Features (14 features)

Lag-1 and lag-3 for each of 7 key pollutants (pm25, pm10, no2, so2, nh3, co, o3). These capture multi-pollutant temporal dynamics beyond the aggregate AQI.

4.2.3 Rolling Statistics (6 features)

For windows w in {3, 6, 24} hours, compute rolling mean and standard deviation of AQI. Windows are shifted by 1 step (df[target].shift(1).rolling(...)) to prevent target leakage. The 24h rolling std captures volatility.

4.2.4 Rate-of-Change Features (4 features)

First-order differences: delta1 = AQI(t) - AQI(t-1), delta3, delta6.
Second-order acceleration: accel = delta1(t) - delta1(t-1).
These capture momentum and inflection points.

4.2.5 Temporal Features (10 features)

Calendar: hour, day-of-week, month, is_weekend.
Cyclical encoding using sin/cos to preserve periodicity:
hour_sin = sin(2*pi*hour/24), hour_cos = cos(2*pi*hour/24)
dow_sin/cos (period=7), month_sin/cos (period=12).

4.2.6 Meteorological Features (17 features)

Raw: temperature, humidity, wind_speed, wind_dir, rainfall, solar_rad, pressure.
Raw pollutants: pm25, pm10, no2, so2, nh3, co, o3, no, nox.
Derived:
- Wind vector decomposition: u = speed*sin(dir), v = speed*cos(dir)
- Temperature-humidity interaction: T*RH/100 (atmospheric stability proxy)
- Solar-temperature product: SR*T/1000 (photochemical activity for O3)

4.3 Pollution Regime Clustering

A key innovation: We identify distinct pollution regimes using unsupervised learning.

Process:
1. Aggregate hourly data to daily features: AQI mean/std/max/min/range, PM2.5/PM10 means, temperature/humidity/wind_speed means
2. StandardScaler normalization
3. KMeans clustering (k=3, n_init=10, random_state=42)
4. Map daily regime labels back to hourly via forward-fill

Result: Three regimes emerge naturally:

Regime	Hourly Observations	Mean AQI	Characterization
0 (Clean)	12,984	~106	Monsoon, post-rain, good dispersion
1 (Moderate)	7,808	~231	Transition, dust storms, post-Diwali
2 (Severe)	5,280	~349	Winter inversions, stubble burning

4.4 Model Architecture

4.4.1 LightGBM Configuration

LightGBM is a gradient boosted decision tree (GBDT) framework. Unlike neural networks, GBDTs do not use epochs -- they build trees sequentially. Each tree corrects the residual errors of all previous trees.

Parameter	Value	Purpose
boosting_type	GBDT	Standard gradient boosting
n_estimators	2000	Max trees (early stopped at ~50 patience)
num_leaves	127	Complexity per tree (2^7 - 1)
learning_rate	0.05	Step size per tree
feature_fraction	0.8	Random 80% features per tree (regularization)

bagging_fraction	0.8	Random 80% samples per tree
bagging_freq	5	Subsample every 5 iterations
min_child_samples	20	Min samples per leaf (prevents overfitting)
reg_alpha	0.1	L1 regularization
reg_lambda	0.1	L2 regularization

4.4.2 Regime-Aware Ensemble

The final model is an ensemble of per-regime LightGBM models:

1. Training: Separate LightGBM is trained on data from each regime
2. Inference: Input data is classified into a regime via the KMeans model, then the corresponding specialized model generates the prediction
3. Fallback: If a regime has <100 training or <20 validation samples, the global model is used instead

This architecture lets each model learn distinct feature-target relationships for each atmospheric condition -- the relationship between wind speed and AQI is fundamentally different during winter inversions vs. monsoon.

4.4.3 Training Protocol

Strict time-based split (no random shuffling, prevents temporal leakage):
- Train: 2020-2022 (26,072 hours)
- Validation: 2023 (8,605 hours) -- used for early stopping only
- Test: 2024 (8,784 hours) -- completely held out, never seen during training

The validation set determines when to stop adding trees (early stopping patience=50). If validation error hasn't improved for 50 consecutive trees, training halts. This prevents overfitting while keeping model complexity optimal.

4.5 Results

4.5.1 One-Step Prediction Performance (Test Set, 2024)

Model	MAE	RMSE	R2	Spike MAE
Persistence	2.04	3.3	0.9992	2.09
LightGBM_Global	0.53	0.92	0.9999	1.31
XGBoost_Global	0.5	0.86	0.9999	1.22
LightGBM_Regime	0.49	1.53	0.9998	0.66

The regime-aware LightGBM achieves the best overall MAE (0.49) and dramatically outperforms all baselines on spike events (Spike MAE = 0.66 vs 1.22 for XGBoost and 2.09 for persistence). This is the primary motivation for regime clustering: high-AQI events follow fundamentally different dynamics.

4.5.2 Six-Hour Recursive Forecast Performance

Horizon	MAE	RMSE	R2	Spike MAE
+1h	2.32	3.45	0.9991	2.72
+2h	3.86	5.77	0.9974	4.46

+3h	5.15	7.65	0.9954	5.83
+4h	6.28	9.22	0.9934	6.61
+5h	7.53	11.28	0.9901	6.94
+6h	8.57	13.31	0.9862	7.8

Error growth is approximately linear (~1.3 MAE per hour). Even at the 6-hour horizon, R2 > 0.986. For a city where AQI ranges 50-500, a 6-hour MAE of 8.57 represents a relative error of <3% -- sufficient for actionable public health decisions.

4.5.3 SHAP Feature Importance

Rank	Feature	Mean |SHAP|
1	aqi_lag1	78.1352
2	aqi_rmean3	11.1571
3	aqi_lag2	8.4884
4	aqi_delta1	1.6958
5	aqi_rmean6	1.4987
6	aqi_delta3	0.8959
7	aqi_lag3	0.2975
8	aqi_delta6	0.1198
9	pm25	0.1096
10	aqi_rmean24	0.0852

aqi_lag1 dominates with a mean |SHAP| of 78.14, confirming strong autoregressive behavior. The 3-hour rolling mean (11.16) and aqi_lag2 (8.49) provide significant complementary signal. Rate-of-change features (delta1=1.70) capture momentum. Raw PM2.5 at rank 9 (0.11) shows the model also leverages current pollutant composition beyond AQI.

4.6 Recursive Multi-Step Forecasting Logic

The recursive_forecast() function predicts AQI for +1h to +6h:

```
For each step h = 1, 2, ..., 6:
  1. Extract feature vector from the last row of the working DataFrame
  2. Predict: y_hat = model.predict(X_input)
  3. Create a new row for timestamp t+1:
     - Set AQI(t+1) = y_hat
     - Update all lag features: lag1 = AQI(t), lag2 = AQI(t-1), ...
     - Recompute rolling mean/std over the extended window
     - Update delta1 = y_hat - AQI(t)
     - Advance temporal features (hour, day, cyclical encodings)
  4. Append the new row to the working DataFrame
  5. Repeat from step 1 with updated data
```

Each step feeds predictions from prior steps, so errors compound -- but our linear error growth profile (+1.3 MAE/hour) shows this is well-controlled.

4.7 Live Inference Pipeline

The inference module (forecasting/inference.py) bridges offline-trained models with real-time API data:

1. Format detection: Auto-detect long (OpenAQ) vs wide format
2. Parameter normalization: Map 'pm2.5'->'pm25', 'nitrogen dioxide'->'no2', etc.
3. Pivot: Long to wide format with hourly averaging across stations
4. AQI computation: Per-hour AQI from NAQI breakpoints. Current AQI overrides last hour.
5. Feature construction: Full 58-feature pipeline. Missing features zero-filled.
6. Recursive prediction: 6-hour forecast with feature updates at each step
7. Post-processing: Trend classification (worsening/stable/improving based on 10% threshold), confidence (high/medium/low based on data hours), uncertainty bands (15% at +1h, 40% at +6h), natural language summary

4.8 Uncertainty Quantification

Prediction intervals: sigma_h = (0.10 + 0.05*h) * y_hat
This produces 15% bands at +1h growing to 40% at +6h. The heuristic was calibrated against recursive evaluation residual distributions. Future work: conformal prediction or quantile regression for rigorous statistical coverage guarantees.

Chapter 5: RAG-Powered AI Assistant

5.1 Architecture Overview

The AI assistant combines Retrieval-Augmented Generation with live data injection:

```
User Query
  -> Query Classifier (intent detection)
  -> Hybrid Retrieval (FAISS dense + BM25 sparse)
  -> Cross-Encoder Reranking (ms-marco-MiniLM-L-6-v2)
  -> Prompt Construction (question + RAG context + live data + forecast)
  -> Llama 3.3 70B (via Groq API)
  -> Hallucination Detection
  -> Confidence Scoring
  -> Response with badges and citations
```

5.2 Query Classification (query_classifier.py)

Keyword-based intent classifier with 7 categories:

- live_data: 'current', 'right now', 'today' -> needs API, skip RAG
- health_advice: 'safe', 'mask', 'exercise' -> needs both API + RAG
- historical: 'trend', 'past year' -> needs RAG, skip API
- comparison: 'compare', 'vs', 'WHO' -> needs both
- factual: 'what is', 'explain', 'how does' -> needs RAG
- policy: 'government', 'GRAP', 'ban' -> needs RAG
- general: catch-all

Each intent maps to a config dict specifying: needs_api, needs_rag, top_k, preferred_topics (for retrieval boosting).

5.3 Embedding & Vector Store

Embedding Model: BAAI/bge-base-en-v1.5 (768 dimensions)
Query Prefix: 'Represent this sentence for searching relevant passages: '
Embedding Cache: In-memory LRU (4096 entries) with NumPy NPZ disk persistence.

Vector Store (vector_store.py):
- FAISS IndexFlatL2 for dense retrieval (GPU-free, exact search)
- Custom BM25 implementation (Okapi BM25, k1=1.5, b=0.75) for sparse retrieval
- No external BM25 dependency -- 60-line pure Python implementation
- Metadata stored as JSON (text, topic, year, credibility_score per chunk)

5.4 Hybrid Search

The retrieve() method combines dense and sparse retrieval:

1. Embed query using BGE model
2. FAISS search: top-20 by L2 distance (dense_weight=0.6)
3. BM25 search: top-20 by keyword relevance (sparse_weight=0.4)
4. Merge and deduplicate candidates
5. Optional topic boost: if intent prefers certain topics (e.g., 'health'), boost matching candidates by 1.3x

5.5 Cross-Encoder Reranking (reranker.py)

Cross-encoder model: cross-encoder/ms-marco-MiniLM-L-6-v2

Process:
1. Form (query, passage) pairs for all candidates
2. Score each pair with the cross-encoder (produces a logit)
3. Normalize scores to [0,1] using sigmoid
4. Combine with source reliability: final = 0.85*ce_score + 0.15*credibility
5. Adaptive top-k: return results above min_score=0.15, up to top_k

This two-stage retrieve-then-rerank architecture is computationally efficient -- the expensive cross-encoder only scores 20 candidates instead of the entire corpus.

5.6 LLM Generation (llm_pipeline.py)

Model: Llama 3.3 70B Versatile (via Groq Cloud API)
Temperature: 0.4 (balanced creativity/accuracy)
top_p: 0.9
Max tokens: 1500

System prompt (SYSTEM_MESSAGE) establishes the AI as a world-class air quality intelligence analyst with expertise in: India NAQI standard, Delhi pollution sources (vehicular 40-50%, stubble burning, industry), seasonal patterns, health science (PM2.5 alveolar penetration, COPD, IHD), policy context (GRAP stages, NCAP targets), WHO guidelines, and measurement science.

10 response rules enforce: answer only what's asked, use markdown, include hyperlinks to authoritative sources (WHO, CPCB, IQAir, PubMed), cite [Ref N] for knowledge base, match tone to question complexity.

5.7 Live Data Context Injection

The system prompt is augmented at runtime with:

1. Current AQI data: All pollutant values, sub-indices, category, dominant pollutant, cigarette equivalent
2. Forecast data: 6-hour predictions with trend, confidence, per-hour AQI and category
3. Dynamic insights: Top 4 current insights (trend alerts, health advisories)

This ensures the LLM always has current factual context, enabling accurate answers to questions like 'Is it safe to jog right now?' without hallucinating numbers.

5.8 Hallucination Detection (hallucination.py)

Post-generation analysis checks:
1. Citation check: If RAG context was provided, does the response contain [Ref N] citations?
2. Numeric claims: More than 3 specific numbers without source attribution -> warning
3. Hedging language: 'I think', 'probably', 'maybe' -> uncertainty signal
4. Combined risk: low/medium/high

If risk is HIGH and RAG context was available, the system automatically regenerates the response with a stricter prompt that forces source citation. This prevents the LLM from making up statistics about Delhi pollution.

5.9 Confidence Scoring (confidence.py)

Each response gets a 0-1 confidence score from four dimensions:

1. Source quality: Average reranker scores + credibility of retrieved sources
2. Citation coverage: How well the response cites available sources
3. Relevance: Does the response address the query? (keyword overlap heuristic)
4. Response quality: Length, structure, specificity

Weights vary by intent (e.g., factual queries weight source quality higher). Grade: high (>=0.7), medium (>=0.4), low (<0.4). Displayed as a badge under each response.

5.10 Conversation Memory (memory.py)

The ConversationMemory class manages multi-turn context:

- Max token budget: 6,000 tokens
- Keep recent: 6 messages verbatim
- Older messages: Automatically summarized using the LLM itself
- Summarization prompt asks for key facts, AQI values, and user preferences

This prevents the conversation from exceeding the LLM context window while preserving important information from earlier turns.

5.11 Prompt Template (prompt_template.py)

The build_prompt() function assembles the user message:

1. User's question (front and centre)
2. Intent hint for calibrating response style
3. Tool-call results (if live API data was fetched)
4. Live data snapshot (current AQI readings)
5. RAG knowledge-base excerpts with [Ref N] labels, sources, and relevance scores

6. Lightweight instructions: answer what's asked, use citations, include links

5.12 Tool Calling (tool_calling.py)

When the query classifier detects 'live_data' intent, the auto_tool_call() function:

1. Checks if a cached snapshot exists (from app.py's current computation)
2. If yes, packages it as a tool result (avoids redundant API calls)
3. If no, calls get_current_aqi() to fetch fresh data from OpenAQ

The tool result is formatted as structured context for the LLM prompt.

Chapter 6: Dynamic Insights Engine

6.1 Overview

The insights engine (insights/engine.py, 432 lines) generates real-time, context-aware analyses WITHOUT LLM calls -- ensuring low latency and deterministic outputs. Each insight is a dataclass with: category, severity (info/warning/critical), title, body, and priority score for display ordering.

6.2 Insight Categories

1. Trend Alerts: Detects >25% pollutant concentration changes in 3-hour windows. Compares recent 3h mean vs earlier 3h mean for PM2.5, PM10, NO2, O3.

2. Diurnal Patterns: Time-of-day contextualization based on IST hour:
   - 7-10 AM: Morning rush hour warning
   - 17-21: Evening pollution peak (boundary layer collapse)
   - 23-04: Nighttime inversion alert
   - 11-15: Midday mixing (best outdoor window)

3. WHO Comparisons: Current readings vs WHO 2021 guidelines. Highlights 5x/10x exceedances. Includes cigarette equivalent (PM2.5 / 22 = daily cigarettes).

4. Forecast Notes: Extracts highlights from ML forecast -- worsening/improving predictions, category transitions, pollutant-specific rising trends.

5. Station Insights: Identifies spatial variation -- when worst station reads 2x+ the best station for PM2.5 or PM10, flags hyperlocal sources.

6. Health Context: Population-specific advisories for respiratory patients, cardiac risk, outdoor exercise, sensitive groups. Severity escalates with AQI.

7. Anomaly Detection: (via station comparisons) Stations deviating significantly from city mean.

6.3 Priority System

Each insight gets a priority score (0-100+):
- AQI emergency (>300): priority 100
- WHO 10x exceedance: priority 95
- AQI worsening forecast: priority 85
- Trend spike (>25%): priority 80 + pct_change
- WHO 5x exceedance: priority 75
- Respiratory risk: priority 70-90
- Evening peak: priority 60

Insights are sorted by priority descending and capped at max_insights (default 6).

Chapter 7: Visualization Module

7.1 Chart Functions (plots.py, 1179 lines)

All charts are interactive Plotly figures with a consistent professional theme:

1. aqi_gauge() -- Semicircular gauge with AQI value, colored by category
2. sub_index_chart() -- Horizontal bar chart of all 6 sub-indices
3. pollutant_vs_who() -- Grouped bar: current value vs WHO guideline (red highlight for exceedance)
4. timeseries_plot() -- Time series for any pollutant with date range
5. pollutant_radar() -- 6-axis radar chart of normalized sub-indices
6. aqi_scale_bar() -- Reference color bar with category labels
7. station_comparison_chart() -- Bar chart comparing stations for a selected pollutant
8. station_aqi_heatmap() -- Heatmap: stations x pollutants, colored by sub-index severity
9. station_detail_chart() -- Per-station Plotly figure with all parameters
10. forecast_chart() -- 6-hour forecast line with AQI category background bands and confidence intervals

7.2 Forecast Chart Design

The forecast_chart() function creates a Plotly figure with:
- AQI category background bands (Good=green through Severe=red) as horizontal shapes
- Confidence interval as a filled area (upper/lower bounds)
- Forecast line: dashed blue with circle markers
- Current AQI: diamond marker at hour 0
- Hover data showing AQI value, category, and confidence range

Chapter 8: Dashboard Application (app.py)

8.1 Overview

The main application (app.py, 1204 lines) is a Streamlit wide-layout dashboard that orchestrates all modules. Key design principles:
- Professional analytics aesthetic (no emojis)
- Inter + JetBrains Mono fonts
- CSS animations: fadeInUp, pulse, shimmer, barGrow
- Responsive design (mobile breakpoints at 768px)

8.2 Data Flow

1. Sidebar: API key inputs, city/country selection, history days slider
2. Data fetch: Three cached (@st.cache_data, TTL=600s) functions fetch latest, hourly, and station data
3. AQI computation: _compute_24h_averages() -> compute_aqi() -> classify
4. Snapshot: Build key-value dict of all readings for chat context
5. Insights: generate_insights() with current AQI, pollutants, hourly data
6. Forecast: forecast_next_6_hours() with hourly data, current AQI, pollutant values

8.3 UI Sections (Top to Bottom)

1. Navigation Bar: Title, live dot, station count, timestamp
2. Hero Banner: Large AQI number, gradient background colored by category, LIVE badge, PM2.5 value, dominant pollutant
3. AQI Scale Strip: 6-segment color bar with category labels
4. Pollutant Cards: 6 cards showing value, sub-index, animated progress bar
5. Key Metrics: Cigarettes/day equivalent, AQI with category, WHO exceedance multiplier
6. Health Advisory: Category-specific guidance + condition-specific cards (asthma, cardiac, allergies, COPD) + recommended actions pills
7. Dynamic Insights: Grid of insight cards sorted by priority
8. 6-Hour AQI Forecast: 3 KPI cards (trend, next-hour, confidence) + Plotly forecast chart + details table
9. Station-Wise Monitoring: 3 tabs (comparison bar chart, heatmap, station detail drill-down)
10. Detailed Analysis: 4 tabs (gauge+radar overview, WHO comparison, trend charts, raw data table)
11. AI Chat: Full conversational interface with RAG, citations, confidence badges, response details
12. Footer: Data source credits and technology attributions

8.4 Chat Integration

The chat system integrates all components:
1. User types a question
2. _retrieve_rag(): Classify intent -> retrieve from vector store (if needs_rag)
3. _execute_tool_call(): Check if live data fetch needed (if needs_api)
4. Build context: System prompt + live snapshot + forecast + insights

5. Build user message: Question + RAG refs + tool results
6. Conversation memory: Trim/summarize history to fit token budget
7. LLM call: chat_with_guard() -> generate + hallucination check + confidence score
8. Display: Markdown answer + confidence badge + expandable details (intent, chunks, hallucination risk)
9. Source citations: If RAG was used, show expandable list of referenced documents
10. Fallback: If LLM fails, _rule_answer() generates a template-based response from live data

Chapter 9: Debugging & Problem Resolution

9.1 API Date Filtering Bug (Critical)

Problem: Predictions were 'a bit off' -- forecast showed AQI ~300 when live reading was ~105.

Root Cause Investigation:
1. Wrote _test_api.py to examine raw API responses
2. Discovered get_hourly_data() used /sensors/{id}/hours endpoint
3. This endpoint returns pre-computed aggregates from 2016-2025(!)
4. It completely IGNORES date_from and date_to parameters
5. The forecast model was trained on recent data but receiving 8-year-old stale data

Fix:
- Switched to /sensors/{id}/measurements endpoint
- This endpoint supports datetime_from filtering correctly
- Added client-side hourly aggregation (group by floor(timestamp, 1h), compute mean)
- Added safety cutoff filter: discard data older than requested window

Verification: After fix, all 6 pollutants present, correct date range (last 14-24h), sensible forecast (AQI 99-114 from current 105).

9.2 Missing PM10 Bug

Problem: PM10 data was missing from API results despite stations having PM10 sensors.

Root Cause: Sensor deduplication was not implemented, so the API client queried legacy (defunct) sensors first, consuming the 50-call rate limit before reaching active PM10 sensors.

Fix: Implemented per-parameter deduplication -- for each location, keep only the sensor with the highest ID (newest hardware) per parameter. This ensured active sensors were queried first, and PM10 data appeared in results.

9.3 Column Name Mismatch

Problem: features.py expected 'timestamp' but the CSV had 'Timestamp' (capital T).

Fix: Added case-insensitive handling in load_and_clean():

```
  if 'Timestamp' in df.columns:
      df.rename(columns={'Timestamp': 'timestamp'}, inplace=True)
```

9.4 LightGBM libomp Dependency (macOS)

Problem: LightGBM import failed with 'libomp not found' on macOS.

Fix: brew install libomp

LightGBM uses OpenMP for parallel tree building. macOS does not ship with libomp by default (unlike Linux which includes it in most distros).

9.5 Unicode in PDF Generation

Problem: generate_report.py crashed with FPDFUnicodeEncodingException on em dashes.

Root Cause: fpdf2's built-in Helvetica/Courier fonts use latin-1 encoding, which does not support Unicode characters like em dash (--), curly quotes, or Greek letters.

Fix: Created _safe() helper that replaces all Unicode characters with ASCII equivalents before rendering. Maps: -- -> --, curly quotes -> straight quotes, Greek letters -> names, subscript numbers -> regular numbers.

9.6 CO Unit Confusion

Problem: CO sub-index was wildly wrong for some stations.

Root Cause: OpenAQ v3 reports CO as 'ppb' but the values (0.5-5.0) are clearly in ppm. True CO ppb for urban Delhi would be 500-10,000.

Fix: Added heuristic threshold: if CO 'ppb' value < 100, treat as ppm and convert using factor 1.145 (ppm to mg/m3). Otherwise, use standard ppb conversion.

9.7 Target Leakage Prevention

Problem: Initial rolling features used current row, leaking the target into inputs.

Fix: All rolling windows shifted by 1 step:
```
  df[target].shift(1).rolling(window=w, min_periods=1).mean()
```

This ensures only past information is used. Same principle applied to all features: lags use shift(h), deltas use diff(h), no future data anywhere.

9.8 Stale Streamlit Cache

Problem: Dashboard showed old data after API fix.

Fix: @st.cache_data has TTL=600s (10 minutes). During development, the old cached results persisted. Killing the Streamlit process and restarting cleared the cache. Added 'Clear cache' button in sidebar for production use.

Chapter 10: Deployment & Configuration

10.1 Environment Setup

Python 3.14 virtual environment:
  python -m venv .venv
  source .venv/bin/activate
  pip install -r requirements.txt

macOS dependency: brew install libomp (for LightGBM)

10.2 Required API Keys

1. OpenAQ API Key: Free at https://explore.openaq.org -> Sign up -> API Keys
   Used for: X-API-Key header on all OpenAQ v3 requests

2. Groq API Key: Free at https://console.groq.com
   Used for: Llama 3.3 70B chat completions

Keys can be provided via:
- Environment variables: OPENAQ_API_KEY, GROQ_API_KEY
- Streamlit sidebar text inputs
- Streamlit Cloud secrets (.streamlit/secrets.toml)

10.3 Running the Dashboard

```
# Launch with environment variables:
OPENAQ_API_KEY="<key>" GROQ_API_KEY="<key>" \
  .venv/bin/streamlit run app.py --server.port 8501
```

10.4 Training the Forecasting Model

```
# Run the training pipeline:
python -m forecasting.train \
  --data /path/to/Delhi_AQI_Combined_2020_2024.csv

# Outputs saved to forecasting/models/:
#   best_model.pkl, feature_cols.json,
#   regime_scaler.pkl, regime_kmeans.pkl, regime_models.pkl,
#   training_results.json, shap_importance.json/png
```

10.5 Key Dependencies (requirements.txt)

Package	Purpose
streamlit	Web dashboard framework

plotly	Interactive visualizations
pandas, numpy	Data manipulation
lightgbm	Gradient boosting (primary model)
xgboost	Gradient boosting (comparison)
shap	Model interpretability
scikit-learn	KMeans, metrics, preprocessing
faiss-cpu	Dense vector search
sentence-transformers	BGE embeddings + cross-encoder
requests	HTTP client for APIs
fpdf2	PDF report generation
joblib	Model serialization

10.6 GitHub Repository

Repository: https://github.com/Prachi-Tewari/delhi-aqi-dashboard
Branch: main
License: Open source

The repository includes all source code, trained model artifacts, SHAP analysis outputs, and this documentation.