SE 4485: Software Engineering Projects

Fall 2025

Detailed Design Documentation

|  |  |
| --- | --- |
| Group Number | Group 1 |
| Project Title | County Level Air Quality Prediction Application |
| Sponsoring Company | Raytheon (Team A) |
| Sponsor(s) | Ryan Havens <Ryan.Havens@rtx.com>,  Marc Perna <marc.perna@rtx.com>,  Trey Williams <trey.williams@rtx.com>,  Trevor Lang <trevor.a.lang@rtx.com> |
| Students | 1. Jay Chung <cwc130330@utdallas.edu>  2. Amelia Quinn <qcb220000>  3. Kevin Melo <ksm220005>  4. AJ Kimbrough <ank210005>  5. David Santos <des210001>  6. Andrew Enright <ame210008> |

**ABSTRACT**

This document presents the detailed design for the County Level Air Quality Prediction (CLAP) web application. The CLAP application predicts next-day county-level Air Quality Index (AQI) categories using historical data provided by the U.S. Environmental Protection Agency (EPA). Building upon the previously defined software architecture, this document focuses on the data structure, component behavior, sequence of operations, and traceability from design to functional and non-functional requirements. The design defines subsystem interactions through well-defined interfaces to support manual data ingestion, preprocessing, model inference, dashboard visualization, and main user interaction features.

**TABLE OF CONTENTS**

1. INTRODUCTION 3
2. GUI (GRAPHICAL USER INTERFACE) DESIGN 4
3. STATIC MODEL (CLASS DIAGRAMS) 6
4. DYNAMIC MODEL (SEQUENCE DIAGRAMS) 8
5. RATIONALE FOR YOUR DETAILED DESIGN MODEL 9
6. TRACEABILITY FROM REQUIREMENTS TO DESIGN MODEL 11
7. EVIDENCE OF CONFIGURATION MANAGEMENT 13
8. ENGINEERING STANDARDS AND MULTIPLE CONSTRAINTS 14
9. ADDITIONAL REFERENCES 14

**LIST OF FIGURES**

Figure 3.1 - CLAP - Class Diagram 4

Figure 4.1 - CLAP - Sequence Diagram 4

**LIST OF TABLES**

Table 7.1 - GitHub - Single File Revision 13

Table 7.2 - GitHub - Difference Link 13

**INTRODUCTION**

This document presents the detailed design model for the County Level Air Quality Prediction (CLAP) web application. Its primary purpose is to define the internal structure, component behavior, and data interactions that realize the system’s components. The design expands the software architecture previously established to provide a blueprint for implementation, integration, and testing.

The CLAP system is a predictive analytics web application that forecasts next-day Air Quality Index (AQI) categories at the county level. It ingests historical AQI data from the U.S. Environmental Protection Agency (EPA), processes and stores it in a local persistence layer, and uses a trained LightGBM machine-learning model to generate predictions. The detailed design describes how the frontend, backend, data layer, and dashboard components interact through defined interfaces to deliver next-day AQI forecasts.

The CLAP detailed design implements a modular, layered client-server approach that ensures clear separation of concerns among presentation, application logic, and data management. The design emphasizes modularity, enabling lightweight local deployment while supporting educational objectives in software architecture, machine learning, and system integration.

The remainder of this document is organized as follows: graphical user interface (GUI) design, static and dynamic models, rationale for the detailed design model, traceability from requirements to design, configuration management, relevant standards and constraints, and references.

**GUI (GRAPHICAL UESR INTERFACE) DESIGN**

GUI (Graphical User Interface) Design

* screen designs (coded or using drawing tool)

The Graphical User Interface (GUI) is designed for clarity, accessibility, and responsiveness to meet WCAG 2.1 AA standards.

* Clarity & Accessibility: Meet WCAG 2.1 AA (e.g. color contrast, keyboard navigation, focus order, aria-labels).
* Consistency: Use a unified visual language (e.g. Tailwind utility classes, CLAP color palette mapped to AQI categories).
* Feedback & Resilience: Loading states and error banners.
* Observability: In-UI surfacing of model metrics and timestamps.

Screens & Components:

* Dashboard (e.g. *App.jsx):*
  + Header with logo, model selector, “Generate Forecast” & “Load Data” buttons.
  + County Selector (e.g. *Select.jsx*): Dropdown with label/value pairs.
  + Stats Strip (e.g *StatsStrip.jsx*): Displays MSE/RMSE/R2 and last update time.
  + Prediction Card (e.g. *PredictionCard.jsx*): Displays predicted AQI value & category chip.
  + Category Probabilities (e.g. *ProbabilitiesList.jsx*): Category bars (e.g. Good, Moderate, Unhealthy for Sensitive Groups, Unhealthy, Very Unhealthy, and Hazardous) with relative bar container.
  + Historical Chart (e.g. *AqiChart.jsx*): 30-day line chart.
  + Multi-Day Forecast (e.g. *MultiDayChart.jsx*): next-N bars/lines based on model output.
  + Alerts (e.g. *Spinner.jsx, ErrorAlert.jsx*): Action/Error Feedback.
* Visual Specs*:*
  + Typography: System UI stack; titles (text-x1/2x1), body (text-sm/base).
  + Spacing & Layout: Cards (rounded-2x1) with (shadow-md), (grid) + (gap-4/6).
  + Color Tokens (e.g. *src/utils/aqi.js*)
    - Good: #00E400
    - Moderate: #FFFF00
    - Unhealthy for Sensitive Groups: #FF7E00
    - Unhealthy: #FF0000
    - Very Unhealthy: #8F3F97
    - Hazardous: #7E0023
  + Responsiveness: Cards wrap, charts reflow.

STATIC MODEL

CLASS DIAGRAMS

* captured in Rose (other tools are also allowed)

The logical structure is layered: Presentation (e.g. React) 🡪 Frontend - Application/API (Flask) 🡪 Data & ML Artifacts (e.g. Feature Engineering, Models, repositories) 🡪 Persistence (CSV).

Backend Integration:

* App Creation: *backend/app.py*
* Data Loading: *backend/data\_source.py*
* Model Inference: *backend/ml\_model.py*
* API Routes: *backend/routes/\*py*
* Frontend Connection: React/Vite app (e.g. *VITE\_API\_BASE\_URL=localhost:5001/api*) set in *frontend/.env* .

Backend Classes / Modules (e.g. Python, backend/):

* *config.py* :
  + Loads configuration values from the environment using *python-dotenv* (e.g. load\_dotenv()).
  + Defines application configuration constraints (e.g. SECRET\_KEY & DEBUG flags, AQI category ranges and colors stored in Config.AQI\_CATEGORIES), and default paths for local CSV data & model artifacts.
  + Acts as single source of truth for system-wide settings used by other backend files.
* *App.py* :
  + Entry point for creating and configuring the Flask application.
  + Creates app instance (e.g. *app = Flask(\_\_name\_\_, static\_folder="../frontend/dist", static\_url\_path="/")*), ensuring that the complied frontend (e.g. Vite build in *frontend/dist/*) is served as the static dashboard.
  + Enables Cross-Origin Resource Sharing (CORS) to allow API calls from the React/Vite frontend (e.g. *from flask\_cors import CORS; CORS(app)*).
  + Registers all blueprints defined in *backend/routes/* using *register\_blueprints(app).*
  + Configures structured logging using *logging* and *RotatingFileHandler*:
    - Logs are written to *logs/app.log* with a JSON formatter.
    - A custom *log\_event(level, msg, \*\*kwargs)* helper is registered under *app.extensions*.
  + Falls back to serving *index.html* from the static folder for non-API routes, allowing deep linking within the Reach single-page app (SPA).
* *data\_source.py* :
  + Provides helper functions for reading AQI data from CSV files.
  + Loads cleaned EPA data into Pandas DataFrames.
  + Supports filtering by county, state, and date range.
  + Used indirectly through utility functions defined in *routes/aqi\_utils.py*
* *ml\_model.py* :
  + Handles machine-learning model operations:
    - Loads trained LightGBM model files (*.pkl* artifacts).
    - Performs prediction and probability calculations for AQI categories.
    - Provides feature-processing helpers used by multiple route modules.
  + Contains functions used by *routes/predict.py* and *routes/refresh.py*.
* Flask Blueprints & Controllers (e.g. *routes/*):

|  |  |
| --- | --- |
| **File:** | **Endpoints / Purpose:** |
| aqi\_utils.py | Provides shared utilities used across routes:   * *logger()* and *log\_event()* wrappers for structured logs. * *ds()* to access the data source (*data\_source.py*). * *predictors()* and *get\_predictor()* for model access. * *Build\_features\_from\_recent()* and *iterative\_forecast()* for feature creation and forecasting logic. |
| categories.py | GET */categories* 🡪 Returns the AQI category configuration defined in *Config.AQI\_CATEGORIES*. |
| counties.py | GET */counties* 🡪 Returns a list of all counties derived from the dataset. |
| errors.py | Registers centralized error handlers (e.g. 404, 500) for consistent API responses. |
| health.py | GET */health* 🡪 Returns system status JSON including timestamps and model-loaded flag. |
| historical.py | GET */aqi/historical…* 🡪 Returns recent AQI values for the specified county and state (30-days by default). |
| index.py | GET / 🡪 Serves *frontend/dist*/*index.html* |
| model\_metrics.py | GET */model/metrics…*  🡪 Returns stored model performance metrics (e.g. MSE, RMSE, and R2). |
| predict.py | POST */aqi/predict* 🡪 Accepts JSON *{county, state, model, days}* and returns next-day AQI category predictions. |
| refresh.py | POST */aqi/refresh* 🡪 Runs a combined refresh that regenerates historical data and predictions in a single pipeline. |

Frontend Components (e.g. React, Vite, frontend/src/):

* Top Level:
  + *App.jsx* : Composition root and main dashboard logic. Manages state for county selection, model selection, prediction results, probabilities, historical data, loading / error flags, and wires button handlers to API calls (e.g. *lib/api.js*). Renders the page using React/Vite and global styles (e.g. *main.jsx, index.css, app.css*).
* Components:
  + *Select.jsx* : Dropbox input for County/State (e.g. */counties*). Updates *App* state.
  + *StatsStrip.jsx :* Displays model / summary metadata (e.g. metrics or timestamps) returned from the backend upon request.
  + *PredictionCard.jsx* : Shows next-day predicted AQI value and its category label. Rendered inline in *App.jsx*.
  + *ProbabilitiesList.jsx* :Renders the six AQI category bars (e.g. Good, Moderate, etc…). Uses color classes from *utils/aqi.js.*
  + *AqiChart.jsx* : Line chart of recent AQI history for the selected county / state (via */aqi/historical*).
  + *AqiLegend.jsx :* Small legend or mapping of AQI categories to colors (with *aqi.js*).
  + *ErrorAlert.jsx* : Inline error banner used when AQI calls fail.
  + *Spinner.jsx* : Loading indicator shown while waiting API responses.
* Utilities & API:
  + *utils.aqi.js* : Central AQI category metadata. Used by *ProbabilitiesList.jsx*, *AqiLegend.jsx*, and etc.
  + *lib/aqi.js* : Thin fetch later around the backend. Uses *import.meta.env.VITE\_API\_BASE\_URL* (from *frontend/.env*).
    - *getCounties() 🡪* GET */counties*
    - *getHistorical({ county, state, days }) 🡪* GET */aqi/historical?county=…&state=…&days=…*
    - *postPredict({ county, state, model, days }) →* ***POST*** */aqi/predict*
    - *getModelMetrics(model) →* GET */model/metrics?model=…*
* Environment:
  + Flask: *frontend/.env*
  + React/Vite: *VITE\_API\_BASE\_URL=http://localhost:5001/api* (used by *lib/api.js*), matching Flask server’s port and */api* prefix (default port: 5173).

Buttons:

* Generate Forecast:
  + Frontend Component(s):
    - Click handler: *Frontend/src/App.jsx*
    - Results Displayed (Components):
      * *frontend/src/component/PredictionCard.jsx* : Predicted AQI + Category.
      * *frontend/src/component/ProbabilitiesList.jsx* : Category probabilities.
      * *frontend/src/component/MultiDayChart.jsx* : (Unused) For forecasting multi-day predictions.
    - API Function used:
      * *frontend/src/lib/api.js 🡪 postPredict({ county, state, model, days })*
  + Backend Route Hit:
    - *backend/routes/predict.py 🡪* POST */aqi/predict*
    - Body (JSON): { *“county”: “…”, “state”: “…”, …* }
    - Response: single-day { *prediction, probs, …* }, or multi-day { predictions: […]}.
* Load Data:
  + Frontend Component(s):
    - Click handler: *Frontend/src/App.jsx*
    - Results Displayed (Components):
      * *frontend/src/component/AqiChart.jsx* : Historical series
      * *frontend/src/component/StatsStrip.jsx* : Metrics/time info.
    - API Function used:
      * *frontend/src/lib/api.js 🡪 getHistorical({ county, state, days })*
      * Calls *getCounties()* on first load, calls *getModelMetrics(model)* to populate stats.
  + Backend Route Hit:
    - *backend/routes/historical.py 🡪* GET */aqi/historical?county=...&state=...&days=N*
    - Initial Data / Filters: *backend/routes/counties.py* → GET */counties*

DYNAMIC MODEL

SEQUENCE DIAGRAMS

* captured in Rose (other tools are also allowed)

RATIONALE FOR YOUR DETAILED DESIGN MODEL

TRACEABILITY FROM REQUIREMENTS TO DETAILED DESIGN MODEL

* provide a mapping between requirements and detailed design model
* clearly describe how each requirement in the *Requirements Documentation* is captured in the design

**EVIDENCE THE DESIGN MODEL HAS BEEN PLACED UNDER CONFIGURATION MANAGEMENT**

The team has selected GitHub as the configuration tool for this project. The tables below provide evidence of configuration management by recording version history, authorship, and reviews of document changes. The *ID* column identifies each entry. The *date of change* column indicates when a modification was made to an existing file, and the v*ersion (before & after)* columns include the associated Git commit hash for distinction. The *author* column refers to the author of the new version. The *difference link* column provides a URL to the GitHub comparison view between two consecutive commits. The format of the difference link is as follows:

“https://github.com/cchung7/rtx\_team1/compare/<ver-before-hash>..<ver-after-hash>”.

**Table 7.1 - Single file revision**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID:** | **Date of Change:** | **Version Before:** | **Version After:** | **Author:** | **Review -Change Summary:** | **Reviewers:** |
| 1 | 10/20/25 | v0.1 () | v0.2 () | Jay Chung (cwc130330) |  | All Team Members |
| 2 |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |

**Table 7.2 - Difference link**

|  |  |
| --- | --- |
| **ID:** | **Difference Link:** |
| 1 | https://github.com/cchung7/rtx\_team1/compare/ |
| 2 |  |
| 3 |  |
| 4 |  |

**ENGINEERING STANDARDS AND MULTIPLE CONSTRAINTS**

Engineering Standards:

* IEEE Std 1016-1998-(Revision-2009): Software Design [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1016-1998-(Revision-2009)-Software-Design.pdf)]

Multiple Constraints:

* Project may utilize one data set so long as multiple fields are used to train the predictive analytics model.
* Frontend must conform to WCAG 2.1 Level AA accessibility requirements for visual content (e.g. SC 1.4.1 “Use of Color”, SC 1.4.3 “Contract (Minimum)”, SC 1.4.11 “Non-text Contrast”).

**ADDITIONAL REFERENCES**

[1] Larman, C., 2012. Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development. Pearson Education

[2] Hyman, B., 1998. Fundamentals of Engineering Design. New Jersey: Prentice Hall

[3] Simon, H.A., 2014. A Student's Introduction to Engineering Design: Pergamon Unified Engineering Series (Vol. 21). Elsevier