Department of Computer Science & Engineering
The University of Texas at Arlington

Detailed Design Specification
CSE 4317: Senior Design II
Summer 2025

Forecast Tx
Severe Weather Prediction Model

Noe Sanchez
Kevin Simbakwira
Kristal Phommalay
Kenil Patel
Mason Berry

Revision History

Revision	Date	Author(s)	Description
0.1	6.18.2025	NS, MB, KP, KP, KS	document creation
1.0	7.23.2025	NS, MB, KP, KP, KS	Updated for Closeout

CONTENTS

List of Figures

1 INTRODUCTION

The severe weather prediction model is a system designed to forecast the likelihood of sever weather events occurring in Texas over the next 5 to 10 years. The model consist of severe main components: a machine learning-based prediction model and a responsive web application. Developed for use by State Farm's risk management and claims teams, the system aims to help anticipate future sever weather patterns and mitigate potential losses. The model uses time series forecasting techniques including neural networks, linear regression, and LSTM, trained on historical data from reliable sources like NOAA and Stored in a PostgreSQL database on The Cloud

The accompanying web application, built with React.js and hosted on Google Cloud firebase application, provides an interactive interface for both general and advance users. General users can explore predictions through graphs, maps, and filters for specific weather types, while advance users have access to detailed model insights, input data, and feature selection. This dual-access approach ensures flexibility and transparency in the understanding of the model's forecasts. By allowing the parameter input and supporting multiple devices, the application enhances accessibility and usability for a broad audience within State Farm.

2 SYSTEM OVERVIEW

The data flow of the severe weather prediction model is organized into four main layers: Data Storage, Data ETL, Machine Learning, and User Interface. These layers form a robust,modular architecture that supports the predictive capabilities and user interactivity. At a high level, data originates from external sources and is ingested through the ETL (Extract, Transform, and Load) layer, where it is cleaned, formatted, and prepared for use. This processed data is then stored in the data storage layer, which ensures it remains secure, traceable, and readily accessible for downstream task. The storage layer acts as the foundation of the system, housing both raw inputs and the results of predictive modeling for consistent and scalable data access.

Once the data is available, the Machine Learning Layer retrieves it for model training, evaluation, and forecasting. This layer serves as the systems's analytical core, applying a range of algorithms to generate accurate weather predictions. The output from this layer is passed to the user interface, which allows users to interact with the data through intuitive visualizations, filters, and controls. Whether viewing forecast or managing data inputs , the UI layer ensures a smooth experience. Altogether, these layers form a streamlined pipeline from raw data acquisition to actionable insights, enabling informed decisions making for risk management and climate preparedness.

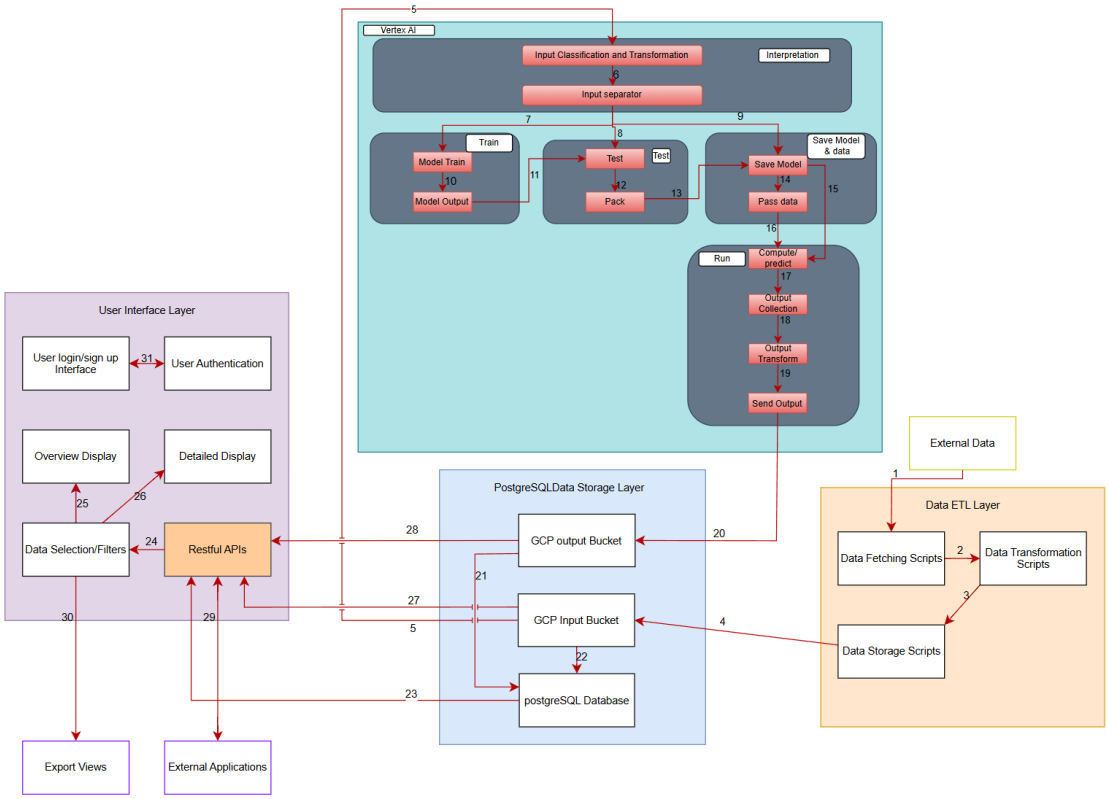

Figure 1: System architecture

3 Data Storage Subsystems

The Data Storage Layer is implemented as a cloud-based solution utilizing the Google Cloud Platform (GCP) for all data storage and management functions. It consists of two primary storage components: GCP Input and Output Buckets, and a PostgreSQL database with PostGIS extensions for spatial data support. These components form the backbone of the data infrastructure, providing persistent, scalable, and geospatially aware storage of both raw input data and processed model outputs.

All storage operations are handled via secure, authenticated API interactions using Python scripts, with GCP IAM policies governing access. This layer is designed to maintain data traceability, version control, and availability for downstream processing and UI layers.

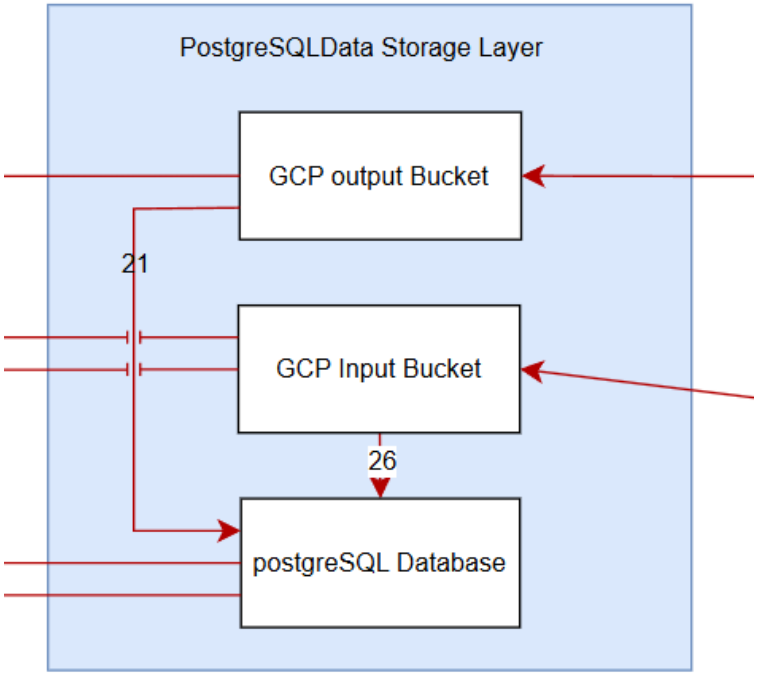

Figure 2: Data Storage Architecture Diagram

3.1 Layer Hardware

There are no physical on-premises hardware requirements. All storage services are hosted in the cloud using managed GCP infrastructure. Virtualized resources in Google's data centers ensure high availability, redundancy, and scalability. Internally, storage operations leverage distributed file systems and SSD-backed storage for optimized I/O performance.

3.2 Layer Operating System

As a fully managed cloud storage and database service, this layer does not depend on a specific operating system from the user's perspective. However, virtual machines accessing these services (e.g., for ETL or ML purposes) typically run Ubuntu 24.04.2 LTS.

3.3 Layer Software Dependencies

- `google-cloud-storage`: Python client library for managing cloud buckets and file transfers.
- `psycopg2 / SQLAlchemy`: PostgreSQL database access and management from Python scripts.
- `PostGIS`: Extension to PostgreSQL enabling spatial queries, geohashing, and spatial indexing for weather event coordinates.
- `gcsfs`: FileSystem interface for treating GCP buckets as file-like objects in Python.
- `dotenv / gcloud SDK`: Securely manage credentials and access configurations.

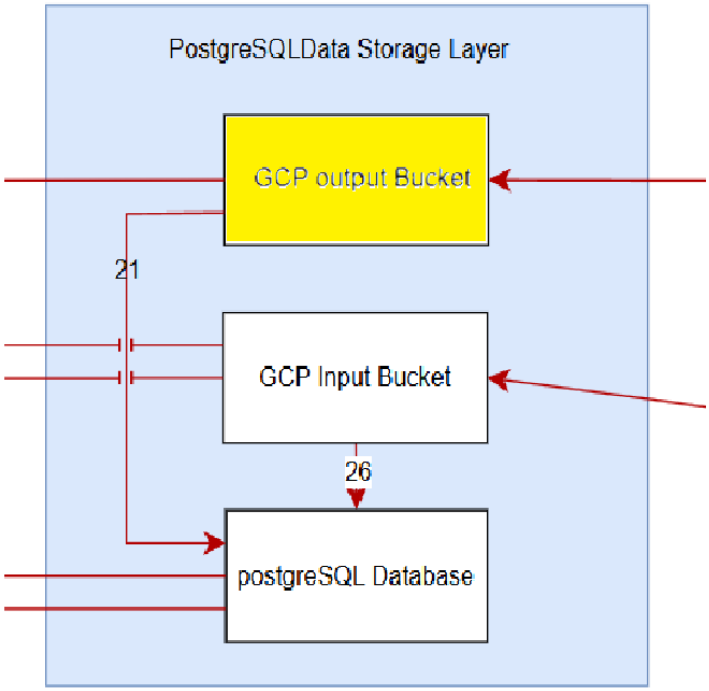

Figure 3: GCP Output Bucket Architecture Diagram

3.4 GCP Output Bucket

The Model Output Data subsystem holds the results of machine learning model predictions. These predictions are saved in CSV or Parquet files, stored in the **GCP Output Bucket**. Each record contains attributes such as:

- `timestamp`
- `latitude`, `longitude`
- `weather_event_type`
- `predicted_severity` or probability

Hardware

No additional hardware is required beyond GCP-managed infrastructure.

Operating System

Ubuntu 24.04.2 LTS is used in the VMs that prepare and upload files to the bucket.

Software Dependencies

`pandas`, `pyarrow`, and `google-cloud-storage` are used for formatting and uploading output files.

Programming Languages

Primarily Python 3.12.3.

Data Structures

Data is tabular with columns such as `[time, lat, lon, event_type]`.

Data Processing

Model output is automatically written to file by Python scripts, then uploaded to the Output Bucket. Optionally, data is inserted into the database using batch SQL inserts for query-ready access.

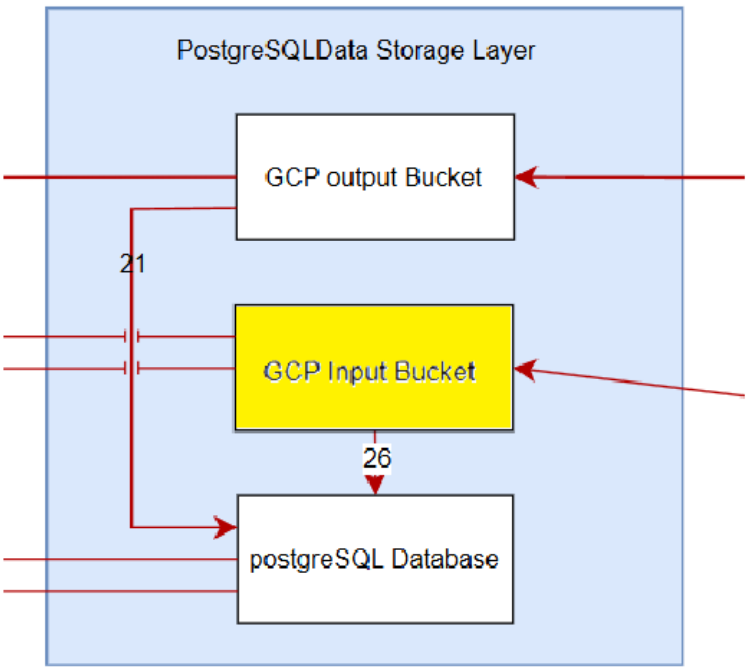

Figure 4: GCP Input Bucket Architecture Diagram

3.5 GCP Input Bucket

This subsystem holds historical weather data acquired from sources like the Copernicus Climate Data Store (CDS) and NOAA. Raw or pre-processed files are uploaded to the **GCP Input Bucket** before transformation.

Hardware

No additional hardware required beyond cloud infrastructure.

Operating System

Ubuntu 24.04.2 LTS is used for data acquisition scripts and upload.

Software Dependencies

`cdsapi`, `cfgrib`, `gcsfs`, and `google-cloud-storage` are used for fetching and storing data.

Programming Languages

Python 3.12.3 for data ingestion, bash scripts for upload coordination.

Data Structures

- Raw GRIB files named as `YYYYMM.grib`.
- Transformed CSVs/Parquets with columns: `[time, lat, lon, temperature, precipitation, etc.]`.

Data Processing

Raw data is fetched monthly, validated, and uploaded. Input buckets serve as transient staging points before processing. Eventually, cleaned and sorted data is uploaded again or inserted into PostgreSQL.

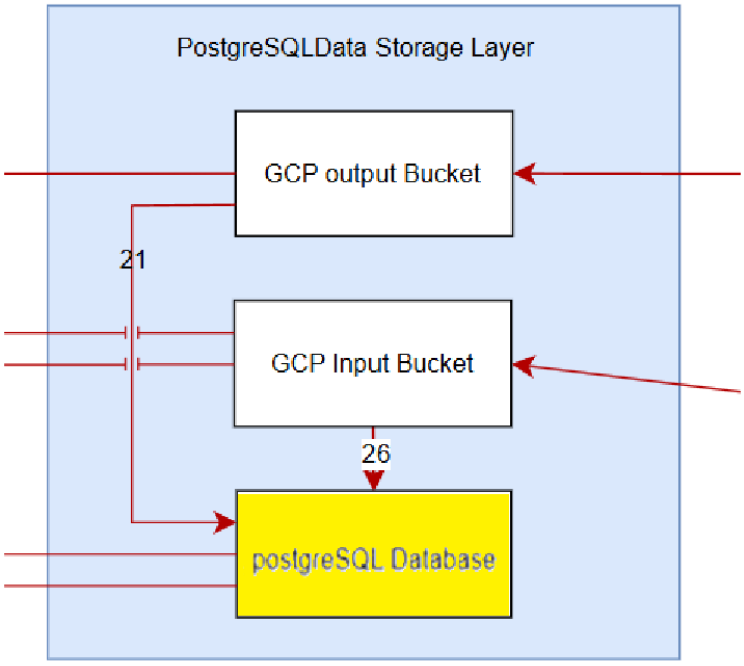

Figure 5: PostgreSQL Database Architecture Diagram

3.6 PostgreSQL Database

The PostgreSQL database contains both input data and model outputs in a structured, relational format with spatial indexing via PostGIS.

Hardware

Fully managed Cloud SQL instance with SSD-backed storage and regional failover.

Operating System

Managed by GCP, abstracted from the user.

Software Dependencies

PostGIS 3.4, PostgreSQL 15+, and client-side SQL tools.

Programming Languages

SQL and Python for queries and automated inserts.

Data Structures

Relational tables:

- `predictions(time TIMESTAMP, lat FLOAT, lon FLOAT, event TEXT, severity FLOAT)`
- `geospatial index` using `geometry(Point, 4326)` for efficient location-based queries

Data Processing

Supports spatial filtering, aggregation, and querying by region or time range. Used to serve filtered data to UI and analytics pipelines.

4 Data ETL Subsystems

In this section, the layer is described in terms of the hardware and software design. The Data ETL Layer is responsible for extracting ERA5 climate reanalysis data from the Copernicus Climate Data Store (CDS), transforming GRIB files into structured CSV/Parquet formats, and loading the processed data into Google Cloud Storage buckets for consumption by the machine learning layer. This layer processes historical weather data from 1955-2024 covering the Texas region at approximately 0.25 degree spatial resolution to support severe weather event prediction modeling for State Farm's predictive analytics requirements.

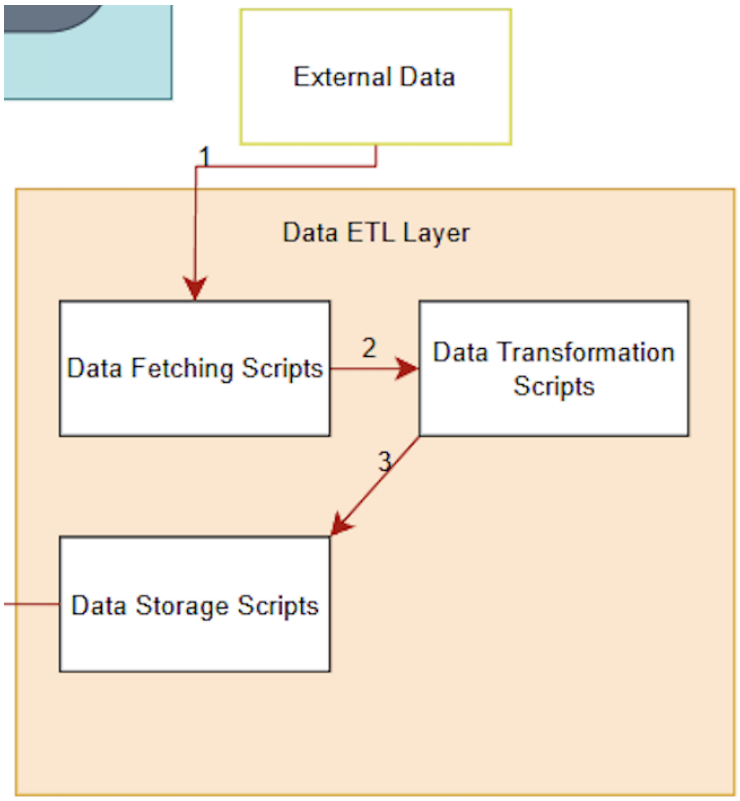

Figure 6: Data Storage and Cloud Upload Process

4.1 Layer Hardware

The layer is designed to run on commodity hardware with configurable resource utilization to accommodate different system capabilities. Minimum requirements include 4GB RAM, 2 CPU cores, and 100GB available storage, while recommended configuration consists of 8GB+ RAM, 4+ CPU cores, and 500GB+ available storage. The current development environment utilizes WSL2 on Windows with 8GB allocated RAM running Ubuntu 24.04.2 LTS. Configurable chunk sizes allow operation on lower-memory systems, and scripts support multi-threading capabilities that can be distributed across multiple machines for improved processing throughput when properly implemented.

4.2 Layer Operating System

The primary operating system is Ubuntu 24.04.2 LTS (Codename: noble), with compatibility across any Linux distribution supporting Python 3.12+ and required dependencies. The system supports container

deployment in WSL2, Docker containers, or native Linux environments, and is compatible with cloud platforms including Google Cloud Compute Engine and AWS EC2.

4.3 Layer Software Dependencies

The system requires Python 3.12.3 with comprehensive package dependencies managed through a virtual environment or conda environment. Core data processing libraries include xarray for multidimensional climate data handling, pandas and numpy for data manipulation and numerical operations, cfgrib and eccodes for GRIB meteorological file processing, and pyarrow for efficient Parquet file operations. Cloud integration dependencies include google-cloud-storage for Google Cloud Storage operations and gcsfs for filesystem interface compatibility. Additional processing libraries encompass dask for parallel computing and large dataset handling, netCDF4 and zarr for alternative climate data format support, matplotlib for data visualization, and cdsapi for Copernicus Climate Data Store API integration. System-level dependencies require libeccodes-dev library installation via package manager (Ubuntu: sudo apt-get install libeccodes-dev).

Prerequisites and Setup Requirements: The system requires establishment of a Copernicus Climate Data Store (CDS) account at https://cds.climate.copernicus.eu/ with API key configuration. Users must create a .cdsapirc file in the home directory containing the CDS API URL and personal access credentials (url: https://cds.climate.copernicus.eu/api/v2 and key: username:api-key). Google Cloud Platform access requires authentication setup using gcloud auth login –no-launch-browser followed by project configuration via gcloud config set project your-project-id to enable automated data upload capabilities to designated storage buckets.

4.4 Data Fetching Scripts

The Data Fetching subsystem implements automated retrieval of ERA5 reanalysis data from the Copernicus Climate Data Store (CDS) API through the get_data2.py script. This subsystem downloads GRIB files containing meteorological variables for the Texas region (36.5 degrees North to 25.8 degrees North latitude, -106.6 degrees West to -93.5 degrees West longitude) at hourly temporal resolution. While 28 variables were initially requested, the final dataset includes available variables with consistent hourly data intervals, excluding variables that only provided 12-hour interval measurements to maintain temporal consistency.

4.4.1 Subsystem Hardware

Stable internet connection with sufficient bandwidth for large file downloads, as GRIB files range 50-200MB each. Temporary local storage capacity of 10-50GB required for batch downloads before cloud upload. Download times average 20 minutes per month depending on CDS queue times, which vary based on system load and request frequency.

4.4.2 Subsystem Operating System

Ubuntu 24.04.2 LTS with Python 3.12.3 runtime environment and GCC 13.3.0 compiler support.

4.4.3 Subsystem Software Dependencies

cdsapi for official Copernicus Climate Data Store API client integration, os and calendar from Python standard library for file system operations and date/time handling.

4.4.4 Subsystem Programming Languages

Primary development in Python 3.12.3 for CDS API interaction and data acquisition workflows.

4.4.5 Subsystem Data Structures

API requests structured as JSON objects containing product_type (reanalysis), format (grib), variable arrays, temporal parameters (year, month, day, time), spatial boundaries defined by area coordinates [North, West, South, East], and download_format specifications. File naming follows YYYYMM.grib convention for consistent pipeline integration.

4.4.6 Subsystem Data Processing

Sequential monthly downloads with error handling and resume capability, automatic retry mechanism for failed downloads with exponential backoff, real-time download status reporting with file size validation, and progress tracking throughout the acquisition process. The system supports customizable date ranges and variable selection for flexible data collection requirements.

4.5 Data Transformation Scripts

The Data Transformation subsystem encompasses the complete pipeline for converting GRIB meteorological files into analysis-ready datasets through four integrated components: era5-organized-converter.py for GRIB to CSV conversion, era5-data-joiner.py for consolidating variables, era5-chronological-sorter.py for temporal ordering, and era5-processing-pipeline2.py for orchestrating the entire transformation workflow. This subsystem processes the available meteorological variables that maintain hourly temporal consistency, handling approximately 250GB of historical data spanning 1955-2024 for the Texas region.

4.5.1 Subsystem Hardware

Configurable memory limits supporting 10,000 to 1,000,000+ rows in memory to accommodate systems from 4GB to 32GB+ RAM, intelligent chunking and temporary file management for datasets exceeding system memory, streaming read/write operations optimized for processing large datasets, and automatic chunk size adjustment based on available system memory for both SSD and traditional hard drive storage systems.

4.5.2 Subsystem Operating System

Ubuntu 24.04.2 LTS with Python 3.12.3 isolated virtual environment for dependency management, multi-process execution support, configurable worker pools, and optimization for ext4 file systems with large file support.

4.5.3 Subsystem Software Dependencies

Core processing libraries including xarray, pandas, and numpy for data manipulation, cfgrib and eccodes for meteorological file format handling, pyarrow for Parquet I/O operations, concurrent.futures for parallel processing coordination, and Python standard library modules (os, glob, argparse, logging, datetime, time, gc, subprocess, shutil, sys, re) for system operations, process management, and comprehensive progress tracking with error reporting.

4.5.4 Subsystem Programming Languages

Python 3.12.3 with command-line argument parsing using argparse, structured logging with custom formatters for array data truncation, and multi-process coordination using process pools and futures for scalable data processing workflows.

4.5.5 Subsystem Data Structures

Input GRIB files contain multidimensional arrays with temporal (hourly timesteps), spatial (latitude/longitude grid points at approximately 0.25 degree resolution), and variable dimensions. Intermediate processing creates organized directory hierarchy: processed/YYYY/MM/variable_name/ containing chunked

CSV files. Final output produces unified time-series datasets in joined/YYYY/ directories with consolidated DataFrames containing chronologically ordered timestamps, spatial coordinates, and all available meteorological measurements as individual columns. Standardized DataFrame schema includes time (ISO 8601 datetime), latitude and longitude (decimal degrees with configurable precision), and value columns containing meteorological measurements in appropriate units.

4.5.6 Subsystem Data Processing

The transformation pipeline operates through four sequential stages: (1) GRIB Conversion where multi-dimensional arrays are flattened into tabular format through coordinate expansion, variable extraction using xarray's to_dataframe() method for automatic dimension alignment, data restructuring converting coordinate systems into explicit columns (time, latitude, longitude, value), and temporal chunking processing data in configurable time segments for memory management; (2) Data Joining performing automatic variable discovery through directory structure scanning, metadata analysis with column structure standardization, incremental processing using configurable chunk sizes (10,000-100,000 rows), spatial-temporal alignment on common time/latitude/longitude grids, and left-join strategy preserving all temporal-spatial coordinates; (3) Chronological Sorting including automatic datetime conversion and validation, multi-level sorting (time primary, latitude secondary, longitude tertiary), optional backup creation, format preservation for both CSV and Parquet, and memory-safe chunk-based processing; (4) Pipeline Orchestration providing configurable batch processing (default 10 months), individual file error handling with comprehensive logging, real-time progress tracking, and automatic cleanup of intermediate files. Processing times average approximately 24 hours for 20-year batches with configurable optimization for different hardware capabilities.

4.6 Data Storage Scripts

The Data Storage subsystem handles automated upload of processed datasets to Google Cloud Storage buckets through the upload_year_range_to_gcs.sh shell script. This subsystem manages the final stage of the ETL pipeline by transferring locally processed CSV files to cloud storage for consumption by the machine learning layer, with upload times requiring approximately 10-15 minutes for two decades of processed data.

4.6.1 Subsystem Hardware

Stable internet connection with sufficient upload bandwidth for large file transfers, temporary local storage for processed datasets before cloud migration, and adequate disk I/O performance for concurrent file reading and network transmission operations.

4.6.2 Subsystem Operating System

Ubuntu 24.04.2 LTS with bash shell environment and Google Cloud SDK integration for authenticated cloud storage operations.

4.6.3 Subsystem Software Dependencies

Google Cloud SDK (gcloud) for authentication and project configuration, gsutil command-line tool for efficient cloud storage operations, and bash shell scripting environment for automated file management and upload coordination.

4.6.4 Subsystem Programming Languages

Bash shell scripting for automated cloud upload workflows with error handling and progress reporting capabilities.

4.6.5 Subsystem Data Structures

Input consists of processed CSV files organized in local directory structure by year: BASE_LOCAL_DIR/YYYY/*.csv. Cloud storage destination follows organized hierarchy: gs://BUCKET_NAME/DEST_ROOT/YYYY/filename.csv. The script processes year ranges specified as command-line arguments and maintains file naming conventions for consistent cloud organization.

4.6.6 Subsystem Data Processing

Sequential year-by-year processing through specified date ranges, automatic directory validation and error handling for missing local data, individual file upload with progress tracking and status reporting, organized cloud storage structure creation maintaining temporal hierarchy, local file cleanup post-upload to manage storage resources, and comprehensive logging of upload operations with success/failure status for each processed year.

Future Enhancement Opportunities: The current system architecture supports several optimization pathways including implementation of properly functioning multi-threading capabilities, extension to process additional meteorological variables as desired for expanded analysis, consolidation of processing stages for improved efficiency, distributed processing across multiple machines for enhanced throughput, and adaptation for alternative geographic regions or temporal ranges to support broader severe weather prediction applications.

5 Machine Learning Subsystems

The machine learning system and its subsystems require powerful computation alongside software that allows us to monitor and run our models. In terms of hardware, the machine learning subsystem requires a virtual machine (VM) with a dedicated GPU, with enough resources to accommodate the large amount of data being processed without bottlenecks. In terms of software, the machine learning subsystem also needs specific versions of OS, CUDA (for GPU), Python, and TensorFlow. Along with that, the VMs that are created need to be in a specific region, specifically Iowa, since that is where our data is stored.

5.1 Layer Hardware

The VM / Vertex AI layer hardware requirements: The VM / Vertex AI layer hardware requirements:

- Region set to Iowa
- Machine: n1-standard-4 (4 vCPU and 15 GB of memory) at the minimum
- GPU: 1 NVIDIA T4 or Higher GPU at the minimum
- Disk Type: balance persistent disk or SSD persistent disk (Recommended)
- Disk size 150 GB (Minimum) or higher

5.2 Layer Operating System

This layer requires a specific OS given by GCP for ML purposes

- OS: Deep learning on Linux
- Version: Deep Learning VM for TensorFlow 2.11 with CUDA 11.3 MI 25

5.3 Layer Software Dependencies

The software for this layer is already provided by GCP, along with the OS; no other software needs to be installed. Those are:

- Python 3.9 to 3.12
- Miniconda version 25.3.1 or the Newest version of Miniconda
- TensorFlow 2.11 with GPU
- CUDA 11.3 or higher
- tmux latest version recommended

5.4 INTERPRETATION 1

Interpretation is a layer that is responsible for interpreting the data that is given by the cloud storage/bucket layer. This layer gets the numpy files from the bucket for processing the data. This layer is also responsible for adding flags to the data based on the inputs to determine whether it is a training or a prediction job.

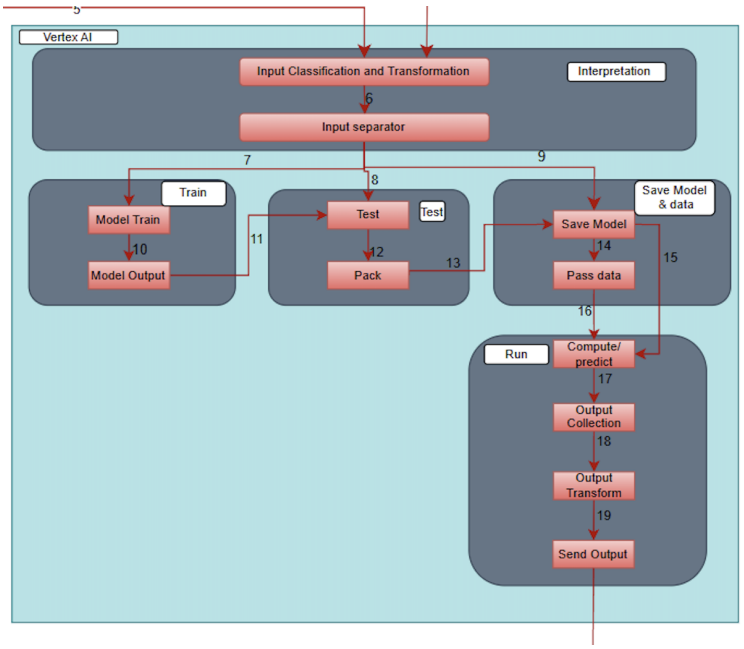

Figure 7: Vertex AI/VM

5.4.1 INTERPRETATION HARDWARE

- Disk storage from VM for caching incoming data

5.4.2 INTERPRETATION OPERATING SYSTEM

Same as the Vertex AI/ VM

- OS: Deep learning on Linux

5.4.3 INTERPRETATION SOFTWARE DEPENDENCIES

- Numpy: For matrix operations
- Pandas: for database operations
- OS library: saving and retrieving files

5.4.4 INTERPRETATION PROGRAMMING LANGUAGES

- Python 3.9 to Python 3.12
- Miniconda version 25.3.1 or the Newest version of Miniconda

5.4.5 INTERPRETATION DATA STRUCTURES

Training/ re-training data coming from buckets are going to CSV files with 20 columns for features and 300K rows per file. Amount of data coming in depends on the training job.

5.4.6 Interpretation Data Processing

The incoming CSV files need to be transformed into Numpy files; this is essential for the model to work. numpy files help the convolution LSTM model do convolutions by grouping and aggregating the data so that learning can be done more efficiently. And each feature related to a specific location can be indexed for faster computing. This also helps reduce the file size.

5.5 Train layer

The train layer is responsible for training the model and also retraining the model. The train layer gets the Numpy files from the Interpretation layer.

5.5.1 TRAIN HARDWARE

- GPU: For training the model
- Disk for storing the output model

5.5.2 TRAIN OPERATING System

- Deep Learning on Linux: same as Vertex AI / VM

5.5.3 TRAIN SOFTWARE DEPENDENCIES

Libraries for Python

- TensorFlow 2.11 with GPU
- CUDA 11.3 or higher: for using and monitoring the GPU
- Numpy: Optimized data input for the model
- keras: a function in Tensorflow for running tests and adding layers
- Math library for calculating errors and other stats
- OS: Python library

5.5.4 TRAIN PROGRAMMING LANGUAGES

- Python 3.9 to 3.12

5.5.5 TRAIN DATA STRUCTURES

The train module requires data to be in numpy files, because the model needs to be able to index a feature based on unique longitude and latitude.

5.5.6 TRAIN DATA PROCESSING

For processing, models need features to be normalized, and the model script does it before training the model; after training, the model will split out a model file with extension ".keras". This file will then be used by the Test module for testing the model. Model training checks for Mean square error and loss, these values are printed out on the terminal while training, for our case, both of these values should be decreasing as the number of epochs (training rounds) increase.

5.6 TEST LAYER

The test layer tests and validates the model using the ".keras" file that was output by the Train layer. This is done to ensure, first, the accuracy of the model on new data (different from training) data, and to obtain real-world results of performance.

5.6.1 TEST HARDWARE

Test is built into the script of Train, but as a separate part; it uses the same hardware as the Train layer, but does not require a GPU.

5.6.2 TEST OPERATING SYSTEM

Test layer uses the same OS as the Train layer

5.6.3 TEST SOFTWARE DEPENDENCIES

Libraries for python

- TensorFlow 2.11 with GPU
- Numpy: For indexing the numpy Test files.
- keras: Testing is done by keras
- Math: For calculating error.
- OS For file retrieval and saving

5.6.4 TEST PROGRAMMING LANGUAGES

Python 3.9 to Python 3.12: same as Train layer.

5.6.5 TEST DATA STRUCTURES

The test module requires a ".keras" file that has the model that TensorFlow can use to evaluate the performance of the model. Test layer also require numpy testing files for testing the model. These files are the same as training in terms of structure but include data that was not used for training the model.

5.6.6 TEST DATA PROCESSING

After testing the model, this layer also packs the model and other log files from testing to send them off to the "Save Model and data" layer. This is necessary because it also adds flags of accuracy for archiving and keeping track of training and testing jobs.

5.7 SAVE MODEL AND DATA

The Save model and Data save, logs, and keep track of all the training jobs that were done; it is also responsible for passing on data to run the model. Save model makes sure that the right model is selected for the right job.

5.7.1 SAVE MODEL AND DATA HARDWARE

This script is part of Vertex AI/ VM, so it can work on that hardware itself.

5.7.2 SAVE MODEL AND DATA OPERATING SYSTEM

Save model and data have the same OS as the Vertex AI and VM.

5.7.3 SAVE MODEL AND DATA SOFTWARE DEPENDENCIES

- Numpy: for adding flags to numpy files
- OS for file retrieval and saving

5.7.4 SAVE MODEL AND DATA PROGRAMMING LANGUAGES

- Python 3.0 or higher
- Bash

5.7.5 SAVE MODEL AND DATA: DATA STRUCTURES

Save model runs a Python script that is waiting for input from Interpretation, so that it can push the data forward and start the run process. I have all the data needed to pull a specific model and send it off to run. These files are ".keras" and CVS files.

5.7.6 SAVE MODEL AND DATA: DATA PROCESSING

Save models and data layer process the incoming request from the interpretation layer and passes on the model files and data to the run module. It processes the request to determine what model to use and what data to give to the run module.

5.8 RUN LAYER

The run module is the most crucial module in the Vertex AI subsystem; it is responsible for predicting, collecting, and outputting the transformed output files. Its other responsibility is to pass on the data to the PostgreSQL data storage layer, for future use and archiving the outputs. It gets 2 inputs from the "Save model and data" layer, the first is the model file itself, and the second is the numpy files. Pulling numpy files from buckets will result in a computational delay.

5.8.1 RUN HARDWARE

Similarly to all the other subsystems, it also runs on the same hardware as the other subsystems; no separate hardware is necessary.

5.8.2 RUN OPERATING SYSTEM

Run works on the same OS as all the other subsystems.

5.8.3 RUN SOFTWARE DEPENDENCIES

- TensorFlow 2.11 with GPU
- Numpy: For indexing the numpy input files.
- Math: For calculating runtime
- OS: for file retrieval and saving
- Pandas: for converting numpy to CSV
- Matplotlib: for plotting data points

5.8.4 RUN PROGRAMMING LANGUAGES

- Python 3.0 or higher

5.8.5 RUN DATA STRUCTURES

Run deals with three types of files: numpy, keras, and CSV. Run module takes the input to the model (keras file) in numpy files, whose output is also a numpy output file with the date, time, and type of run in the file name, and then a CSV for transferring data

5.8.6 RUN DATA PROCESSING

Data processing in the Run modules processes the input data using a Python script and a model file. The output, which is a numpy file, is then processed to convert it into a CSV for data storage and a PostgreSQL database. The final step in data processing is to transfer the CSV to the GCP bucket or PostgreSQL database.

6 User Interface Subsystems

In this section, the layer is described in terms of the hardware and software design. Specific implementation details, such as hardware components, programming languages, software dependencies, operating systems, etc. should be discussed. Any unnecessary items can be omitted (for example, a pure software module without any specific hardware should not include a hardware subsection). The organization, titles, and content of the sections below can be modified as necessary for the project.

6.1 Layer Operating System

The User Interface layer operates as a web-based application that runs within modern web browsers across multiple operating systems. The layer is designed to be platform-agnostic and supports the following operating systems:

- **Desktop Systems**: Windows 10/11, macOS 10.14+, and Linux distributions with modern browser support
- **Mobile Systems**: iOS 12+ and Android 8.0+ for responsive mobile access
- **Server Environment**: The Firebase hosting infrastructure runs on Google's proprietary server operating systems

Cross-platform compatibility is achieved through adherence to web standards and responsive design principles, ensuring consistent functionality across all supported platforms.

6.2 Layer Software Dependencies

The User Interface layer requires the following core software dependencies:

- **React.js 18.x**: Primary frontend framework for component-based UI development
- **D3.js 7.x**: Data visualization library for interactive map rendering and weather data presentation
- **Firebase SDK 9.x**: Client-side SDK for authentication, hosting, and real-time database connectivity
- **Axios 1.x**: HTTP client library for RESTful API communication with the backend services
- **Material-UI 5.x**: Component library providing consistent UI elements and responsive design patterns
- **React Router 6.x**: Client-side routing for single-page application navigation
- **Webpack 5.x**: Module bundler for asset compilation and optimization
- **Babel 7.x**: JavaScript transpiler for cross-browser compatibility

Additional development dependencies include ESLint for code quality, Jest for unit testing, and various polyfills for legacy browser support.

6.3 User Authentication

The User Authentication subsystem manages secure login and access control to the application using Firebase Authentication and OAuth providers. Users can sign in using either Google or Microsoft accounts, and access is restricted to a predefined list of authorized email addresses. This subsystem is implemented in React and is integrated with Firebase services to support real-time authentication and secure session management.

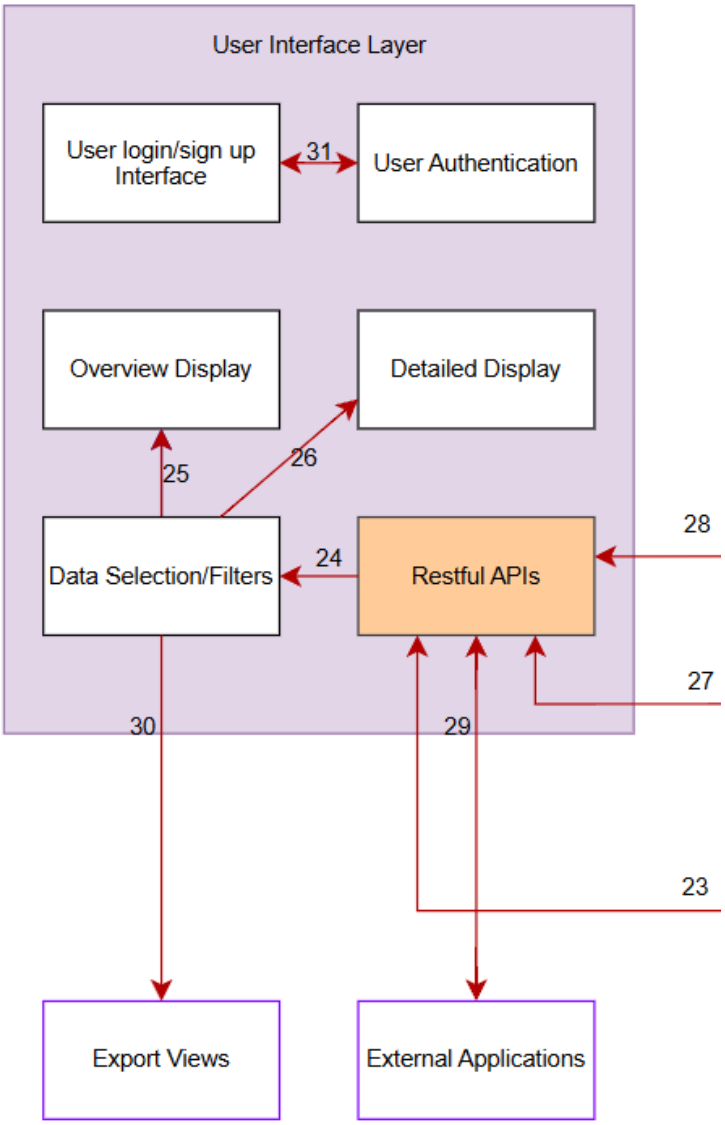

Figure 8: User Authentication subsystem architecture diagram

6.3.1 Subsystem Operating System

The User Authentication subsystem runs in a cross-platform browser environment and does not require any special operating system beyond standard web browser support.

6.3.2 Subsystem Software Dependencies

- **Firebase Authentication SDK**: Provides secure login using Google and Microsoft OAuth providers
- **React Router**: Handles navigation upon successful login
- **React (18.x)**: Used to build the component-based login interface

6.3.3 Subsystem Programming Languages

- **JavaScript (ES6+)**: Core programming language for the authentication logic

- **JSX**: React templating syntax for creating UI components
- **CSS3**: Used for styling the login interface and layout

6.3.4 Subsystem Data Structures

- **User Object**: Retrieved from Firebase, includes user ID, display name, and email
- **Authorized Email List**: Hardcoded array used to determine access eligibility
- **Session State**: Maintained by Firebase to track active users

6.3.5 Subsystem Data Processing

The authentication subsystem executes the following steps during the login flow:

- **OAuth Sign-In Flow**: Initiates a secure pop-up login using either Google or Microsoft as the identity provider
- **Email-Based Access Control**: Validates the user's email against an approved list before granting access
- **Session Termination**: Automatically logs out unauthorized users and displays an alert
- **Client-Side Navigation**: Redirects authorized users to the dashboard upon successful login

6.4 Map Display and Detailed Display

The Map Display subsystem renders an interactive map focused on the state of Texas, featuring animated weather overlays and real-time visualizations of environmental data. Implemented in React using the Leaflet.js ecosystem, this component delivers a rich, user-responsive experience with time-aware data playback and dynamic weather indicators such as wind, hail, and thunderstorms.

6.4.1 Subsystem Operating System

The Map Display subsystem runs entirely within a web browser and is cross-platform compatible. It does not require any additional operating system dependencies beyond a standard modern web environment.

6.4.2 Subsystem Software Dependencies

- **React 18.x**: Component-based framework for dynamic UI rendering.
- **Leaflet 1.8.x**: Base map rendering and interaction handling.
- **leaflet-timedimension**: Adds time dimension playback capabilities for animated data overlays.
- **leaflet-velocity**: Used to render animated wind field vectors.
- **leaflet-fullscreen**: Adds a fullscreen toggle button to the map interface.
- **GeoJSON**: For loading and masking specific US state boundaries (Texas).
- **Stadia Maps Tiles**: External map tile provider for light, dark, and satellite base layers.

6.4.3 Subsystem Programming Languages

- **JavaScript (React/ES6+)**: Logic, state management, and rendering.
- **CSS3**: Styling and responsive layout of the map interface and legend.
- **HTML5**: For structure and use of the Canvas and DOM APIs via Leaflet plugins.

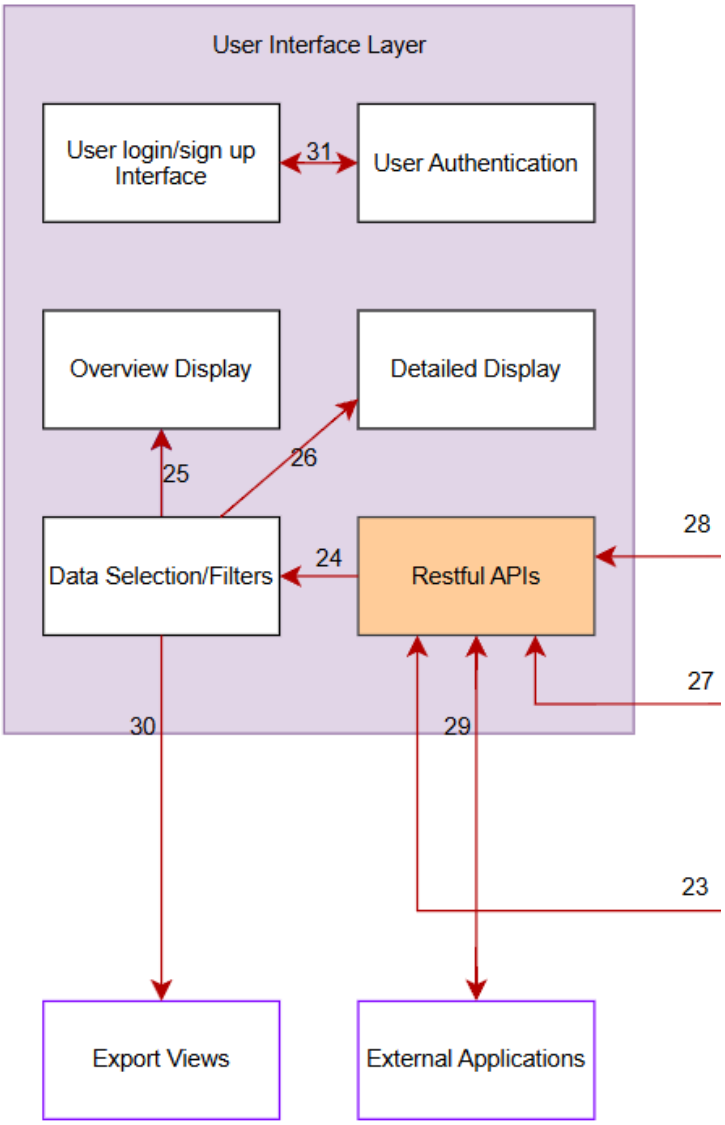

Figure 9: Map Display subsystem architecture diagram

6.4.4 Subsystem Data Structures

- **GeoJSON Features**: Represents Texas boundary and state masking geometry.
- **City Coordinate Arrays**: Coordinates for 20 major cities used to simulate thunderstorm activity.
- **Wind Data Time Series**: Time-indexed wind velocity arrays in meteorological grid format.
- **Layer References**: Refs to dynamically managed Leaflet layers (wind, hail, thunderstorm).

6.4.5 Subsystem Data Processing

The map display subsystem implements the following dynamic processing tasks:

- **Time-Dimension Playback**: A time slider dynamically updates map layers to reflect weather conditions at each hourly interval.

- **Wind Vector Rendering**: Renders wind fields using ‘leaflet-velocity‘ with u/v component arrays.
- **Event-Driven Layer Updates**: React hooks and event listeners ensure map layers reflect changes in time, view, and data layer selection.
- **Map Masking and Borders**: Uses a GeoJSON polygon with holes to mask everything outside of Texas and highlight its border.
- **Storm Animation**: Displays GIF markers on simulated storm centers using Leaflet custom icons.

6.5 Data Filters and Selection

The Data Filters and Selection subsystem enables users to refine displayed weather data using predefined filtering controls: **Year**, **Season**, and **Peril Type**. Implemented using React, this subsystem handles user interactions with dropdown menus and synchronizes the selected values with the application's state to facilitate data queries and visualizations. A reset button is also provided to clear all filters and return to the default view.

6.5.1 Subsystem Operating System

The Data Filters and Selection subsystem operates within a cross-platform browser environment. It does not introduce any additional operating system requirements beyond those of a modern web browser capable of running React applications (e.g., Chrome, Firefox, Edge).

6.5.2 Subsystem Software Dependencies

- **React 18.x**: JavaScript library for building component-based user interfaces.
- **CSS Modules**: Scoped CSS used for styling individual filter components (e.g., dropdowns, buttons).

6.5.3 Subsystem Programming Languages

- **JavaScript (ES6+)**: Implements component logic and handles user input.
- **JSX**: Defines the structure and rendering of React components.
- **CSS3**: Used for styling the filter interface via modular class definitions.

6.5.4 Subsystem Data Structures

- **Component State**: Stores the currently selected values for season, year, and peril using the React `useState` hook.
- **Filter Props**: Each select component receives props for its current value and change handler from the parent.
- **Dropdown Option Arrays**: Static arrays are used to populate dropdowns (e.g., years 2025–2029, perils like `HAIL`).

6.5.5 Subsystem Data Processing

The subsystem currently supports basic filter synchronization and input management:

- **State Synchronization**: When a user changes a filter, the selection is propagated to the parent component using callback functions.

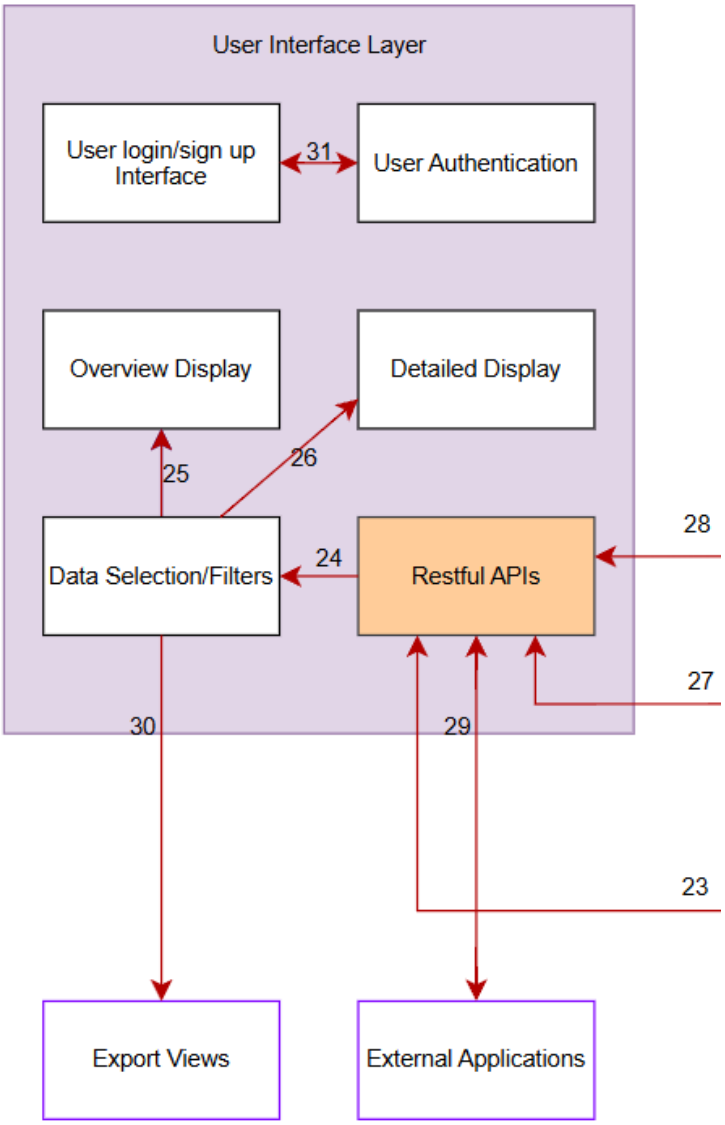

Figure 10: Data Filters and Selection subsystem architecture diagram

- **Dropdown Rendering**: Filters are rendered as native HTML `<select>` elements, populated with predefined option lists.
- **Reset Functionality**: The `Reset Filters` button clears all current selections and restores the default state.

6.6 RESTful API Integration

The RESTful API Integration subsystem manages communication between the frontend interface and backend services hosted through Google Cloud Functions. It is responsible for sending and receiving data via HTTP requests, maintaining API state consistency, and handling errors and retries for robust client-server interaction. This subsystem ensures reliable and secure data exchange between the user interface and cloud-hosted machine learning services.

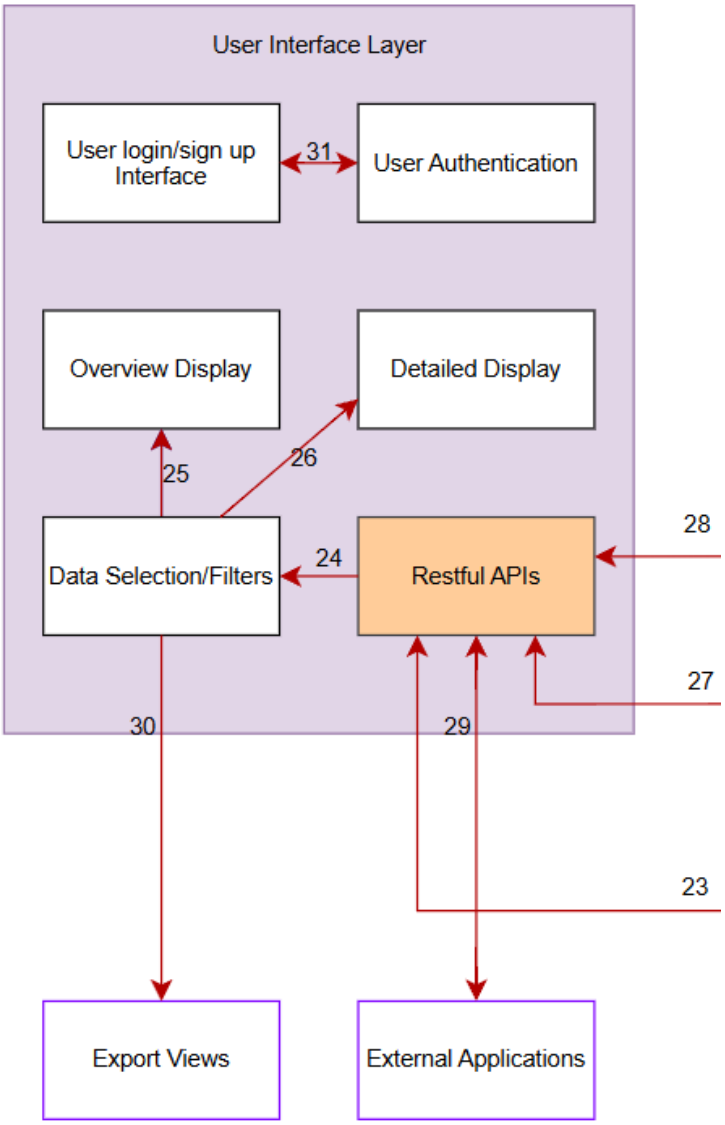

Figure 11: RESTful API Integration subsystem architecture diagram

6.6.1 Subsystem Operating System

The RESTful API Integration subsystem operates within a cross-platform browser environment and interacts with cloud-based endpoints, with no additional operating system dependencies required on the client side.

6.6.2 Subsystem Software Dependencies

- **Axios**: HTTP client used for making REST API calls with support for headers, interceptors, and error handling
- **React Query 4.x**: Used for API data fetching, caching, and synchronization of UI state with remote data
- **AbortController API**: Provides cancellation support for pending API requests

6.6.3 Subsystem Programming Languages

- **JavaScript (ES6+)**: Used to implement frontend-side API interactions and handle asynchronous operations
- **TypeScript**: Utilized for type-checking API request/response objects and enforcing interface contracts
- **JSON**: Used as the primary data exchange format between the frontend and Google Cloud Functions

6.6.4 Subsystem Data Structures

- **API Request Configuration**: Includes endpoint URLs, request payloads, headers, and authentication tokens
- **Response Cache Structure**: Maintains in-memory or indexed cache of API responses for performance optimization
- **Error Handling Objects**: Represent structured error data with retry flags and display information
- **Request State Management**: Tracks loading, success, and failure states of each API request

6.6.5 Subsystem Data Processing

The subsystem supports the following processing workflows:

- **HTTP Request Lifecycle Management**: Handles outbound requests to Google Cloud Functions and processes responses with retry and timeout logic
- **Data Serialization/Deserialization**: Converts JavaScript objects to JSON for transmission and parses responses upon receipt
- **Cache Invalidation Strategy**: Implements smart caching and expiration policies based on response types and data freshness
- **Error Recovery Protocols**: Incorporates error categorization, fallback strategies, and user-facing recovery messages

6.7 User Login Interface

The User Login Interface subsystem provides a responsive and interactive entry point for user authentication. It supports login using Google and Microsoft accounts via Firebase Authentication, leveraging OAuth 2.0. Instead of a traditional username/password form, this interface uses pop-up authentication and enforces access control through a predefined list of authorized email addresses. Upon successful login, users are redirected to the dashboard.

6.7.1 Subsystem Operating System

The login interface operates in a cross-platform browser environment and does not impose any additional operating system requirements.

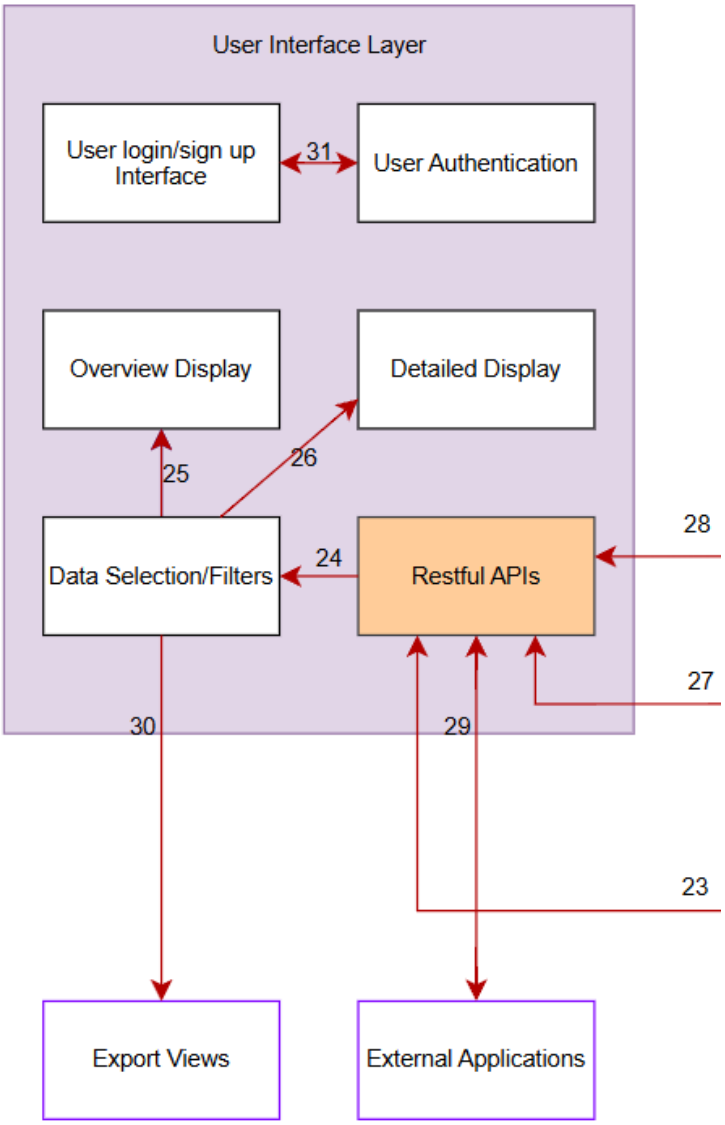

Figure 12: User Login Interface subsystem architecture diagram

6.7.2 Subsystem Software Dependencies

- **Firebase Authentication SDK**: Enables OAuth-based login using Google and Microsoft providers
- **React Router 6.x**: Handles client-side navigation after successful login
- **React (18.x)**: Used for building UI components and managing authentication logic
- **Firebase SDK**: Provides secure integration with Firebase Authentication

6.7.3 Subsystem Programming Languages

- **JavaScript (ES6+)**: Core logic for authentication workflows and error handling
- **JSX**: Templating syntax for rendering login UI components in React
- **CSS3**: Used for layout styling, button design, and responsive alignment

6.7.4 Subsystem Data Structures

- **User Object**: Retrieved from Firebase, includes user ID, email, and display name
- **Authorized Email List**: Static array of allowed emails used to enforce access control
- **Authentication State**: Firebase-managed session state reflecting current user status

6.7.5 Subsystem Data Processing

The login interface performs the following key actions:

- **OAuth 2.0 Sign-In**: Initiates pop-up authentication with the selected provider (Google or Microsoft)
- **Email Whitelisting**: Validates the logged-in user's email against a static list of allowed accounts
- **Navigation Logic**: Redirects authorized users to the dashboard and blocks unauthorized users with alerts
- **Logout Flow**: Ensures unauthorized users are signed out immediately after failed access checks