Nimbus – A Weather App, but Cooler

Khoa Dang Nguyen   
Department of Computer Science  
University of Texas at San Antonio  
Austin, United States  
khoa.nguyen3@my.utsa.edu  
  
Nam Nguyen  
Department of Computer Science  
University of Texas at San Antonio  
San Antonio, United States  
nam.nguyen4@my.utsa.edu  
  
Jovan Dazzlyn Salazar-Flores  
Department of Computer Science   
University of Texas at San Antonio  
San Antonio, United States  
jovan.salazar-flores@my.utsa.edu

*Abstract*— Our final project in this class is called Nimbus, a Flask-Jinja-based web application deployed on Google Cloud through App Engine Standard. Nimbus provides a feature that allows users to search for current and weekly weather forecasts by providing a city location. It also displays a temperature chart alongside the local time. One of our key features is input validation, which enables users to simply type and submit a city name without any format restrictions. The feature employs a large-language model API to handle input processing which searches for a city name corresponding to the provided input, then passes that information along to our weather API to display the weather in that location.

Keywords— Flask, Google Cloud, App Engine, API, Python, weather, forecast, Groq, web development

# Introduction

Accurate and accessible weather information is crucial for planning outdoor activities, such as hikes, fishing trips, or visits to the local tennis court. However, raw data from weather APIs requires a degree of technical knowledge to retrieve and interpret. To address this challenge, our team created a single-page web application that allows user to enter the name of a city in natural language and receive that city’s local time, current conditions, eight-day forecast, as well as a visualization of the city’s temperature trends according to that forecast.

In our solution, free-form input is normalized using a large-language-model API. The model’s output is then used in order to perform geocoding using the OpenWeatherMap API, providing us a more concrete location. Next, weather data is fetched from the Open-Meteo API. Finally, temperature trends for the location over the next eight days are rendered as line charts using the Matplotlib library. The entire service is hosted on Google Cloud using App Engine Standard.

# Project Goals

## Functional Goals

* Parse user input. Collect unstructured location text from the user and parse it into structured JSON data.
* Fetch weather data. Retrieve and process both the current conditions and 8-day forecast for the user’s location.
* Handle time zones. Look up the location’s time zone and display its local time alongside the weather data.
* Visualize forecasts. Generate and embed a chart showing weather trends based on the location’s eight-day forecast.
* Single-page UI. Provide all of the desired functionality within a simple single-page web application.

## Learning & Development Goals

* Cloud deployment. Deploy on the Google Cloud Platform using App Engine Standard and build familiarity with both.
* Secrets management. Securely manage API keys locally and in production.
* API usage. Gain a deep understanding of how API functions and learn to implement them effectively in projects to get real-time data.
* JSON Data processing. Utilize JSON type data to transfer data between different APIs and Python modules to process the information.

# Key Features

## Smart LLM-integrated input validation

Users can freely enter the search bar without having to follow a certain format (e.g., LA), and our validation will understand and match it with the city mentioned in the text (in this case, Los Angeles, California, USA). The input validation can also recognize the historical name of an entered location. For instance, if the user enters Saigon (the former name of Ho Chi Minh City), this feature will also automatically match the current city name, Ho Chi Minh City.

## Accurate current and forecast weather data

This core feature will provide accurate weather conditions data for the current time at a specific location, including temperature, humidity, wind speed, and sky conditions. Additionally, it provides precise weather conditions for the upcoming 8-day forecast, along with the highest and lowest temperatures per day.

## Dynamic temperature forecast visualization

Based on the forecast information, the user will also have access to the graphs illustrating a visual view of the change in temperature for the upcoming 8 days in both Celsius and Fahrenheit.

## Dynamic theme and user-friendly web design

Nimbus offers intuitive and user-centric web design that employs dynamic themes based on different weather conditions and times of day. The web application automatically applies themes such as day/night, rainy, clear sky, storm, etc. In addition, Nimbus is also fully responsive as we offer both mobile view and desktop view for all devices.

# Tools & Technology

## Programming Languages

* Frontend: HTML, CSS, and JavaScript were used for page structure, visual styling, and client-side functionality.
* Backend: Python 3.12 was used for server-side logic, data classes, and API calls.

## Frameworks

Flask serves as the core web application framework in this project for routing, request handling, and rendering the frontend by populating the page’s underlying data dependencies. The Jinja2 template engine then injects index.html with dynamic data from the backend and displays it directly within the HTML template.

## APIs and Libraries

* Groq (LLama3-70B) API for natural language processing to parse unstructured text data into JSON containing city, state, and country.
* OpenWeatherMap API for geocoding city, state, and country into latitude and longitude coordinates.
* Open-Meteo API for obtaining current weather data and 8-day forecast data from latitude and longitude.
* Matplotlib to generate temperature trends graphs from 8-day forecast data.
* python-dotenv for loading API keys from local .env during development.
* Google Cloud Secret Manager for fetching API keys securely in production.

## Deployment

The Flask application and all relevant files are hosted on the Google Cloud Platform via the App Engine Standard environment, which provides detailed logging and a largely preconfigured runtime environment. Gunicorn is used in production for handling HTTP requests.

## Version Control

All the code lives in a public GitHub repository, which also hosts additional development branches, issue tracking, and a ReadMe for initial setup instructions.

# Methodology & Architecture

## Overview of Modules

### weather.py

Responsible for communicating with external geocoding and weather API providers, as well as defining the data classes used to store fetched JSON responses as Python objects. This module finds a location’s coordinates, fetches the current conditions and eight-day forecast, and makes the parsed data available to the rest of the app.

### main.py

Acts as the central controller and entry point for our Flask application. It receives the user’s location input data as text and processes it using our other modules. Once all the necessary data is finished processing, it sends the rendered web page with all the relevant information to the frontend and allows that data to be displayed to the user through the browser.

### AI\_helper.py

Handles our API calls to Groq, a large-language-model API service, sends the user’s input for the model to interpret. LLM will receive raw text data directly from the user through main.py and will automatically match locations based on the given data. It will then return a JSON object with the key and data, including city name, state, and country. If the user input is not valid or LLM cannot find a current city that matches the description (e.g., Atlantis), LLM will return a JSON object containing only errors.

### secrets\_helper.py

Keeps our API keys out of our code. In development, this module helps us load keys from a local file. In production, it pulls them securely from google Cloud’s Secret Manager. The rest of the modules are then able to request specific API keys without worrying about which environment the code is being executed from.

### graph.py

Graph.py is responsible for handling the logic to display a graph that represents the forecast’s high, low, and average temperatures. By using the Python library, Matplotlib, we’re able to visually plot points on a graph to the temperature data and illustrate trends.

### weather\_info\_codes.py

This contains a python dictionary that maps the extracted weather codes from the APIs to text descriptions.

### index.html

Index.html is the only front-end template rendered by main.py. It uses Jinja2 templating to display current weather data and forecast data, which includes the current weather, 8-day forecast, temperature, graph data, and local time zone.

## Data Workflow

The data workflow for our project, as illustrated in Fig. 1, begins with input from the user. The user is prompted to enter a city name, state, and country through our website. This input is then transferred to the backend, which is hosted on google app engine.

Once the request is received, the backend makes API calls to third party services using the provided location details. The API responds with current weather data and 8-day forecast that includes temperature, weather code, weather icon, weather description, humidity, wind speed, dates, maximum temperatures, and minimum temperatures.

Upon receiving the weather information, it is parsed and processed with relevant information. This data is then returned to the front end, where it will display the information to the user through a clean and responsive website.

Our project is deployed on Google app engine, which allows us to scale user demand, handle requests efficiently, and deploy updates in the future if needed. This cloud-based data workflow ensures that users can get fast and accurate weather information at any time, while taking advantage of Google app engine’s ease of use, scalability, and reliability.

## Software Development Methodology

Our project, Nimbus, was mainly developed based on the Iterative Development Methodology. We initially developed a fully functional prototype with core functionalities, and from there started expanding and adding additional features.

Feature enhancements per iteration:

1. The first functionality to be developed was retrieval of API data given a City, State, and Country. How this data was obtained or validated were not concerns at this time. This was accomplished by simply developing the weather.py module as a standalone python script run on the command line.
2. Our original prototype, now incorporating Flask, simply focused on core features such as data fetching by making API calls to the Open-Meteo API and

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2: Early prototype.

retrieving the forecast for a static location (San Antonio City, Texas).

1. We then modified this prototype to load City, State, and Country from the URL using GET requests, which then allowed the user to load weather data for different locations (Fig. 2).
2. User Input & Validation: Added a user location search bar with different fields (city name, state, city) with basic input validation (this only limits the search to the United States).
3. Graph Rendering: Implemented graph generation using matplotlib to visualize temperature forecasts for the upcoming weeks of the location.
4. Front-End improvement: further restructure the front-end, rewrite HTML, add styling from CSS to improve UI.

A diagram of a data flow

AI-generated content may be incorrect.

Figure : Data flow diagram.

1. Groq input validation: remove the old input validation and integrate Groq LLM input validation, which allows the user to expand the search proximity to anywhere in the world.

## Testing Process

Our testing process consisted of three main areas: input validation, error handling, and performance benchmarking.

For testing input validation through our large-language-model API, we entered well-known cities such as “New York, US”, “Tokyo”, “Paris, France”, etc., and verified that the expected JSON response was being returned. We tested these inputs in different ways, using abbreviations at times such as “LA” for “Los Angeles”, or providing inputs without any separating commas. We also provided historical names for cities, such as “Saigon” for “Ho Chi Minh City”, or “Constantinople” for “Istanbul”.

For error handling, we entered non-existent cities such as “Atlantis” or “Gotham” to confirm that our “Invalid location” error was being correctly set. We also entered city names with misspellings, such as “New Yurk” and “Hong Kung”, and verified that the model was able to match them regardless of misspellings. More major misspellings, such as “Pearus” for “Paris”, triggered our “Invalid location” error message. Additionally, we simulated problems with the Groq API by raising an exception just before the call would normally be made, which triggered the “Problem with validation service” error as expected.

Finally, we measured the response times for various portions of our backend, as well as the total elapsed time before returning the rendered page to the user. For comparison, the page took 2822.06 milliseconds before being rendered upon a fresh request for “Los Angeles” from the user. However, refreshing that page (and, therefore, not changing the current location from Los Angeles) resulted in a much lower 650.58 milliseconds before rendering. This load time would be even lower, but technical debt from earlier design decisions means that our temperature graphs are always rendered, regardless of caching.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3: Nimbus (full-page desktop view for daytime).

# Code Implementation

The core features of our project that were implemented highlight the logic behind our weather website’s functionality, API integration, and deployment as shown in the images below.

In this snippet, we parse the user input directly into the Groq validation function to get appropriate JSON data.

try:

jsonString = groqValidateInput(request.args.get("searchInput")) # Have Groq validate it, returning results as JSON

print("Groq input validation was successful.\n")

except Exception as exc: # In case something goes wrong during Groq input validation

jsonString = '{"Error":"service\_unavailable"}'

print("Groq input validation failed: %s\n", exc)

We then get latitude and longitude using the fields from Groq’s processed input (City, State, Country) and pass these coordinates to our get\_current\_weather function to get the current data for the location. After that, we also call the get\_forecast function, which returns the data for the 8-day forecast of the given location.

lat, lon = get\_lat\_long(city, state, country, OWM\_API\_KEY) # Get lat/lon through OWM API  
t0 = time.perf\_counter()

cached\_data['current']=get\_current\_weather(lat, lon)

t1 = time.perf\_counter()

print(f" Current weather fetch took {(t1 - t0) \* 1000:.2f} ms")

t0 = time.perf\_counter()

cached\_data['forecast'] = get\_forecast(lat, lon)

t1 = time.perf\_counter()

print(f" Forecast fetch took {(t1 - t0) \* 1000:.2f} ms\n")

In the following snippet, once we have the required data, we pass that data directly to the page template for rendering by Jinja2, which will display the relevant data in the appropriate fields.

return render\_template(

A screenshot of a weather forecast

AI-generated content may be incorrect.Screens screenshot of a weather forecast

AI-generated content may be incorrect.

Figure 4: Nimbus (scrollable mobile view for nighttime).

"index.html",

city = city,

state = state,

country = country,

error = error,

current = current,

forecast = forecast,

local\_time = local\_time)

# Project outcomes

Our team was able to meet our original goals for the project. We were able to take our original barebones prototype, which simply retrieved data via API calls and rendered it in plain text, and iterated on it until all those goals were met.

In its current form, Nimbus parses input from the user and validates it as a real-world location. It takes this location and retrieves the corresponding current and eight-day weather data.

The time zone for the location is obtained, allowing that location’s local time to be displayed to the user in real time.

The eight-day forecast data is processed to chart the location’s temperature trends, also allowing for the user to switch between Fahrenheit and Celsius.

Finally, all this functionality is delivered via a responsive, single-page web application (Fig 3.). Furthermore, our application is also compatible with mobile devices and dynamically changes its theme based on the time of day at the location of interest (Fig 4.).

# individual contributions

## Khoa Dang Nguyen

Designed the main interface for Nimbus, including logo, wireframe, background weather theme, and front-end structure. Designed and styled front-end template with user-friendly CSS, enabling a responsive view on both mobile and desktop. Planned and sketched primary features, set out primary objectives, plans, and deadlines for team to complete. Created Groq LLM integrated input validation function (AI\_helper.py) with tailored prompts to handle raw user input and return an accurate and machine-readable location. Designed an auto-generated temperature graph (graph.py) based on forecast data collected from the API using Matplotlib. Deployed the given graphs to the HTML templates. Included JavaScript in the front end for client-side interaction, auto-toggling day/night theme, and fetching local time (update in real time) using the time zone data from main.py. Reformatted and structured third-party API responses (e.g., OpenWeatherMap, Open-Meteo) for front-end display compatibility.

## Nam Nguyen

Deployed locally based python code to Google Cloud using Google App Engine Standard, defining app.yaml for build instructions. Created initial main.py and weather.html template to convert the command-line-based program into a web app prototype, using Flask. Implemented secrets\_helper.py and Google Secret Manager to securely fetch API keys from Secret Manager when executed through App Engine, and from .env when executed locally. Refactored main.py to reduce code redundancy and increase resiliency. Implemented data caching to reduce the frequency of API calls. Implemented parsing of time zone codes into time zone objects via pytz to display local times (later replaced by a different local time implementation). Debugged code from other members.

## Jovan Dazzlyn Salazar-Flores

Initial implementation of the project and backend. Integrated and extracted data from OpenWeatherMap geocoding API and Open-Meteo weather forecast API. Initial set-up of GitHub repository. Created and worked on weather.py with contributions from others. Created weather\_code\_info.py which helps extract descriptions based on weather codes. Initial implementation of requirements.txt. Set up static/icons directory which stores images used for display based on the weather codes. Created cloudbuild.yaml to help automate deployment to google app engine by triggering a build whenever changes/pushes are made to the GitHub repository.

# Lessons Learned & Potential Improvements

## Lessons Learned

* The initial solution of using a .env file to manage secrets without embedding them directly into the GitHub repository worked just fine during local prototyping but was incompatible with App Engine Standard. Rather than hardcoding secrets directly into app.yaml, Secret Manager was instead integrated into the project.
* The volume and frequency of API calls can quickly grow out of hand, especially if the site were to experience a much higher amount of traffic than encountered during the development process. While the entire set of geocoding and weather data are too large to be cached entirely, caching the user’s most recent request at least prevents duplicate API calls from being made on each page refresh.
* The use of a large-language-model as a solution for natural language processing and location input validation required increasingly strict prompts, as edge cases would occasionally trigger unexpected responses, causing JSON parsing to fail. Large-language-models, being non-deterministic, can be tricky to integrate if unexpected responses aren’t accounted for.
* When using Google App Engine, static image files such as matplotlib-generated graphs cannot be overwritten in the project directory in the cloud, although this may work fine during local development. One solution that we were able to solve was to save the generated graph images to /temp, a folder that stores temporary and writable files by Google App Engine.

## Potential Improvements

* In the application’s current state, caching is performed through a global object variable, rather than on a per-user basis, meaning that each user’s location searches would directly affect other users. As a proof-of-concept and demo, the team found this acceptable, but ideally we would implement this caching per-user through session variables. In Flask, session variables are stored in the user’s cookies, which would necessitate the need to account for users who have cookies disabled.
* More API validation and fallback logic would improve the reliability of the service. With more time, all API calls would have more robust try-except logic, and would also attempt multiple calls in the case of timeouts. Groq input validation features a small degree of exception handling, but this would ideally also be extended to the OpenWeatherMap, Open-Meteo, and Google Secret Manager APIs. The current codebase also implicitly trusts that these APIs will always return valid data in a particular format, which means that further validation of API response data would be necessary for a properly robust and secure implementation.
* With more time, the site’s styling could be further tightened up to better handle scaling and varied screen size dimensions. Depending on the user’s screen resolution, the forecast data may be too large at times, requiring vertical and/or horizontal scrollbars within the left panel. Within a certain range of horizontal page widths, header elements may overlap with each other. This could be addressed through more consistent use of flex boxes, and by reconfiguring the page’s CSS styling to use relative sizes, rather than fixed ones.
* For a weather app like Nimbus, we believe that an alert feature based on location and weather conditions would be very helpful to the user. As an example, in the scenario where the local temperature rises above 105 °F (or 40.5 °C), a pop-up alert would be displayed in the center of the app and show warnings of the high heat index. In addition, the application would also include any government warnings of natural disasters or weather conditions that could be potentially hazardous (tornadoes, hurricanes, floods, tsunamis, and earthquakes) and are relevant to the user’s searched area. This would allow Nimbus users to make more informed decisions, or to better prepare themselves for adverse weather conditions or events. However, due to resource and time constraints, we were unable to implement this feature (especially at a global scale) since some government warning APIs like the NWS API are limited to the United States.

# Project Repository and Live Deployment

* The full source code with complete deployment instructions is available on GitHub: [GitHub Repository](https://github.com/JDSxc/Weather-Proj)
* Google App Engine deployment: [Nimbus](https://elegant-canto-467120-n6.uc.r.appspot.com/)
* Project Demo: [Panopto Recording Demo presentation link](http://utsa.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=69f97289-f826-42f4-894b-b33100398484)