WeatherWise

Aditya Kumar

16/12/2025

1 Introduction

1.1 Project Overview

WeatherWise is a comprehensive real-time weather dashboard application that leverages modern web technologies to deliver accurate meteorological information to users. The application is built upon the Flask web framework, a lightweight yet powerful Python-based solution that provides the necessary infrastructure for handling HTTP requests, routing, and server-side logic. The weather data is sourced from OpenWeatherMap API, a widely-used and reliable meteorological data provider that offers current conditions and forecast information for locations worldwide.

The user interface employs contemporary design paradigms, specifically glassmorphism—a visual design trend characterized by semi-transparent elements with blur effects that create a frosted glass appearance. This aesthetic is complemented by the Dracula color theme, a carefully curated dark color palette that reduces eye strain while maintaining visual appeal and readability.

1.2 Project Objectives

The primary objectives of this project include:

- Providing real-time weather information with minimal latency
- Implementing a robust caching mechanism to optimize API usage and reduce costs
- Creating an intuitive user interface that supports multiple search modalities
- Ensuring system reliability through comprehensive error handling and retry mechanisms
- Maintaining security best practices for credential management
- Delivering responsive design that functions across various device form factors

2 System Architecture

2.1 Architectural Design Philosophy

The application follows a client-server architecture pattern, separating concerns between the presentation layer (frontend) and the business logic layer (backend). This separation enables independent development, testing, and scaling of each component. The backend serves as an intermediary between the frontend and the external OpenWeatherMap API, providing data transformation, caching, and error handling services.

2.2 Backend Implementation

2.2.1 Framework Selection

Flask was selected as the backend framework due to its lightweight nature, extensive ecosystem, and flexibility. Unlike more opinionated frameworks, Flask allows developers to structure applications according to specific requirements without imposing unnecessary constraints. The framework's WSGI compliance ensures compatibility with various deployment environments and production servers.

2.2.2 Core Components

The backend architecture is organized into two principal Python modules, each serving distinct responsibilities:

- `app.py`: This module constitutes the main application server, implementing routing logic, caching mechanisms, API integration with OpenWeatherMap, error handling, and logging functionality. It serves as the central orchestrator for all backend operations.
- `config.py`: This configuration module centralizes all system parameters, including API credentials, timeout values, retry attempts, default location settings, and cache duration. By consolidating configuration in a single location, the application adheres to the principle of single responsibility and facilitates easier maintenance and deployment across different environments.

2.2.3 Caching Mechanism

The application implements an in-memory caching system to optimize performance and reduce unnecessary API calls to OpenWeatherMap. Each API request to external services incurs latency and consumes quota allocations; therefore, caching frequently requested data significantly improves response times and reduces operational costs.

The caching implementation stores weather data for 600 seconds (10 minutes), a duration that balances data freshness with API efficiency. Weather conditions typically do not change dramatically within this timeframe, making it an appropriate cache lifetime. The cache key incorporates location identifiers (city name or coordinates) and unit preferences (metric or imperial) to ensure accurate data retrieval.

The cache validation function implements timestamp-based expiration:

```python
def is_cache_valid(cache_key):
    if cache_key not in cache:
            return False

        cached_time = cache[cache_key].get('timestamp')
        if not cached_time:
            return False

        time_diff = datetime.now() - cached_time
        return time_diff.total_seconds() < Config.CACHE_DURATION
```

This function performs three validation checks: first, it verifies the cache key exists; second, it confirms a timestamp is present; third, it calculates the time difference between the current moment and the cached timestamp, comparing it against the configured cache duration. Only when all conditions are satisfied does the function return True, indicating the cached data remains valid.

2.2.4 Data Retrieval Function

The `fetch_weather_data` function encapsulates the logic for communicating with the OpenWeatherMap API. This function accepts two parameters: an endpoint identifier (either 'weather' for current conditions or 'forecast' for predictions) and a dictionary of query parameters. The function constructs the complete API URL, appends the authentication key, and executes the HTTP request with appropriate timeout constraints.

2.3 Frontend Architecture

2.3.1 Technology Stack

The frontend is constructed using standard web technologies: HTML5 for semantic markup and structure, CSS3 for styling and visual presentation, and JavaScript for dynamic behavior and asynchronous communication with the backend. This technology stack ensures broad browser compatibility and leverages native web platform capabilities without requiring complex build processes or transpilation.

2.3.2 Data Visualization

Chart.js, a popular JavaScript charting library, provides interactive visualizations of temperature and humidity trends across the forecast period. The library generates responsive, canvas-based charts that adapt to various screen sizes while maintaining readability and interactivity. Users can hover over data points to view precise values, enhancing the interpretability of meteorological trends.

2.3.3 Responsive Design

The interface implements responsive design principles, utilizing CSS media queries and flexible layout techniques to ensure optimal presentation across desktop computers, tablets, and mobile devices. The glassmorphism effects are carefully calibrated to maintain visual appeal while ensuring text remains legible against varying background contexts.

3 Key Features

3.1 Weather Data Retrieval

3.1.1 Dual Search Modalities

The application supports two distinct methods for location specification, accommodating different user preferences and use cases:

- **City Name-Based Queries**: Users can enter a city name in a text input field. The backend forwards this query to OpenWeatherMap, which performs geocoding to resolve the city name to geographic coordinates and returns the corresponding weather data. This approach is intuitive for users who think in terms of place names rather than coordinates.
- **GPS Coordinate-Based Queries**: Users can provide precise latitude and longitude coordinates, either manually or through browser geolocation APIs. This method offers greater precision and is particularly useful for locations without well-defined city names, such as rural areas or specific landmarks. The coordinate-based approach also eliminates ambiguity when multiple cities share the same name.

3.1.2 Current Weather and Forecasts

The application provides two temporal perspectives on weather conditions:

- **Current Weather**: Real-time meteorological data including temperature, humidity, atmospheric pressure, wind speed and direction, cloud coverage, and general weather conditions (clear, cloudy, rainy, etc.). This information is updated frequently and reflects the most recent observations.
- **Five-Day Forecast**: Predictive weather data extending five days into the future, with data points at three-hour intervals. This granular forecast enables users to plan activities with awareness of anticipated weather patterns. The forecast includes the same meteorological parameters as current weather, allowing for comprehensive planning.

3.2 Error Handling and Resilience

3.2.1 Retry Logic

Network communication is inherently unreliable; transient failures can occur due to temporary connectivity issues, server overload, or intermediate network problems. To mitigate these issues, the application implements exponential retry logic with three attempts for failed API requests:

```python
    for attempt in range(Config.RETRY_ATTEMPTS):
        try:
            response = requests.get(
                url,
                params=params,
                timeout=Config.REQUEST_TIMEOUT
            )
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.warning(f"Attempt {attempt + 1} failed: {str(e)}")
            if attempt == Config.RETRY_ATTEMPTS - 1:
                logger.error(f"Failed to fetch weather data after {Config.↩
                    RETRY_ATTEMPTS} attempts")
                return None
```

This implementation attempts the request up to three times. Each attempt is wrapped in a try-except block to catch request exceptions. The `raise_for_status()` method ensures that HTTP error codes (4xx client errors, 5xx server errors) are treated as exceptions. If an attempt fails, the error is logged with a warning level, and the loop continues to the next attempt. Only after all three attempts have been exhausted does the function log an error and return None, signaling complete failure to the calling code.

3.2.2 Timeout Configuration

Each API request is configured with a 10-second timeout, preventing indefinite blocking when the remote server becomes unresponsive. Without timeout constraints, a single hanging request could render the entire application unresponsive. The timeout value represents a balance between allowing sufficient time for legitimate responses while preventing excessive wait times that degrade user experience.

3.2.3 Logging Infrastructure

The application employs Python's built-in logging module to record significant events, errors, and diagnostic information. Log messages are categorized by severity (INFO, WARNING, ERROR), enabling operators to filter and prioritize issues. Cache hits, API call attempts, and failures are all logged, providing visibility into application behavior and facilitating troubleshooting.

3.3 Configuration Management

3.3.1 Environment Variables

Sensitive credentials, particularly the OpenWeatherMap API key, are managed through environment variables rather than being hardcoded in source files. The `python-dotenv` library loads these variables from a `.env` file during application initialization. This approach offers several advantages:

- **Security**: API keys and secrets are not committed to version control, reducing the risk of credential exposure
- **Flexibility**: Different environments (development, staging, production) can use different credentials without code modification
- **Compliance**: Separation of code and configuration aligns with twelve-factor app methodology and security best practices

3.3.2 Centralized Configuration Class

All system parameters are consolidated in a single `Config` class, providing a centralized reference point for application settings:

```python
class Config:
    OPENWEATHER_API_KEY = os.getenv('OPENWEATHER_API_KEY', '')
    OPENWEATHER_BASE_URL = 'https://api.openweathermap.org/data/2.5'
    CACHE_DURATION = 600
    DEFAULT_CITY = 'Gaya'
    DEFAULT_COUNTRY = 'IN'
    DEFAULT_LAT = 24.7955
    DEFAULT_LON = 84.9994
    REQUEST_TIMEOUT = 10
    RETRY_ATTEMPTS = 3
    DEFAULT_UNITS = 'metric'
```

This configuration includes the OpenWeatherMap API base URL, cache duration, default location (Gaya, India), request timeout, retry attempts, and default temperature units (metric system). The default location serves as a fallback when users decline to share their geographic location or when geolocation is unavailable.

4 API Endpoints

4.1 Current Weather Endpoint

4.1.1 Endpoint Specification

`GET /api/weather/current`

This endpoint retrieves current meteorological conditions for a specified location. The endpoint accepts query parameters to specify the location and desired unit system.

4.1.2 Query Parameters

- `city`: City name as a string (e.g., "Delhi", "London"). This parameter is optional if coordinates are provided. The OpenWeatherMap API performs geocoding to resolve city names to coordinates.
- `lat`: Latitude coordinate as a decimal number (e.g., 24.7955). Must be provided in conjunction with the `lon` parameter. This parameter is optional if a city name is provided.
- `lon`: Longitude coordinate as a decimal number (e.g., 84.9994). Must be provided in conjunction with the `lat` parameter.
- `units`: Temperature unit system, either `metric` (Celsius, meters/second for wind) or `imperial` (Fahrenheit, miles/hour for wind). Defaults to `metric` if not specified.

4.1.3 Response Format

The endpoint returns JSON-formatted data containing temperature, humidity, atmospheric pressure, wind speed and direction, cloud coverage, weather description, and location information. In case of errors, the endpoint returns a JSON object with an error message and an appropriate HTTP status code.

4.1.4 Implementation Logic

The endpoint first validates that either a city name or coordinate pair is provided. It then constructs a cache key incorporating the location identifier and unit preference. The cache is checked for valid data; if found, the cached response is returned immediately. If the cache is empty or expired, the endpoint calls the `fetch_weather_data` function to retrieve fresh data from OpenWeatherMap, caches the result, and returns it to the client.

4.2 Forecast Endpoint

4.2.1 Endpoint Specification

```
GET /api/weather/forecast
```

This endpoint provides a five-day weather forecast with data points at three-hour intervals, enabling users to anticipate weather conditions for planning purposes.

4.2.2 Query Parameters

The forecast endpoint accepts identical query parameters as the current weather endpoint: `city`, `lat`, `lon`, and `units`. The parameter semantics and validation logic are consistent across both endpoints, ensuring a uniform API interface.

4.2.3 Response Format

The response contains an array of forecast objects, each representing a three-hour time slice. Each object includes timestamp, temperature, humidity, precipitation probability, wind conditions, and weather description. The forecast data enables the frontend to generate trend charts and multi-day weather displays.

4.3 Configuration Endpoint

4.3.1 Endpoint Specification

`GET /api/config`

This endpoint exposes application configuration to the frontend, enabling the client to initialize with appropriate default values without hardcoding them in JavaScript.

4.3.2 Response Format

The endpoint returns a JSON object containing default city, country code, coordinates, temperature units, and cache duration. This information allows the frontend to display weather for the default location immediately upon page load, improving perceived performance and user experience.

5 Security Considerations

5.1 Credential Management

5.1.1 Environment Variable Storage

API credentials are stored exclusively in environment variables, never in source code. The `.env` file containing these variables is explicitly listed in `.gitignore`, preventing accidental commits to version control systems. This practice is critical for preventing credential leakage, particularly when code is shared publicly on platforms like GitHub.

5.1.2 API Key Validation

Before attempting any external API calls, the application validates that the OpenWeatherMap API key is present. If the key is missing, the application logs a prominent warning message and refuses to make API requests, preventing cryptic error messages and failed requests.

5.2 Request Validation

5.2.1 Parameter Validation

All API endpoints implement comprehensive parameter validation. The current weather and forecast endpoints verify that either a city name or coordinate pair is provided, returning HTTP

400 (Bad Request) status codes with descriptive error messages when validation fails. This prevents malformed requests from propagating to the OpenWeatherMap API and provides clear feedback to client developers.

5.2.2 Timeout Mechanisms

Each outbound HTTP request includes a timeout parameter set to 10 seconds. This prevents indefinite blocking when the remote server becomes unresponsive or network connectivity is degraded. Without timeouts, a single slow request could exhaust server resources and degrade service for all users.

5.2.3 Error Response Handling

The application implements custom error handlers for HTTP 404 (Not Found) and 500 (Internal Server Error) status codes. These handlers return JSON-formatted error messages rather than HTML error pages, ensuring consistency with the API's JSON-based interface and enabling clients to programmatically handle errors.

6 Performance Optimization

6.1 Caching Strategy

6.1.1 Cache Key Design

Cache keys are constructed by concatenating location identifiers (city name or coordinates) with unit preferences. This ensures that requests for the same location with different unit systems retrieve distinct cached data, preventing unit conversion errors. For example, `current_Delhi_metric` and `current_Delhi_imperial` represent separate cache entries.

6.1.2 Cache Duration Rationale

The 600-second (10-minute) cache duration represents a carefully considered balance between data freshness and API efficiency. Weather conditions typically evolve gradually; a 10-minute delay in updates rarely impacts user decision-making while significantly reducing API calls. For a moderately trafficked application, this caching strategy can reduce API calls by 90% or more.

6.1.3 Cache Invalidation

The current implementation uses time-based cache invalidation, where entries expire after a fixed duration. This approach is simple and predictable, though it does not account for scenarios where weather conditions change rapidly (e.g., sudden storms). Future enhancements could implement more sophisticated invalidation strategies, such as event-based invalidation triggered by severe weather alerts.

6.2 CORS Configuration

6.2.1 Cross-Origin Resource Sharing

The application enables CORS (Cross-Origin Resource Sharing) through the Flask-CORS extension. CORS is necessary because modern browsers enforce the same-origin policy, which prevents JavaScript running on one domain from accessing resources on another domain. By enabling CORS, the backend explicitly permits the frontend (which may be served from a different port during development) to make API requests.

6.2.2 Security Implications

While CORS enables necessary cross-origin communication, it must be configured carefully to avoid security vulnerabilities. The current implementation permits all origins, which is acceptable for development but should be restricted to specific domains in production environments to prevent unauthorized access.

7 Screenshot

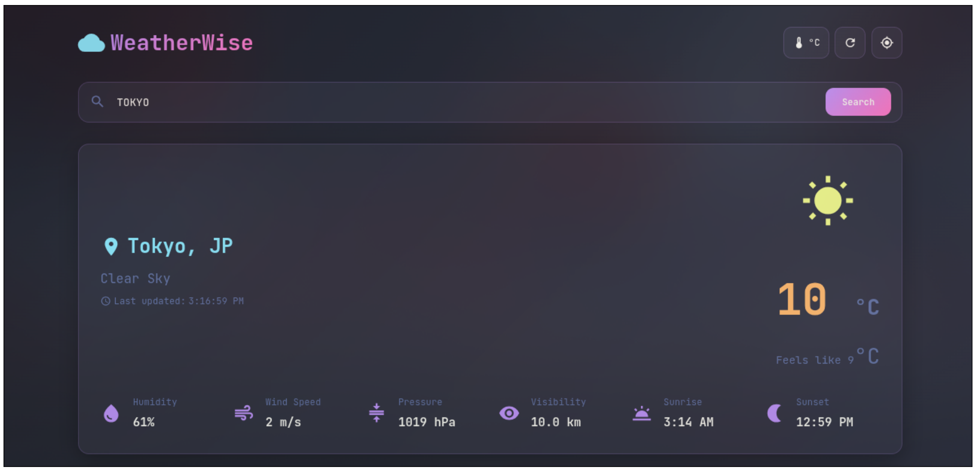


Figure 1: Header

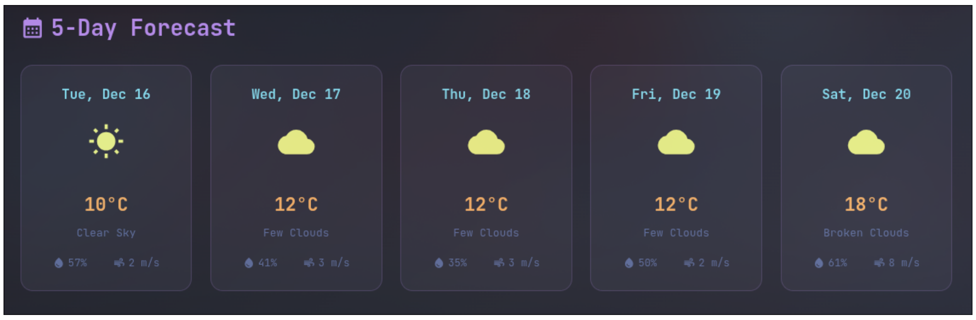
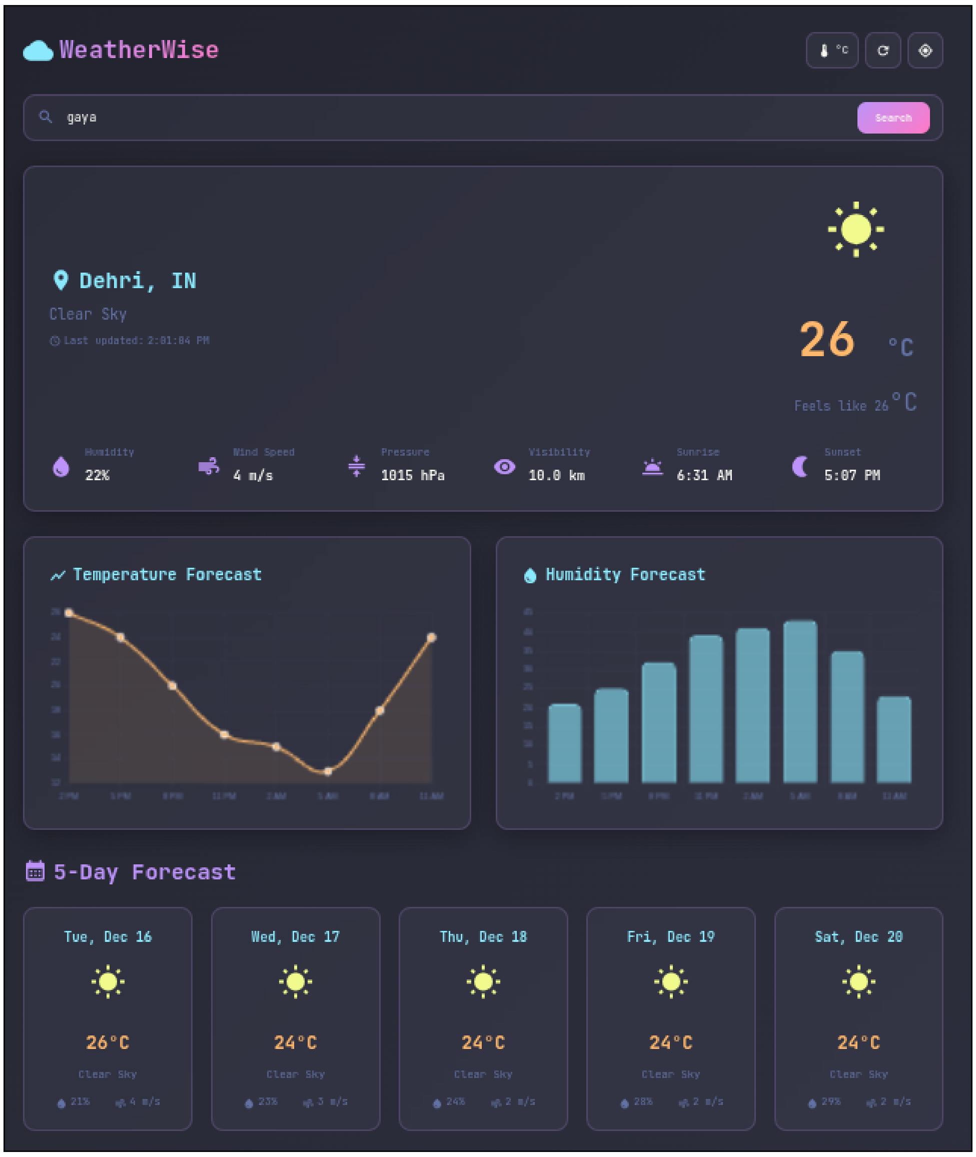


Figure 2: Five-Day Forecast


WeatherWise

°C

gaya

Search

Dehri, IN

Clear Sky

Last updated: 2:01:04 PM

26 °C

Feels like 26 °C

Humidity	Wind Speed	Pressure	Visibility	Sunrise	Sunset
22%	4 m/s	1015 hPa	10.0 km	6:31 AM	5:07 PM

Temperature Forecast

Humidity Forecast

5-Day Forecast

Tue, Dec 16	Wed, Dec 17	Thu, Dec 18	Fri, Dec 19	Sat, Dec 20
26°C	24°C	24°C	24°C	24°C
Clear Sky	Clear Sky	Clear Sky	Clear Sky	Clear Sky
21% 4 m/s	23% 3 m/s	24% 2 m/s	28% 2 m/s	29% 2 m/s

Figure 3: Overview

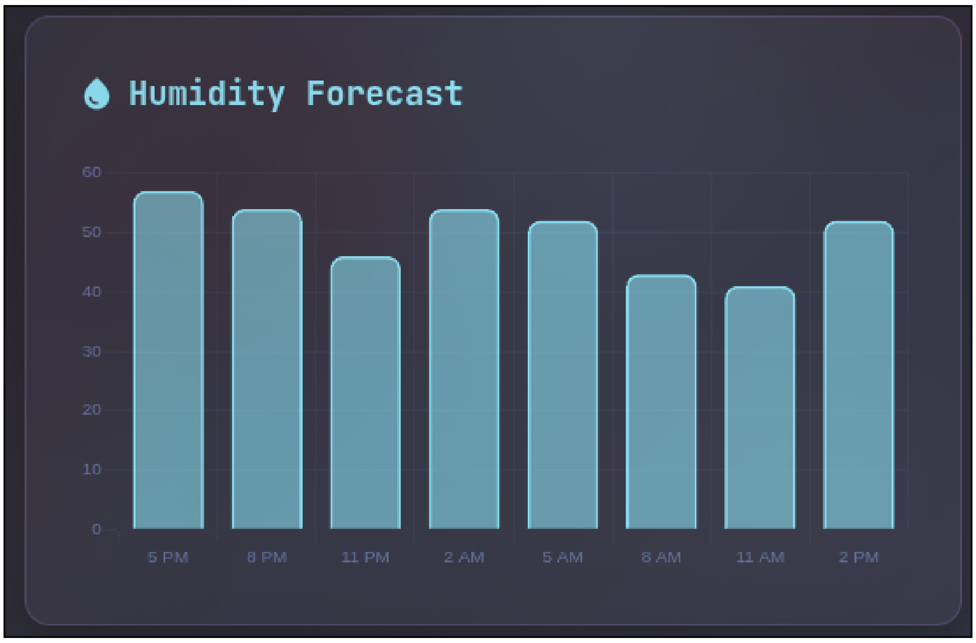


Figure 4: Overview

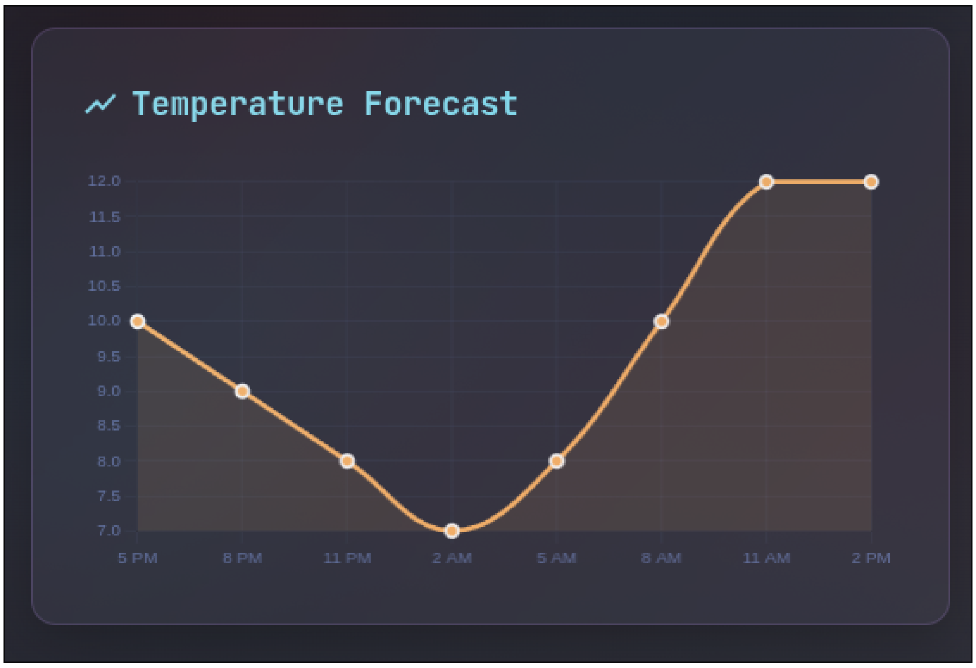


Figure 5: Temperature Forecast Graph

8 Conclusion

8.1 Project Achievements

WeatherWise successfully demonstrates the integration of multiple technologies to create a functional, user-friendly weather dashboard. The application combines robust backend engineering with modern frontend design principles, resulting in a system that is both reliable and aesthetically pleasing. Key achievements include:

- Implementation of an efficient caching mechanism that reduces API costs and improves response times
- Comprehensive error handling and retry logic that ensures system resilience
- Secure credential management following industry best practices
- Responsive user interface that functions across diverse device types
- Clean, maintainable code architecture that facilitates future enhancements

8.2 Technical Stack Summary

The application leverages the following technologies:

- **Backend**: Python 3.x provides the programming language foundation; Flask serves as the web framework; Flask-CORS enables cross-origin requests; Requests library handles HTTP communication
- **External API**: OpenWeatherMap API v2.5 supplies meteorological data
- **Frontend**: HTML5 provides semantic structure; CSS3 implements styling and animations; JavaScript enables dynamic behavior; Chart.js generates interactive visualizations
- **Design**: Glassmorphism aesthetic with Dracula color scheme creates a modern, visually appealing interface

8.3 Future Enhancement Opportunities

While the current implementation is fully functional, several enhancements could further improve the application:

- Integration of additional weather data sources for redundancy and comparison
- Implementation of user accounts to save preferred locations and settings
- Addition of severe weather alerts and notifications
- Historical weather data visualization for trend analysis
- Progressive Web App (PWA) capabilities for offline functionality
- Internationalization support for multiple languages

8.4 Authors

This project was developed by Aditya Kumar and Ashif Rahman, demonstrating proficiency in full-stack web development, API integration, and modern design principles.

8.5 License

WeatherWise is released under the GNU General Public License v3.0 (GPL-3.0), ensuring that the software remains free and open-source. This license permits users to use, modify, and distribute the software while requiring that derivative works also remain open-source under the same license terms.