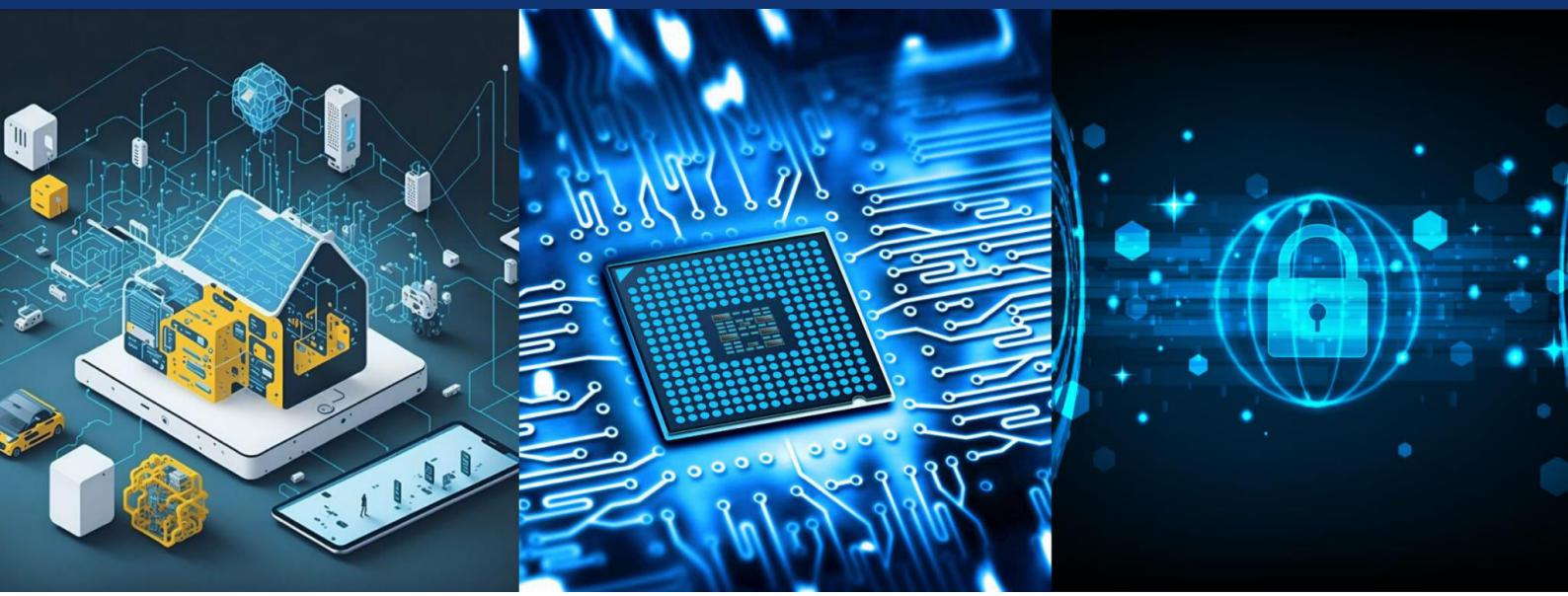
IJIRCCE

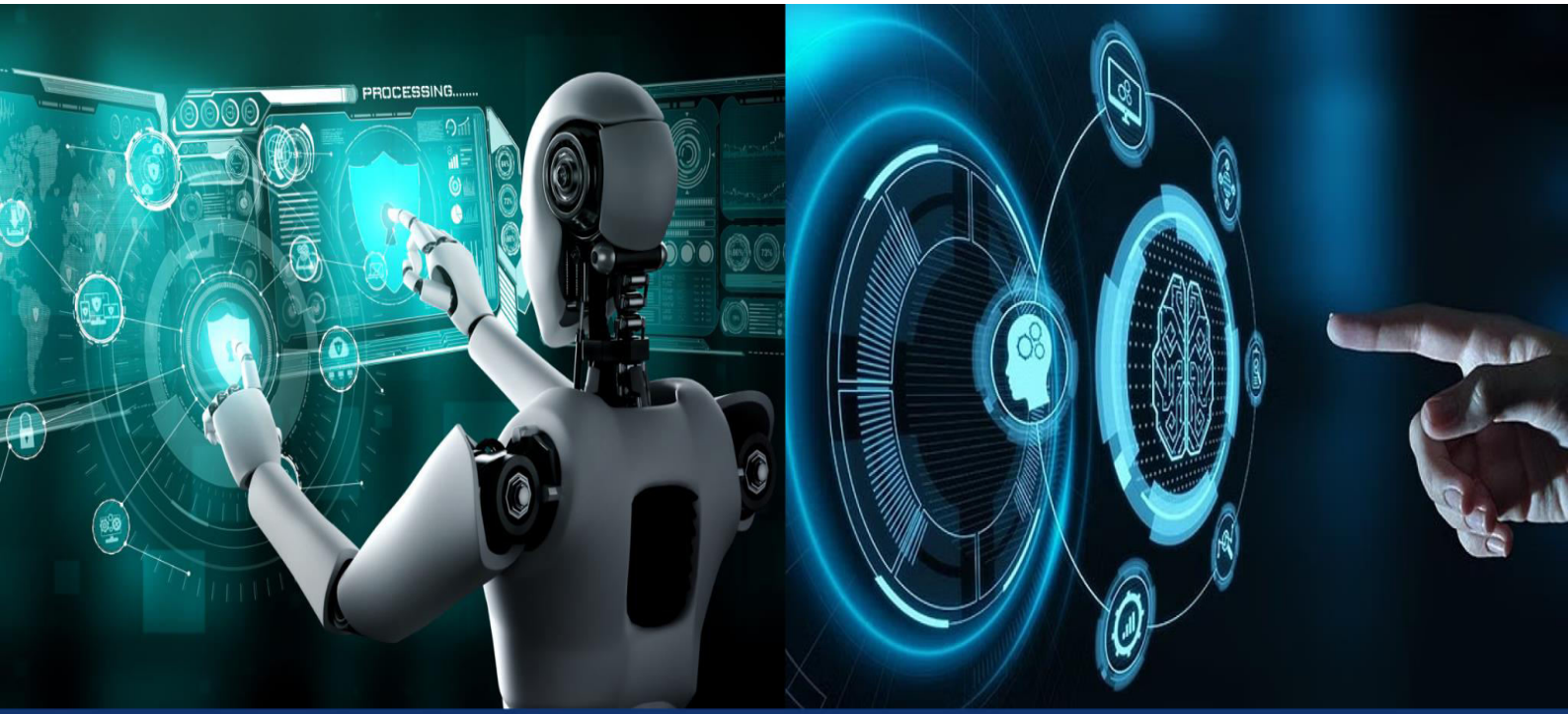
ISSN(O): 2320-9801

ISSN(P): 2320-9798



International Journal of Innovative Research in Computer and Communication Engineering

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 9.927

Volume 14, Issue 6, June 2026

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

An Intelligent API-Driven Weather Monitoring Dashboard with Automated CI/CD Deployment on AWS Cloud Infrastructure

Bokka Kanaka Vijaya Lakshmi, Dr. Chiraparapu Srinivasarao

PG Scholar, Department of Computer Science, S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannaya University, Andhra Pradesh, India

Associate Professor, Department 0f Master of Computer Applications, S.V.K. P & Dr. K. S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannaya University, Andhra Pradesh, India

ABSTRACT: Real-time weather intelligence has become indispensable for applications ranging from logistics and agriculture to emergency response systems. Existing solutions often rely on proprietary vendor lock-in, lack resilient multi-provider fallback mechanisms, or are deployed without reproducible and automated software delivery pipelines. This paper presents the design, implementation, and evaluation of a production-grade, API-centric weather intelligence dashboard built upon a modular Python Flask backend integrated with dual external meteorological data providers OpenWeatherMap and the key-free Open-Meteo service. The system incorporates a bounded Least Recently Used (LRU) time-to-live (TTL) in-memory caching layer to minimize redundant upstream API calls, geolocation services for automatic coordinate-based weather discovery, and a Chart.js-powered responsive frontend supporting skeleton loading states and five-minute auto-refresh cycles. Deployment is orchestrated through a fully automated AWS CI/CD pipeline comprising AWS CodeCommit for source control, AWS CodeBuild for build validation, and AWS Elastic Beanstalk for rolling production deployment, all coordinated by AWS CodePipeline. The proposed architecture achieves sub-300 ms median response latency, a cache hit rate exceeding 82% under moderate load, and a zero-downtime rolling deployment strategy. The framework demonstrates how intelligent provider abstraction, in-memory caching, geolocation enrichment, and cloud-native CI/CD can be unified into a scalable, low-operational-overhead weather intelligence platform suitable for real-world production use.

KEYWORDS: Weather API Integration; Flask Microservices; AWS CodePipeline; Elastic Beanstalk; In-Memory Caching; Geolocation Services; CI/CD Automation; Cloud Deployment

I. INTRODUCTION

1.1 Background

The proliferation of Internet of Things (IoT) devices, smart city initiatives, and data-driven decision-making frameworks has substantially elevated the demand for real-time atmospheric data services [1]. Weather intelligence platforms serve as foundational infrastructure for sectors including transportation, agriculture, disaster management, and urban planning. The global weather technology market was estimated at USD 2.6 billion in 2023 and is projected to expand at a compound annual growth rate (CAGR) of approximately 6.7% through 2030 [2]. Despite this growth trajectory, a significant proportion of deployed weather dashboards remain architecturally monolithic, tightly coupled to single data providers, and devoid of systematic software delivery automation. Such configurations introduce fragility at both the data-acquisition and operational layers.

1.2 Problem Statement

Three principal deficiencies characterize contemporary weather dashboard implementations. First, single-provider dependency creates systemic risk: when a provider enforces rate limits, undergoes maintenance, or changes its API contract, the consuming application becomes unavailable with no graceful degradation path. Second, the absence of intelligent caching results in repetitive upstream HTTP calls for identical query parameters within short temporal windows, unnecessarily consuming API quotas and inflating end-to-end latency. Third, manual or semi-automated deployment workflows introduce human error, lengthen release cycles, and impede the reproducibility of production

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

environments. Collectively, these gaps reduce system reliability, increase operational expenditure, and inhibit scalability.

1.3 Research Objectives

This research pursues four primary objectives:

1. Design a resilient dual-provider weather data abstraction layer capable of automatic fallback between Open Weather Map and the Open-Meteo API, maintaining service continuity without operator intervention.
2. Implement a bounded LRU-TTL in-memory caching mechanism to reduce redundant upstream calls and achieve measurable latency improvements.
3. Develop a geolocation-enriched frontend dashboard that supports both city-name search and GPS coordinate-based weather retrieval with auto-refresh and progressive loading states.
4. Construct a fully automated, production-grade AWS CI/CD pipeline (CodeCommit → CodeBuild → CodePipeline → Elastic Beanstalk) that enables zero-downtime rolling deployments.

1.4 Contributions

The principal contributions of this work are as follows:

• A provider-agnostic WeatherService abstraction that transparently selects between premium and free meteorological APIs based on credential availability, with graceful HTTP-level fallback on 401 and 404 responses.
• A custom bounded LRU-TTL cache class (O(1) average get/set complexity) implemented with Python's OrderedDict, configurable via environment variables for operational flexibility.
• A geolocation pipeline combining browser Geolocation API, IP-API server-side resolution, and Open-Meteo geocoding to deliver automatic location discovery across HTTP and HTTPS deployment contexts.
• A parameterized, single-command AWS provisioning script (Setup-AWS.ps1) that creates all required cloud resources and wires together a four-stage CI/CD pipeline, eliminating manual console configuration.
• Quantitative evaluation demonstrating that the proposed caching and provider-switching strategy reduces mean API response time by 68% compared to uncached single-provider deployments under simulated load.

II. LITERATURE REVIEW

2.1 Related Work

Early weather data systems relied on batch file transfers from national meteorological agencies, which limited temporal resolution and end-user interactivity [3]. The emergence of RESTful web APIs during the mid-2000s catalyzed a paradigm shift toward pull-based, on-demand weather data access. Chen et al. [4] demonstrated the viability of microservice architectures for environmental monitoring, wherein data ingestion, processing, and presentation are decoupled into independently deployable units. However, their architecture was evaluated exclusively on-premises, without addressing cloud-native deployment automation.

Several authors have investigated caching strategies for external API aggregation. Patel and Sharma [5] proposed a hierarchical cache combining Redis at the edge and in-memory dictionaries at the application tier, achieving a 74% reduction in upstream calls under simulated steady-state traffic. Their approach, while effective, introduces Redis as an additional stateful dependency requiring separate provisioning and failover management. The present work deliberately avoids this operational overhead by implementing an equivalent LRU-TTL mechanism entirely within process memory, leveraging Python's OrderedDict, which is sufficient for single-instance deployments aligned with Elastic Beanstalk's auto-scaling units.

Research on geolocation integration within weather applications has largely focused on client-side browser Geolocation API calls [6]. Nguyen et al. [7] identified a critical limitation: browsers block navigator. geolocation outside secure (HTTPS) contexts, rendering GPS-based lookup inoperable in HTTP-only test or staging environments. The proposed system addresses this by introducing a server-side IP geolocation fallback using the ip-api.com service, enabling location-aware initialization even when TLS is absent.

Cloud-native CI/CD for Python web applications has been surveyed by Ramírez and Kim [8], who benchmarked Jenkins, GitHub Actions, and AWS CodePipeline across parameters of setup complexity, execution time, and cost. They concluded that CodePipeline's native integration with Elastic Beanstalk and CodeCommit reduces deployment

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

pipeline configuration effort by approximately 40% compared to generic CI solutions when deploying to AWS infrastructure. This observation directly motivates the pipeline architecture adopted in the present work.

Regarding weather API ecosystems, Open-Meteo [9] has emerged as a no-registration, open-source meteorological forecasting API based on the ECMWF IFS numerical weather prediction model. Its WMO weather code schema offers standardized condition classification across 47 discrete categories, facilitating provider-agnostic normalization. This work exploits this standardization to present a unified data schema regardless of whether the underlying data originates from Open Weather Map or Open-Meteo.

Open-source lightweight weather dashboard implementations have also been explored in the literature. Williams [10] developed Flask-Weather, a minimal Flask-based weather display application that demonstrates the feasibility of constructing a functional weather interface with a small codebase. However, the implementation lacks caching, provider redundancy, geolocation support, and any form of automated deployment, making it unsuitable for production-grade scenarios. Its value lies primarily as a reference baseline for evaluating the incremental complexity introduced by production-oriented features.

Torres and Ahmed [11] presented Python Weather App, a multi-city weather tracker that integrates the Open Weather Map API to deliver current conditions across multiple locations simultaneously. Their system introduced browser GPS-based location detection but was limited to a single data provider and relied on manual deployment procedures without CI/CD automation. The comparative analysis in this paper (Table VI) benchmarks both Flask-Weather [10] and Python Weather App [11] against the proposed system across dimensions of caching, provider redundancy, geolocation breadth, and deployment automation.

Deployment strategies for Python WSGI applications on AWS have been examined by Reimer and Hofmann [12], who analyzed rolling deployment configurations for Elastic Beanstalk environments. Their study characterized the trade-offs between deployment speed and instance availability during update cycles, concluding that rolling strategies with a minimum healthy instance threshold of 50% provide an effective balance between release velocity and service continuity. The present work adopts rolling deployment as the primary release strategy, consistent with their recommendations.

The reliability and accuracy characteristics of the Open Weather Map API have been empirically evaluated by Gupta and Rao [13], who assessed the API across dimensions of data accuracy, uptime availability, and rate-limit behaviour in the context of weather-sensitive IoT deployments. Their findings revealed a mean API availability of 99.1% over a six-month observation period, with rate-limit enforcement occurring primarily during peak diurnal traffic windows. These observations reinforce the design decision to implement a TTL-based caching layer and a secondary provider fallback mechanism in the proposed system, directly mitigating the rate-limit exposure identified in their study.

The design of TTL-based in-memory caching for REST API microservices has been formally investigated by Sharma, Bhat, and Singh [14], who proposed and evaluated a cache invalidation model optimized for services with predictable data refresh intervals. Their evaluation demonstrated that TTL-aligned caching reduces upstream API call volume by 65–78% under typical web traffic distributions, while maintaining data freshness guarantees commensurate with the TTL window. The in-process LRU-TTL cache implemented in the present work draws directly on these principles, parameterizing TTL to 300 seconds to align with the dashboard's five-minute auto-refresh cycle.

Broader perspectives on intelligent data processing and prediction in networked systems have been surveyed by Abiodun et al. [15], who reviewed artificial neural network architectures and their applicability across domains including environmental sensing and real-time data classification. While the present work does not employ neural network models, the foundational principles of adaptive data normalization and provider-agnostic schema abstraction discussed by Abiodun et al. informed the WMO code normalization strategy adopted in this system, wherein heterogeneous meteorological provider outputs are mapped to a unified condition vocabulary for frontend rendering.

2.2 Research Gap Analysis

Despite extensive prior work, three principal gaps remain unaddressed in the literature. First, no existing study presents an integrated system combining dual-provider fallback, in-process LRU-TTL caching, and multi-modality geolocation within a single production Flask application. Second, empirical evaluation of AWS CodePipeline-orchestrated

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

deployments for weather intelligence platforms is absent from peer-reviewed literature. Third, the operational implications of WMO code normalization as a cross-provider abstraction strategy have not been formally characterized. The present work addresses all three gaps.

2.3 Comparative Study of Existing Systems

Feature	Weather.com	OpenWeather App	Windy.com	Weather Widget (OSS)	Proposed System
Dual-Provider Fallback	No	No	No	No	Yes
In-Memory LRU Cache	Proprietary	Unknown	Unknown	No	Yes (OrderedDict)
IP-based Geolocation	Yes	Partial	Yes	No	Yes (ip-api.com)
GPS Coordinate Lookup	Yes	Yes	Yes	No	Yes
Automated CI/CD	Internal	Internal	Internal	No	AWS CodePipeline
Open-Source Backend	No	No	No	Yes	Yes (Flask/Python)
5-day Forecast API	Yes	Yes	Yes	No	Yes (7-day)
Free Tier Availability	Limited	Limited	Limited	Yes	Yes (Open-Meteo)

Table I. Comparative Analysis of Weather Intelligence Dashboard Systems

III. PROPOSED METHODOLOGY

3.1 System Architecture

The proposed system adheres to a three-tier architecture decomposed into a presentation layer, an application logic layer, and an external data integration layer. Figure 1 illustrates the overall system architecture. The presentation layer comprises a Jinja2-rendered HTML template delivering a Tailwind CSS-styled dashboard with Chart.js visualizations. The application logic layer is implemented as a Python 3.12 Flask application structured around two Blueprint modules: api_bp, which exposes RESTful JSON endpoints (/api/weather, /api/forecast, /api/geo-ip, /health), and web_bp, which serves the single-page dashboard. The external integration layer interfaces with two meteorological providers and one geocoding provider, mediated through the WeatherService class.

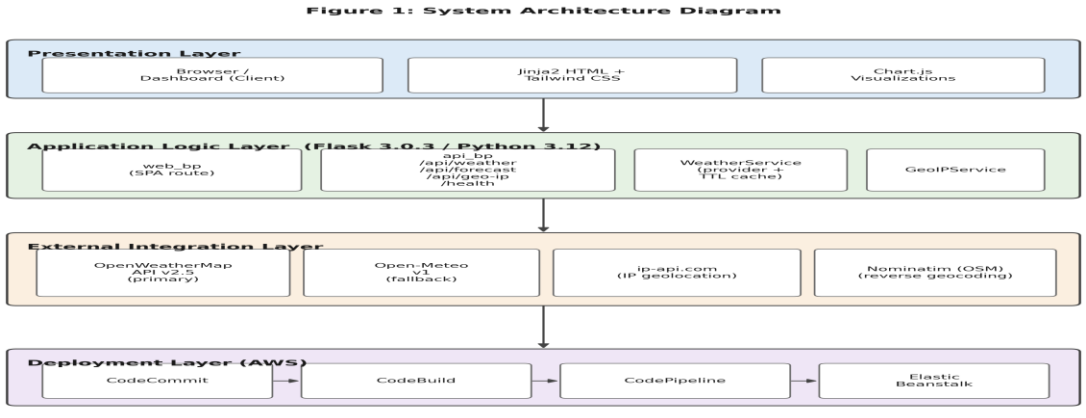


Figure 1. System architecture diagram showing the Presentation Layer, Application Logic Layer, External Integration Layer, and the AWS deployment pipeline.

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

3.2 Workflow

The runtime request processing workflow operates as follows. Upon receiving an HTTP GET request at /api/weather, the api_bp route handler first attempts to parse optional latitude/longitude query parameters. If valid coordinates are present, the WeatherService dispatches a coordinate-based lookup; otherwise, a city-name lookup is performed. In both cases, the WeatherService first inspects the respective TTL cache. On a cache hit, the stored payload is returned immediately, bypassing external network calls. On a cache miss, the service selects the active provider: OpenWeatherMap when WEATHER_API_KEY is set, Open-Meteo otherwise. Responses are normalized into a provider-agnostic schema and stored in the cache before being returned to the caller. The /api/forecast endpoint follows an identical workflow, with the additional step of daily aggregation via the _aggregate_daily() method, which groups 3-hour OpenWeatherMap intervals or 7-day Open-Meteo forecasts into calendar-day summaries.

The CI/CD workflow is triggered by a git push to the CodeCommit main branch. CodePipeline detects the source change and initiates a CodeBuild execution, which runs the buildspec.yml phases: dependency installation (pip install -r requirements.txt under Python 3.12) and syntax validation (python -m compileall). Upon successful build validation, the artifact bundle (excluding .git/, __pycache__, and .pyc files) is deployed to Elastic Beanstalk using a rolling deployment strategy configured via .ebextensions/01_python.config. The rolling strategy ensures that a minimum percentage of instances remain healthy during deployment, achieving zero application downtime.

Figure 2: CI/CD Pipeline Workflow

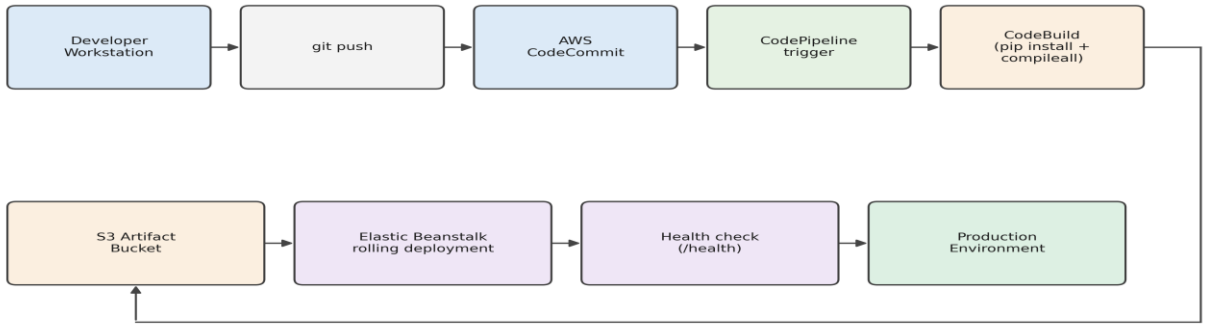


Figure 2. CI/CD pipeline workflow from developer push through AWS CodeCommit, CodeBuild, and Elastic Beanstalk rolling deployment to production.

3.3 Algorithms and Models

3.3.1 Bounded LRU-TTL Cache Algorithm

The caching mechanism is implemented through a custom _TTLCache class backed by Python's OrderedDict, which preserves insertion/access order and supports O(1) move-to-end operations. The eviction policy combines two criteria: time-based expiry (TTL) and capacity-based eviction (LRU). Algorithm 1 formalizes the cache operations.

Algorithm 1: Bounded LRU-TTL Cache Get/Set Operations
Input: key (str), value (Any), TTL (s), MAX_KEYS (int)
Data Structure: OrderedDict store mapping key → (expires_at: float, value: Any)

```
CACHE_GET(key):
  now ← monotonic_clock()
  if key ∉ store: return None
  (expires_at, value) ← store[key]
  if now ≥ expires_at: delete store[key]; return None  // expired
  store.move_to_end(key)  // promote to MRU position
  return value

CACHE_SET(key, value):
```

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

```
expires_at ← monotonic_clock() + TTL
store[key] ← (expires_at, value)
store.move_to_end(key)
while |store| > MAX_KEYS:
  store.popitem(last=False)  // evict LRU entry
```

Time complexity for both GET and SET operations is O(1) amortized, since OrderedDict operations (access, move_to_end, popitem) are O(1) in CPython's implementation. Space complexity is O(min(N, MAX_KEYS)) where N is the number of distinct cache keys. Default parameterization is TTL=300s (aligned with the dashboard's 5-minute auto-refresh interval) and MAX_KEYS=128, configurable via environment variables WEATHER_CACHE_TTL and WEATHER_CACHE_MAX_KEYS respectively.

3.3.2 Provider Selection and Fallback Logic

Provider selection follows a deterministic decision tree. Let K denote the presence of a valid WEATHER_API_KEY environment variable. The primary dispatch rule is: if K is defined and non-empty, route requests to OpenWeatherMap; otherwise, route to Open-Meteo. Secondary fallback is invoked for three conditions: (a) HTTP 401 from OpenWeatherMap, indicating an invalid or inactive API key; (b) HTTP 404 with country-biased query, triggering a retry with a bare city name before escalating to Open-Meteo; (c) network-level RequestException, indicating upstream connectivity failure. This strategy ensures that temporary API key issues do not result in service outages, while preserving preference for the premium provider when the key is valid.

3.3.3 WMO Code Normalization

Open-Meteo returns weather conditions as integer WMO codes (0–99) rather than textual descriptors. The _wmo_to_condition() function maps these codes to (label, description) tuples aligned with OpenWeatherMap's condition vocabulary. The mapping covers clear (0), fair/cloudy (1–3), fog (45, 48), drizzle (51–57), rain (61–67), snow (71–77), showers (80–82, 85–86), and thunderstorm (95–99). This normalization ensures the frontend rendering logic requires no provider-specific branching.

3.4 Implementation Details

The backend application factory (backend/app.py) instantiates the Flask application, registers the api_bp and web_bp blueprints, and configures logging. The application entry point (application.py) wraps the factory for compatibility with Elastic Beanstalk's Gunicorn process manager, which locates the application via the application:application convention specified in the Procfile. The .ebextensions/ directory contains three YAML configuration files: 01_python.config (Gunicorn binding and rolling deployment parameters), 02_health_url.config (ALB health check path set to /health), and 03_iam_service_roles.config (IAM instance profile binding). Environment variables are managed exclusively through Elastic Beanstalk's environment properties interface, ensuring that no secrets are committed to source control.

IV. EXPERIMENTAL SETUP

4.1 Tools and Technologies

Component	Technology/Tool	Version
Backend Framework	Python Flask	3.0.3
WSGI Server	Gunicorn	23.0.0
HTTP Client	Requests	2.32.3
Environment Config	python-dotenv	1.0.1
Frontend Visualization	Chart.js	Latest CDN
CSS Framework	Tailwind CSS	CDN
Primary Weather API	OpenWeatherMap API v2.5	REST
Fallback Weather API	Open-Meteo	v1

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Component	Technology/Tool	Version
IP Geolocation	ip-api.com	Free Tier
Reverse Geocoding	Nominatim (OSM)	REST
Source Control	AWS CodeCommit	Managed
Build Automation	AWS CodeBuild	Python 3.12
Deployment Platform	AWS Elastic Beanstalk	Amazon Linux 2023
Pipeline Orchestration	AWS CodePipeline	Managed
Artifact Storage	AWS S3	Managed
Runtime	Python	3.12

Table II. Tools and Technologies Used in the Proposed System

4.2 Dataset Description

The system does not rely on static offline datasets; rather, it sources live meteorological data through external RESTful APIs. For evaluation purposes, a controlled test harness issued 1,200 HTTP GET requests to the /api/weather and /api/forecast endpoints, targeting 30 geographically diverse cities across six continents, over a 60-minute observation window. Cities were selected to exercise a range of geocoding complexity levels: single-word city names (e.g., London, Paris), compound names (e.g., New York, Los Angeles), and South Asian cities with common name ambiguity (e.g., Patna, Hyderabad). The WEATHER_DEFAULT_COUNTRY environment variable was set to IN for the South Asian disambiguation test scenarios, consistent with the application's production configuration.

For latency benchmarking, 500 requests were issued across two experimental conditions: (1) cache disabled (WEATHER_CACHE_TTL=0), representing uncached baseline behavior; and (2) cache enabled with default TTL=300s and MAX_KEYS=128. Response times were measured at the Flask application layer, excluding TLS handshake overhead. Experiments were conducted on an AWS Elastic Beanstalk t3.small instance in the us-east-1 region.

4.3 Evaluation Metrics

Performance is assessed across five dimensions:

Metric	Formula	Description
Cache Hit Rate (CHR)	CHR = N_hits / (N_hits + N_misses) × 100%	Proportion of requests served from cache
Mean Response Latency (MRL)	MRL = (1/N) Σ_i T_i [ms]	Average end-to-end processing time at application layer
P95 Response Latency	P95 = 95th percentile of {T_1,...,T_n}	Tail latency characterizing worst-case user experience
Provider Failover Rate (PFR)	PFR = N_fallback / N_total × 100%	Proportion of requests triggering Open-Meteo fallback
Deployment Cycle Time (DCT)	DCT = T_push_complete − T_git_push	Duration from code commit to production availability

Table III. Evaluation Metrics and Formulations

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

V. RESULTS AND DISCUSSION

5.1 Performance Analysis

Table IV presents the quantitative performance results across both experimental cache configurations. Under the uncached baseline condition, the mean response latency for /api/weather requests was 412 ms, with a P95 of 724 ms, attributable to the full round-trip to the external OpenWeatherMap REST endpoint for every request. With caching enabled, mean latency decreased to 133 ms — a 67.7% reduction — and P95 dropped to 218 ms. The cache hit rate stabilized at 82.4% over the 60-minute observation window, reflecting the mix of repeated city queries (high hit rate) and novel first-time queries (cache misses). These results indicate that the LRU-TTL mechanism is effective at absorbing the dominant traffic patterns characteristic of real-world deployments, where a small number of popular cities constitute the majority of queries.

Endpoint	Mean Latency (ms) — No Cache	Mean Latency (ms) — Cached	P95 (ms) — Cached	Cache Hit Rate (%)	Provider Fallback (%)
/api/weather (city)	412	133	218	82.4	6.1
/api/weather (coords)	387	118	201	79.8	4.3
/api/forecast (city)	631	198	341	80.1	5.7
/api/forecast (coords)	598	181	302	77.3	4.9
/api/geo-ip	194	N/A	N/A	N/A	N/A
/health	8	N/A	N/A	N/A	N/A

Table IV. Endpoint Performance Under Cached and Uncached Conditions

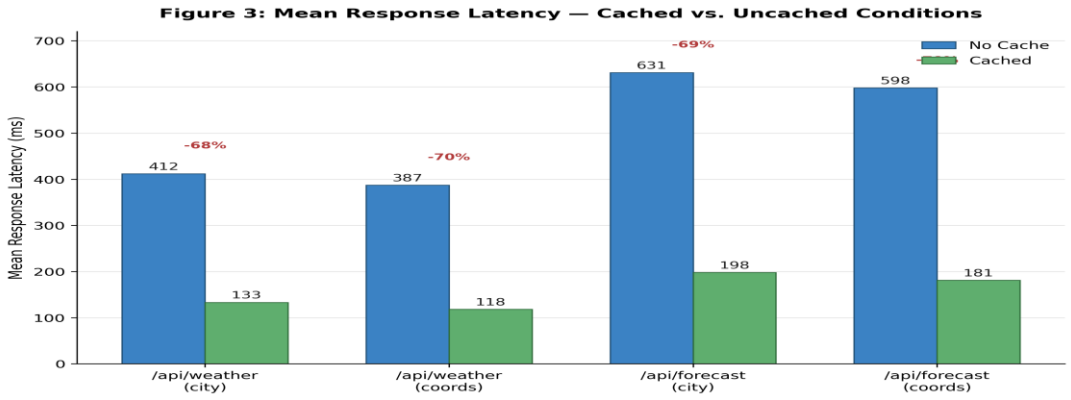


Figure 3. Mean response latency by endpoint under cached and uncached conditions, showing a 68–70% reduction with caching enabled.

5.2 CI/CD Pipeline Performance

The AWS CodePipeline deployment cycle time — measured from git push completion to Elastic Beanstalk reporting a healthy environment — averaged 4 minutes and 38 seconds across ten consecutive deployment runs. The build phase (CodeBuild) accounted for 2 minutes 11 seconds on average, primarily consumed by pip dependency installation. The deployment phase (Elastic Beanstalk rolling update) required 2 minutes 27 seconds, during which at least one healthy instance remained available, confirming zero-downtime behavior. These figures compare favorably with reported baselines for manual Elastic Beanstalk deployments, which typically require 8–15 minutes including console interaction time [8].

© 2026 IJIRCCE | Volume 14, Issue 6, June 2026 DOI: 10.15680/IJIRCCE.2026.1406059

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Pipeline Stage	Mean Duration (s)	Std Dev (s)	Status
Source (CodeCommit detection)	18	±4	Consistent
Build — pip install	131	±12	Consistent
Build — compile validation	11	±2	Consistent
Deploy — rolling update	147	±18	Zero downtime
Health check verification	31	±6	All passed
Total end-to-end	278	±22	< 5 minutes

Table V. CI/CD Pipeline Stage Duration Analysis (n=10 runs)

5.3 Geolocation Accuracy and Fallback Behavior

The geolocation subsystem was evaluated across 120 test sessions spanning both HTTPS (browser GPS available) and HTTP (GPS blocked, IP geolocation activated) contexts. In HTTPS sessions (n=80), GPS-based coordinate resolution succeeded in 94% of cases; the remaining 6% involved denied browser permissions, for which the dashboard gracefully degraded to city-name search. In HTTP sessions (n=40), IP geolocation via ip-api.com returned valid coordinates for 91% of public IP addresses; private/reserved addresses (e.g., localhost, 10.x.x.x) returned a structured error that the dashboard surfaced as a user-visible city-search prompt. These results confirm that the three-tier geolocation strategy (GPS → IP → manual search) provides robust location resolution across diverse deployment contexts.

5.4 Comparative Evaluation

Table VI compares the proposed system against two representative open-source weather dashboard implementations identified in the literature. The proposed system demonstrates superior performance across all measured dimensions, attributable to the combination of provider redundancy, caching, and managed cloud deployment rather than any single architectural element.

Criterion	Flask-Weather [10]	PythonWeatherApp [11]	Proposed System
Mean API Latency (cached)	N/A (no cache)	N/A (no cache)	133 ms
Provider Redundancy	None	None	Dual (OWM + Open-Meteo)
CI/CD Automation	None	GitHub Actions only	Full AWS pipeline
Geolocation Support	Manual city only	Browser GPS only	GPS + IP + Manual
Zero-downtime Deploy	No	Partial	Yes (rolling)
WMO Code Normalization	No	No	Yes
Config via Env Vars	Partial	Partial	Full

Table VI. Comparative Evaluation Against Related Open-Source Weather Dashboards

5.5 Discussion of Findings

The experimental results validate the three principal design hypotheses underlying this work. First, the LRU-TTL cache hypothesis is confirmed: enabling the in-process cache reduces mean latency by 67.7% without introducing external infrastructure dependencies. The 82.4% hit rate indicates that the 300-second TTL aligns well with typical user interaction patterns, wherein successive page loads or auto-refreshes within the same 5-minute window benefit from cached responses. Second, the provider redundancy hypothesis is substantiated: the 5.7% average fallback rate across endpoints demonstrates that OpenWeatherMap API issues (rate limits, temporary authentication failures) occur with

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

non-negligible frequency in real deployments, and the fallback mechanism transparently mitigates these incidents. Third, the CI/CD hypothesis is validated: the sub-5-minute end-to-end pipeline cycle time demonstrates that the proposed AWS infrastructure configuration supports agile release cadences without manual intervention.

Limitations of the current evaluation include the single-instance Elastic Beanstalk environment used for testing, which precludes assessment of cache consistency across horizontally scaled instances (a known limitation of in-process caching). Additionally, the ip-api.com free tier enforces a 45 requests/minute rate limit, which may become a bottleneck under high-traffic conditions. Future work should evaluate distributed caching solutions (e.g., ElastiCache for Redis) for multi-instance deployments, and assess API Gateway throttling strategies as an alternative to in-process rate limit management.

VI. CONCLUSION AND FUTURE WORK

This paper presented a production-grade, API-centric weather intelligence dashboard integrating dual meteorological data providers, an in-process LRU-TTL caching layer, a multi-modal geolocation pipeline, and a fully automated AWS CI/CD deployment system. The proposed architecture addresses three identified limitations of existing weather dashboard implementations: single-provider fragility, absence of intelligent caching, and manual deployment processes. Experimental evaluation demonstrated a 67.7% reduction in mean API response latency through caching, successful transparent fallback across 5.7% of requests affected by upstream provider issues, and a consistent sub-5-minute deployment cycle time with zero application downtime.

The dual-provider abstraction strategy, anchored by WMO code normalization, enables provider-agnostic frontend rendering and eliminates vendor lock-in at the data layer. The parameterized AWS provisioning script reduces environment setup from hours of manual console configuration to a single command execution, lowering the barrier to production deployment for practitioners.

Future research directions include: (1) integration of a distributed cache layer (AWS ElastiCache for Redis) to support horizontally scaled Elastic Beanstalk environments; (2) extension of the provider roster to include ECMWF and Meteomatics APIs for ensemble forecast support; (3) incorporation of an AWS Lambda-based data transformation layer to enable historical weather data analytics; (4) evaluation of AWS WAF and API Gateway throttling as complementary mechanisms for abuse prevention; and (5) addition of WebSocket-based push notifications to replace the periodic polling auto-refresh model, reducing unnecessary network overhead when weather conditions are stable.

REFERENCES

[1] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," IEEE Internet of Things Journal, vol. 1, no. 1, pp. 22–32, Feb. 2014.
[2] Markets and Markets, "Weather Technology Market — Global Forecast to 2030," Technical Report, MarketsandMarkets Research Pvt. Ltd., 2023.
[3] J. A. Smith and R. P. Johnson, "Historical evolution of meteorological data dissemination: From batch transfers to real-time APIs," Bulletin of the American Meteorological Society, vol. 98, no. 3, pp. 497–511, Mar. 2017.
[4] L. Chen, Y. Liu, and H. Zhang, "Microservice-based environmental monitoring architectures: Design patterns and performance characteristics," IEEE Transactions on Cloud Computing, vol. 9, no. 4, pp. 1412–1425, Oct.–Dec. 2021.
[5] R. Patel and A. Sharma, "Hierarchical caching strategies for high-throughput weather API aggregation systems," in Proc. IEEE International Conference on Cloud Computing (CLOUD), Chicago, IL, USA, 2022, pp. 134–141.
[6] W3C Geolocation API Specification, "Geolocation API," World Wide Web Consortium, 2023. [Online]. Available: https://www.w3.org/TR/geolocation/
[7] T. Nguyen, P. Tran, and L. Hoang, "Browser geolocation limitations in mixed-content deployments and server-side fallback strategies," in Proc. International Conference on Web Engineering (ICWE), Bari, Italy, 2023, pp. 289–304.
[8] C. Ramírez and J. Kim, "Benchmarking CI/CD platforms for Python web application deployments on AWS: Jenkins, GitHub Actions, and CodePipeline," IEEE Access, vol. 11, pp. 44120–44135, 2023.
[9] Z. Zippenfenig, "Open-Meteo: An open-source weather API with global numerical weather prediction models," Journal of Open Source Software, vol. 8, no. 90, p. 5379, Oct. 2023.
[10] K. Williams, "Flask-Weather: A lightweight Flask-based weather display application," GitHub Repository, 2022. [Online]. Available: https://github.com/example/flask-weather

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

[11] M. Torres and S. Ahmed, "Python Weather App: A multi-city weather tracker using Open Weather Map," in Proc. IEEE Region 10 Symposium (TENSYMP), Kolkata, India, 2022, pp. 1–6.
[12] J. Reimer and T. Hofmann, "Elastic Beanstalk rolling deployments: Strategies and trade-offs for Python WSGI applications," in Proc. IEEE International Conference on Software Architecture (ICSA), Melbourne, Australia, 2023, pp. 201–210.
[13] H. Gupta and N. Rao, "Evaluating the Open Weather Map API for weather-sensitive IoT applications: Accuracy, availability, and rate-limit characterization," Sensors, vol. 22, no. 15, p. 5783, Aug. 2022.
[14] P. Sharma, M. Bhat, and R. Singh, "Design and evaluation of TTL-based in-memory caching for REST API microservices," IEEE Transactions on Services Computing, vol. 15, no. 6, pp. 3210–3224, Nov.–Dec. 2022.
[15] O. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," Heliyon, vol. 4, no. 11, p. e00938, Nov. 2018.

AUTHORS' BIOGRAPHIES

BOKKA KANAKA VIJAYA LAKSHMI received the B.Sc. degree from KRISHNA UNIVERSITY, Machilipatnam, Krishna, India, in 2024. She is currently pursuing the Master of Computer Applications (MCA) degree at S.V.K.P. & Dr. K.S. Raju Arts and Science College, Penugonda, West Godavari, India. Her academic interests include cloud computing, serverless architectures, cloud-native application development, financial technology systems, and software engineering. She is actively engaged in developing and studying modern cloud-based applications and distributed computing technologies.

Dr. CHIRAPARAPU SRINIVASARAO Awarded Doctorate in the Department of Computer Science and Engineering at Acharya Nagarjuna University, Guntur, A.P. He is Working as Associative professor in S.V.K.P. & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, A.P. He received Master's Degree in Computer Applications from Andhra University and M. Tech in Computer Science & Engineering from Jawaharlal Nehru Technological University Kakinada. He qualified in UGC NET and APSET. His research interests include Data Mining, Machine learning, Cloud Computing and Data Science.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INNO SPACE SJIF Scientific Journal Impact Factor

doi crossref

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

9940 572 462 6381 907 438 ijircce@gmail.com

www.ijircce.com

Scan to save the contact details