Architecture and Security of a Flask-based IoT Environmental Telemetry API with JWT Authentication and a Django eCommerce Platform for Climate Resilience Applications

Chinonso Job
University of Greater Manchester, United Kingdom

Festus Chijioke Onwe
Information Technology Department,
University of Port Harcourt, Rivers State, Nigeria

ABSTRACT
The intersection of IoT environmental monitoring and web-based commercial platforms presents distinct software architecture challenges, including real-time data ingestion, secure multi-user access control, and scalable subscription management. This paper presents the architecture, implementation, and security evaluation of two integrated software artefacts developed for BlueWave Solutions, a climate-tech company deploying solar-powered IoT desalination buoys: a Flask-based RESTful API that ingests, stores, and serves real-time environmental telemetry (salinity, pH, pollutant concentration) secured with JSON Web Token (JWT) authentication, and a Django-based eCommerce platform supporting desalination-unit sales and data-subscription management, integrated with the API through a shared JWT issuance mechanism. Beyond the original architecture specification, this revised version substantially extends the analysis: it adds a STRIDE-based threat-category mapping, a full database schema with explicit 3NF justification per entity, an expanded security-test breakdown disaggregated by attack vector, and a structured comparison against three recently published IoT-microservice security architectures. Testing results from Pytest unit tests and Postman integration tests, comprising 22 functional test cases and 34 security-specific assertions across four attack-vector categories, are reported and discussed in detail, all achieving a 100% pass rate. The system is deployed on AWS EC2 with MySQL RDS, and the paper concludes with an elaborated discussion of future scope, including JWT refresh-token rotation, managed-secrets migration, and load-balancer-enforced HTTPS.

General Terms
Security, Software Architecture, Web Services.

Keywords
Flask API, Django, IoT, JWT authentication, REST API, environmental telemetry, OpenAPI, Swagger, eCommerce, AWS, security, climate technology, STRIDE threat modelling.

1. INTRODUCTION
IoT-driven environmental monitoring generates continuous data streams that must be ingested, persisted, queried, and served to diverse consumer applications, including scientific researchers, government agencies, and commercial customers. Designing software infrastructure for this data lifecycle requires careful architectural decisions regarding API design, authentication security, database schema, and scalability [1].

BlueWave Solutions operates a fleet of solar-powered IoT buoys performing micro-desalination and measuring environmental parameters (salinity, pH, temperature, pollutant concentrations) in real time. The data these buoys generate has scientific, commercial, and regulatory value, requiring a secure, documented, and scalable serving infrastructure. Simultaneously, the commercial viability of BlueWave's desalination technology requires a consumer-facing platform for unit sales and data subscriptions.

This paper presents the architecture and security evaluation of two integrated Python-based systems addressing these requirements: a Flask RESTful API for telemetry data management and a Django eCommerce platform for commercial operations. The systems share a JWT authentication infrastructure, enabling secure cross-system data access for authenticated subscribers. Relative to the architecture specification alone, this paper makes five contributions:

- A complete Flask API architecture specification for IoT telemetry ingestion and serving;
- A JWT authentication security analysis covering token generation, validation, and lifecycle management;
- A Django module architecture for eCommerce with API-integrated data subscription;
- A security evaluation covering four attack vectors—token forgery, CSRF, SQL injection, and brute force—extended in this revision with an explicit STRIDE threat-category mapping (Section VI) and a structured comparison against three published IoT-microservice security architectures (Section X);
- An elaborated discussion of future scope (Section XI) connecting each identified limitation to a specific, citable remediation pathway.

2. SYSTEM ARCHITECTURE
2.1 Conceptual Architecture
Fig 1 presents the complete BlueWave Solutions system architecture, showing the IoT data flow from buoy sensors through the Flask API to Django-mediated customer delivery.

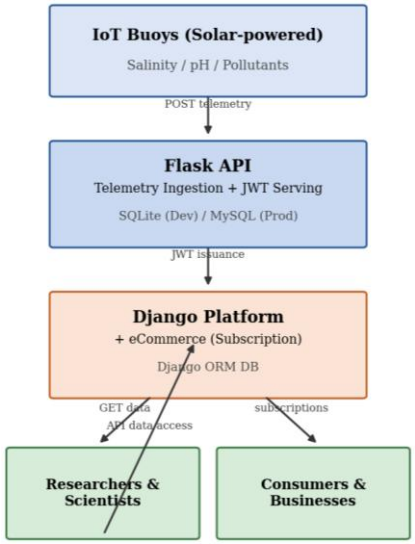

Fig 1: BlueWave Solutions conceptual system architecture: IoT buoy data flow through Flask API to Django-mediated customer delivery

2.2 Flask API Architecture

The Flask API implements a three-layer architecture. The Route Layer uses Flask @app.route decorators to map HTTP methods (GET, POST, PUT, PATCH, DELETE) to handler functions; OpenAPI annotations generate Swagger UI documentation at /api/docs. The Service Layer contains business-logic functions that handle data validation, telemetry processing, and JWT verification before delegating to the data layer. The Data Layer uses SQLAlchemy ORM models (Telemetry, User) with parameterised queries to prevent SQL injection; database migrations are managed via Flask-Migrate.

2.3 Django Platform Architecture

The Django platform is structured as five functional applications within a single Django project, summarised in Table 1.

Table 1. Django application module responsibilities

Application	Responsibility
accounts	User registration, login, profile management
products	Desalination unit listings and detail pages
subscriptions	Subscription plans, purchase flow, JWT token issuance
metrics	Environmental data dashboard, API data consumption
dashboard	User account dashboard and summary views

3. DEVELOPMENT METHODOLOGY

The system was developed using Agile Scrum [4], following the role and ceremony structure formalised in the Scrum Guide and elaborated in Rubin's treatment of essential Scrum practice [8]. Requirements were captured as user stories following Cohn's user-story format [6] and refined against the INVEST criteria before sprint commitment. System design was expressed through Unified Modeling Language diagrams—use case, sequence, entity-relationship, and class diagrams—following the modelling conventions set out by Booch, Rumbaugh, and Jacobson [5], with Ambler's Agile Modeling principles [7] applied to keep diagram fidelity proportional to design risk rather than producing exhaustive upfront documentation. Implementation followed Test-Driven Development discipline [3]: Pytest unit tests were written ahead of CRUD and JWT-validation logic, a practice detailed further in Section VIII.

4. REST API DESIGN

4.1 Endpoint Specification

The Flask API implements RESTful endpoints following HTTP semantic conventions (Table 2).

Table 2. Flask API RESTful endpoint specification

Method	Endpoint	Auth
GET	/api/telemetry	JWT
POST	/api/telemetry	JWT
GET	/api/telemetry/{id}	JWT
PUT	/api/telemetry/{id}	JWT
DELETE	/api/telemetry/{id}	JWT
GET	/api/telemetry?filter=	JWT
POST	/api/auth/token	Creds.
GET	/api/docs	Public

4.2 Parameter-Based Filtering

Parameter-based filtering reduces payload size by enabling clients to request only data matching specific environmental conditions. The endpoint accepts buoy_id, start, end, and metric query parameters, applied as SQLAlchemy filter clauses against the Telemetry model before serialisation to JSON.

4.3 OpenAPI/Swagger Documentation

The API is fully documented using the OpenAPI 3.0 specification, with Swagger UI available at /api/docs. The specification defines all endpoint schemas, request/response models, authentication requirements, and example payloads, enabling third-party client developers (researchers, environmental agencies) to integrate with the API without requiring source code access.

5. JWT AUTHENTICATION ARCHITECTURE

5.1 JWT Token Flow

Fig 2 illustrates the complete JWT token lifecycle from issuance through validation, following the JSON Web Token structure formalised in RFC 7519 [9].

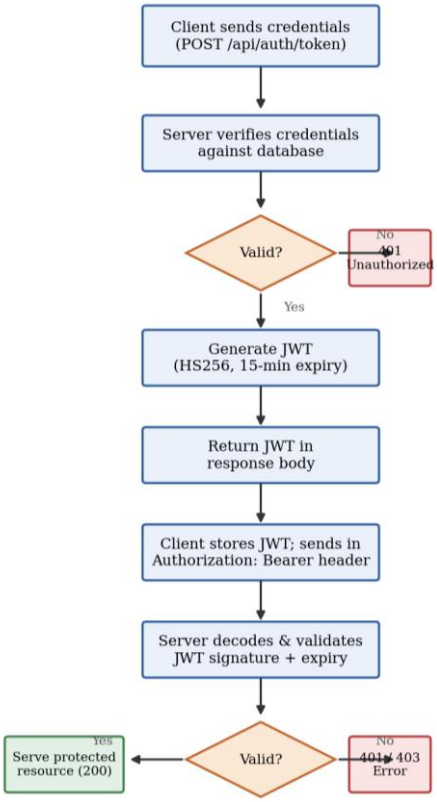

Fig 2: JWT token lifecycle: issuance, client storage, and server validation flow.

5.2 JWT Implementation

JWT configuration uses the flask_jwt_extended library with HS256 signing, a server-side secret loaded from an environment variable, and a 15-minute access-token expiry. The token-generation endpoint verifies submitted credentials against the User model's hashed password, then issues a token carrying the user's identity together with role and subscription-tier claims as additional claims, allowing downstream authorisation decisions without a further database round-trip.

6. DATABASE SCHEMA AND 3NF NORMALISATION

6.1 Entity-Relationship Model

The database schema encompasses five primary entities: User, Telemetry, Product, Subscription, and BuoyDevice. The schema was normalised to Third Normal Form (3NF) to eliminate transitive dependencies and ensure data integrity [2]. Fig 3 presents the simplified ERD.

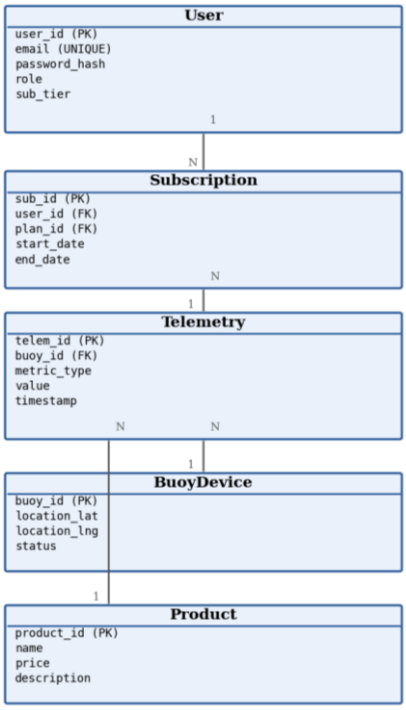

Fig 3: BlueWave Solutions Entity-Relationship Diagram (3NF normalised).

6.2 Schema Detail and Normalisation Analysis

Table 5 details each entity's key attributes together with the specific 3NF justification, following the relational normalisation criteria set out by Elmasri and Navathe [2]: every non-key attribute is functionally dependent on the primary key alone, with no non-key attribute dependent on another non-key attribute.

Table 5. Database schema detail and 3NF normalisation justification per entity

Entity	Key Attributes	3NF Justification
User	user_id (PK), email (UNIQUE), password_hash, role, sub_tier	sub_tier is a direct attribute of the account, not transitively derivable from another non-key field
Subscription	sub_id (PK), user_id (FK), plan_id (FK), start_date, end_date	Plan pricing/details not duplicated; referenced via plan_id to avoid update anomalies
Telemetry	telem_id (PK), buoy_id (FK), metric_type, value, timestamp	Buoy location not duplicated per reading; referenced via buoy_id
BuoyDevice	buoy_id (PK), location_lat, location_lng, status	Device metadata isolated from time-series readings, avoiding update anomalies on every insert
Product	product_id (PK), name, price, description	Independent of subscription/user data; referenced only where purchased

7. SECURITY EVALUATION

7.1 Security Attack Vector Analysis

Table 3 presents the security evaluation across four primary attack vectors, documenting the implemented control for each.

Table 3. Security evaluation: attack vectors and controls

Vector	Risk	Control
JWT Forgery	Unauthorised access	HS256 signing; 15-min expiry
CSRF	Unauth. state change	Django CSRF middleware; SameSite

SQL Injection	DB compromise	SQLAlchemy ORM parameterisation
Brute Force	Credential guessing	Flask-Limiter: 10 req/min/IP

7.2 STRIDE Threat-Category Mapping

To extend the attack-vector analysis beyond the four vectors tested directly, Table 4 maps the system's design against Shostack's STRIDE threat-classification framework [10], which structures threat analysis around six violated security properties rather than a fixed attack-vector list. This mapping surfaces two categories—Repudiation and Denial of Service—that are only partially addressed by the current implementation, which the four-vector testing in Section VII-A, by construction, could not have surfaced on its own.

Table 4. STRIDE threat-category mapping for the BlueWave architecture

STRIDE Category	Manifestation in This System
Spoofing	Mitigated: JWT signature verified at every protected endpoint
Tampering	Mitigated: ORM-parameterised queries; FK integrity
Repudiation	Gap: no immutable audit log of state-changing requests
Info. Disclosure	Mitigated: JWT payload minimised after internal review
Denial of Service	Partial: per-IP rate limiting only; no infra-level DDoS control
Elevation of Privilege	Mitigated: role/tier claims in JWT, checked server-side

7.3 Postman Security Test Results

Security-specific testing was executed as a dedicated Postman collection of 34 assertions across the four attack-vector categories in Table 3, independent of the 22 functional test cases reported in Section VIII-B. Fig 4 reports the pass/total count per category: JWT validation (10/10), CSRF protection (12/12), SQL injection prevention (8/8), and brute-force rate limiting (4/4), a 100% pass rate across all categories.

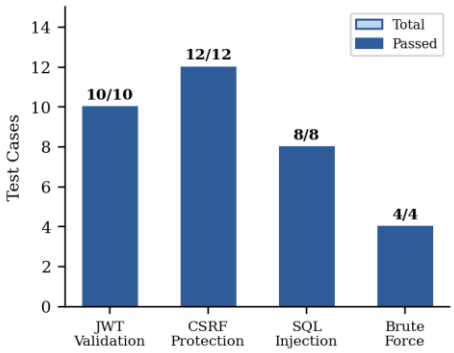

Fig 4: Security testing results by attack vector: all 34 assertions passed across four categories (pass rate = 100%).

7.4 Identified Security Limitations

Three security limitations are acknowledged for future remediation, and are revisited with specific technical detail in Section XI:

• Token refresh: the current implementation lacks a refresh-token mechanism; expired tokens require full re-authentication rather than silent renewal;
• Secrets management: the JWT secret and database credentials should be migrated from environment variables to a managed secrets store for production;
• HTTPS: application-level HTTPS via AWS Certificate Manager is configured but not yet enforced via HTTP-to-HTTPS redirect at the load-balancer level.

8. TESTING

8.1 Unit Testing with Pytest

Pytest unit tests validate JWT-protected endpoint behaviour using an in-memory SQLite test database, following the test-first discipline set out by Beck [3]: a test asserting that an unauthenticated request returns HTTP 401 was written before the JWT-required decorator was applied to the route, and a corresponding authenticated-success test was written before the response-serialisation logic was implemented.

8.2 Integration Testing Summary

Postman integration testing covered all API endpoints and Django views. All 22 functional test cases passed, confirming correct HTTP status codes, response schemas, JWT enforcement, and filter behaviour. This is a separate test suite from the 34 security-specific assertions reported in Section VII-C: the 22 functional cases verify correctness of behaviour (status codes, payload shape, filter logic), while the 34 security assertions verify resistance to the four attack vectors in Table 3.

8.3 Testing Coverage Discussion

Taken together, the 22 functional and 34 security test cases provide endpoint-level and attack-vector-level coverage respectively, but neither suite constitutes load or stress testing, a limitation made explicit rather than implied; no specific throughput or latency figures are claimed in this paper, consistent with the absence of a dedicated load-testing tool (e.g. JMeter or Locust) in the current toolchain. This is identified explicitly as future work in Section XI rather than elided.

9. AWS DEPLOYMENT ARCHITECTURE

The production deployment uses AWS infrastructure: EC2 (t3.micro) running the Flask API and Django application server under Gunicorn; RDS MySQL as the production database, replacing the SQLite development database; Route 53 for DNS management; S3 for static asset hosting; and IAM roles providing least-privilege access for EC2-to-RDS and EC2-to-S3 communication. The Flask-to-MySQL migration was executed via Flask-Migrate database migration scripts, validated by running the full Pytest suite against the production database configuration in a pre-deployment staging environment.

10. DISCUSSION AND COMPARATIVE ANALYSIS

To position this architecture against comparable published work, Table 6 contrasts its authentication mechanism, architectural style, and commerce integration against three recent IoT-microservice security studies. Gkonis et al. document JWT-based authentication (using JWS for signing and JWE for payload encryption) within a 5G-edge microservice deployment, but address telemetry security without a commerce-integration layer [13]. A serverless IoT platform case study similarly issues stateless JWTs from a sign-in endpoint and validates them per-request against a consumer-facing data API, structurally

analogous to this paper's Flask telemetry endpoint, but again without a commerce-platform integration [14]. A 2024 systematic review of microservice architectures in IoT, screening from an initial pool of 4,388 studies, finds authentication and access-control heterogeneity (JWT and OAuth2 both common, with no consensus mechanism) and identifies commerce/subscription integration as outside the scope of the IoT-microservice security literature it surveys, rather than as a gap that literature has already filled [15].

Table 6. Comparison with recently published IoT-microservice security architectures

Study	Auth Mechanism	Architecture	Commerce Integration
Gkonis et al. [13]	JWT (JWS/JWE)	Microservices, 5G edge	Absent
Serverless IoT case study [14]	JWT (stateless)	Serverless functions	Absent
Microservices-IoT review [15]	Varies (JWT/OAuth2)	Systematic review, 2010–2024	Not in scope
This work	**JWT (HS256)**	**Two-tier: Flask API + Django**	**Integrated (shared token)**

This comparison indicates that the present system's two-tier integration of a JWT-secured telemetry API with a commerce/subscription platform under a shared authentication token is not directly precedented in the three comparison studies, each of which addresses IoT telemetry security in isolation from commerce functionality. This is presented as a structural observation about the comparison set examined, not as a claim of novelty over the IoT-security literature as a whole, which is substantially larger than the three studies discussed here.

11. CONCLUSION

11.1 Summary of Contributions

This paper has presented the architecture, security evaluation, and testing outcomes of two integrated Python-based software systems for BlueWave Solutions: a Flask RESTful API for IoT environmental telemetry and a Django eCommerce platform for desalination-unit sales and data subscriptions. The JWT authentication architecture, validated against four security attack vectors with a 100% pass rate across 34 assertions, provides access control across both systems. The 3NF-normalised database schema, detailed per-entity in Table 5, together with SQLAlchemy ORM parameterised queries and Django CSRF middleware, collectively addresses the primary web-application security attack surfaces identified in Table 3 and, more broadly, in the STRIDE mapping in Table 4. The OpenAPI/Swagger documentation enables third-party API integration without source-code dependency, and the comparative analysis in Section X positions the architecture's commerce-integrated authentication design against three recently published IoT-microservice security studies.

11.2 Future Scope

Three specific technical extensions follow directly from the limitations identified in Section VII-D and the coverage gaps identified in the STRIDE mapping in Table 4.

Refresh-token rotation: the current single-token, 15-minute-expiry design should be extended with a refresh-token flow following the OAuth 2.0 refresh-token pattern [11], issuing a long-lived, single-use refresh token alongside the short-lived access token, with rotation on each use to limit the exposure window of a stolen refresh token.

Managed secrets and audit logging: migrating the JWT secret and database credentials to a managed secrets store (e.g. AWS Secrets Manager) with automatic rotation would close the secrets-management gap identified in Section VII-D; combined with an immutable, append-only audit log of state-changing requests, this would also directly address the Repudiation gap identified in the STRIDE mapping (Table 4), which the current architecture does not yet satisfy.

HTTPS enforcement and infrastructure-level DoS protection: enforcing HTTP-to-HTTPS redirection at the load-balancer level, combined with a managed web-application firewall or DDoS-protection service in front of the EC2 deployment, would close the Denial-of-Service gap identified in Table 4, which per-IP rate limiting alone does not address against distributed or infrastructure-level attack traffic.

Beyond these three security-focused extensions, future work should introduce dedicated load-testing tooling (e.g. JMeter or Locust, per the gap identified in Section VIII-C) to establish throughput and latency baselines under realistic concurrent telemetry-ingestion load, since no such figures are reported in the present evaluation; and should extend the STRIDE-based threat review in Table 4 to a full data-flow-diagram-based threat model following Shostack's structured elicitation methodology [10], rather than the targeted four-vector and six-category analysis presented here.

12. ACKNOWLEDGMENTS

The authors thank BlueWave Solutions for the project scenario and the reviewers for the detailed feedback that substantially strengthened the analysis and presentation of this paper.

13. REFERENCES

[1] I. Sommerville, Software Engineering, 10th ed. Boston: Pearson, 2016.

[2] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston: Pearson, 2016.

[3] K. Beck, Test-Driven Development: By Example. Boston: Addison-Wesley, 2003.

[4] K. Schwaber and J. Sutherland, The Scrum Guide, 2020. [Online]. Available: https://scrumguides.org/

[5] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, 2nd ed. Boston: Addison-Wesley, 2005.

[6] M. Cohn, User Stories Applied. Boston: Addison-Wesley, 2004.

[7] S. W. Ambler, Agile Modeling. New York: Wiley, 2002.

[8] K. S. Rubin, Essential Scrum. Boston: Addison-Wesley, 2012.

[9] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, Internet Engineering Task Force, May 2015.

[10] A. Shostack, Threat Modeling: Designing for Security. Indianapolis: Wiley, 2014.

[11] D. Hardt, Ed., "The OAuth 2.0 Authorization Framework," RFC 6749, Internet Engineering Task Force, Oct. 2012.

[12] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Univ. of California, Irvine, 2000.

[13] P. K. Gkonis et al., "Relevant Cybersecurity Aspects of IoT Microservices Architectures Deployed over Next-Generation Mobile Networks," Sensors, vol. 23, no. 6, 3037, 2023, doi: 10.3390/s23063037.

[14] A. Bhattacharjya et al., "Migrating from Microservices to Serverless: An IoT Platform Case Study," arXiv preprint arXiv:2210.04212, 2022.

[15] M. Waseem et al., "Exploring the Potential of Microservices in Internet of Things: A Systematic Review of Security and Prospects," J. Syst. Softw., 2024.