ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Architecture and Security Design of a Dual-Service Iot Telemetry and Ecommerce Platform: A Flask–Django Case Study with JWT Authentication and Layered Test Automation

Nwokpuru Samuel Abafu[1], Chinonso Job[2], Onwe, Festus Chijioke[3]

[1]Department of Computing, Software Engineering, university of greater manchester

[2]Department of Computer Science, University of greater Manchester, United Kingdom.

[3]Information Technology Department, University of Port Harcourt, Rivers State, Nigeria.

DOI: https://doi.org/10.51583/IJLTEMAS.2026.150700021

Received: 12 June 2026; Accepted: 17 July 2026; Published: 06 August 2026

ABSTRACT

This paper documents the architecture, security design, and database evolution of a two-tier system built to support an environmental-telemetry and micro-desalination business scenario, comprising a Flask REST API for Internet of Things (IoT) buoy telemetry ingestion and a Django eCommerce platform for product sales and subscription management. We describe the system's modular separation of concerns (a stateless, JWT-secured ingestion API decoupled from a session-oriented commerce platform), the specific security controls applied at each layer (JWT-based stateless authentication for the API; Django's built-in Cross-Site Request Forgery and SQL-injection middleware for the platform), and the database migration from SQLite to MySQL undertaken specifically in response to a measured concurrency limitation under bulk telemetry ingestion. We report the integration mechanism connecting the two otherwise loosely coupled services (token-based cross-service authentication and shared schema conventions) and the specific coordination overhead this integration introduced. We position the architecture against established microservice design literature and REST architectural-style principles, and we provide a candid account of which design decisions are well supported by that literature and which represent pragmatic compromises specific to a timeand resource-constrained development context, together with the concrete changes (CI/CD pipeline integration, containerisation, formal load testing) that would be required before the architecture could be considered production-ready rather than a validated prototype.

Keywords—Software architecture, REST API design, JWT authentication, microservices, Flask, Django, database migration, IoT telemetry systems.

INTRODUCTION

Internet of Things (IoT) telemetry systems and customer-facing eCommerce platforms are frequently built as separate services with distinct architectural requirements: telemetry ingestion typically prioritises throughput, statelessness, and machine-to-machine authentication, while customer acing commerce platforms prioritise session management, user experience, and human-facing authentication flows [1], [2]. This paper documents the architecture of a system that deliberately implements both service types separately— a Flask-based telemetry API and a Django-based eCommerce platform—while integrating them through a shared authentication token and consistent schema conventions, for

N. S. Abafu is with the Department of Computing, Software Engineering. Manuscript prepared for submission; corresponding author email to be inserted by the author prior to submission.

an environmental-monitoring and micro-desalination business scenario in which IoT buoys transmit telemetry data that both feeds an operational API and informs the commerce platform's product and subscription content.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

We report the system's architecture, the specific security controls implemented at each layer, the database evolution undertaken during development, and the integration mechanism connecting the two services, and we evaluate each design decision against the relevant architectural and security literature. Consistent with the system's status as a validated academic prototype rather than a production deployment, we explicitly identify which findings are well supported by the literature and which represent context-specific pragmatic compromises, and we specify the concrete changes required before the architecture could be considered production-ready.

Contributions

- A documented two-tier architecture separating stateless, high-throughput telemetry ingestion from sessionoriented commerce functionality, evaluated against REST architectural-style principles [1] and microservice design literature [2].
- A security design account specifying the JWT-based stateless authentication mechanism for the API [3] and the Django middleware-based protections for the commerce platform [4], evaluated against OWASP's Top Ten guidance [5].
- A documented database migration (SQLite to MySQL) motivated by a measured concurrency limitation, with the specific evidence that motivated the migration and the schema-consistency issues the migration introduced.
- A candid assessment of the specific architectural decisions requiring further work—absent CI/CD integration, absent containerisation, informal rather than tool-based load validation—before the system could be considered production-ready.

Related Work

REST Architectural Style and Microservice Decomposition

Fielding's foundational specification of REST architectural constraints—statelessness, a uniform interface, and resourcebased addressing—provides the design basis for the telemetry API's CRUD-and-filtering interface [1]. Newman's treatment of microservice design emphasises that service boundaries should be drawn around independently deployable, independently scalable units of business capability [2]; the present system's separation of telemetry ingestion from commerce functionality follows this principle, since the two services have materially different throughput, statefulness, and scaling characteristics, discussed further in Section III.

Token-Based Authentication

JSON Web Tokens, as specified in RFC 7519, provide a compact, self-contained mechanism for representing authentication claims that can be validated by a resource server without a round-trip to a central session store, making them well suited to stateless API authentication of exactly the kind the telemetry API requires [3]. This stands in contrast to the session-based authentication Django's default user-authentication framework provides for the commerce platform, motivating the integration design discussed in Section V.

Database Concurrency and Migration Literature

Stonebraker's analysis of SQL versus NoSQL database trade-offs notes that lightweight, file-based relational databases such as SQLite are well suited to development and lowconcurrency contexts but are explicitly not designed for the write-concurrency profile multi-client production systems require [6], directly relevant to the migration motivation documented in Section III-B. Fawzy et al.'s practitioner survey on data management in Agile contexts identifies schema evolution during active development as a recurring source of friction precisely in projects undergoing exactly this kind of midproject database migration [7].

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

System Architecture

Two-Tier Service Decomposition

The system comprises two independently deployable services. The Flask telemetry API exposes CRUD operations, parameter-based filtering (e.g. by date range and geographic location), and bulk-ingestion endpoints for IoT buoy telemetry data (salinity, pH, dissolved oxygen, and related environmental parameters), returning JSON-formatted responses and documented via an OpenAPI 3.1 specification rendered through Swagger UI [8]. The Django eCommerce platform exposes product listings for micro-desalination units, subscriptionmanagement workflows for ongoing telemetry data access, and user account management, rendered through server-side templates with Bootstrap-based responsive styling. Table I summarises the architectural contrast between the two services.

This decomposition follows Newman's principle of drawing service boundaries around units with distinct scaling and statefulness profiles [2]: the telemetry API's bulk-ingestion endpoint must handle bursty, potentially large-volume writes from IoT devices, while the commerce platform's request profile is steady and human-paced, and conflating the two into a single service would force a single scaling and statefulness model onto workloads that do not share one.

TABLE I

Architectural Contrast Between the Two Services

Dimension	Flask Telemetry API	Django eCommerce Platform
Statefulness	Stateless (perrequest JWT validation)	Session-oriented (authenticated user sessions)
Primary clients	IoT buoys, machine clients, dashboards	Humanusers via browser
Throughput profile	Bursty, bulk-ingestion capable	Steady, human-paced request rate
Authentication [3]	JWT bearer tokens	Django session/cookiebased auth
Documentation [8]	OpenAPI/Swagger	N/A (server-rendered UI)

Database Layer

The system used SQLite during initial development for its zero-configuration, file-based simplicity, consistent with SQLite's well-documented suitability for development and low-concurrency contexts [6]. Informal load testing using Postman bulk-request scripts simulating concurrent telemetry ingestion identified write-concurrency failures under SQLite at request volumes representative of a moderate-scale buoy deployment, directly consistent with the concurrency limitation Stonebraker's analysis predicts for file-based relational databases under multi-client write load [6]. This finding motivated migration to MySQL [9], after which the same bulkingestion test scripts completed successfully with measurably improved throughput. The migration was not without cost: schema inconsistencies between the SQLite-era and MySQLera Object-Relational Mapping configurations broke several existing Pytest fixtures, requiring schema refactoring and testfixture updates—a friction point directly consistent with Fawzy et al.'s finding that schema evolution is a recurring source of difficulty in actively developed Agile projects [7].

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Security Design

Stateless API Authentication

The telemetry API's security model relies on JWT bearer tokens issued at login and validated server-side on every subsequent request, following RFC 7519's specification for self-contained authentication claims [3]. Token validation logic explicitly rejects expired or malformed tokens with a 401 Unauthorized response, and token payloads were deliberately minimised to exclude sensitive fields after an internal review identified that an early implementation had included unnecessary sensitive data in the token payload—an instance of the general JWT security guidance that token payloads, while not encrypted by default, should be treated as readable by any party in possession of the token and scoped accordingly.

Commerce Platform Middleware Protections

The Django eCommerce platform relies on Django's builtin Cross-Site Request Forgery protection middleware and parameterised-query-based Object-Relational Mapping to prevent SQL injection, consistent with Django's documented security architecture [4] and with OWASP's Top Ten guidance identifying both vulnerability classes as high-priority risks for web applications handling user-submitted forms and payment-adjacent data [5]. Input-validation testing confirmed that script-injection attempts submitted through subscription and product-listing forms were correctly rejected.

Security Gaps Relative to Comprehensive Coverage

Consistent with OWASP's recommendation that automated security scanning (e.g. OWASP ZAP) be integrated into the testing process rather than applied selectively and manually [5], the security testing performed on this system—JWT penetration testing, CSRF/SQL-injection middleware validation, and targeted input-validation testing—did not include automated, systematic fuzz testing across the full input space of either service. This is a specific, identifiable gap relative to comprehensive industry-standard security validation, distinct from the security controls themselves, which are correctly implemented for the threat classes they address.

Cross-Service Integration

The two services are loosely coupled but require specific integration points to function as a coherent system. JWT tokens issued by the Flask API are validated within Djangoside workflows that display telemetry-derived content (e.g. environmental metrics shown alongside product listings), requiring the Django platform to implement JWT validation logic in addition to its native session-based authentication—two parallel authentication mechanisms operating within the same platform. Database schema conventions (field naming, data types for shared entities such as subscription identifiers) were maintained consistently across both services' independently managed databases to support this integration without a shared database layer, which would have violated the independentdeployability principle motivating the two-tier decomposition in the first place [2]. This integration pattern introduced measurable coordination overhead: changes to the API's response schema required corresponding updates to the Django-side consumption logic, and the two services' independent test suites did not automatically catch schema drift between them, a limitation discussed further in Section VI.

DISCUSSION: PRODUCTION-READINESS ASSESSMENT

We assess the architecture's production-readiness candidly, distinguishing well-supported design decisions from contextspecific compromises.

Well supported by the architectural literature: the twotier service decomposition and JWT-based API authentication. Both decisions follow established principles—REST statelessness [1], microservice boundary-drawing around distinct scaling profiles [2], and standards-based token authentication [3]—rather than representing ad hoc or idiosyncratic choices, and we would expect both decisions to generalise well to comparable systems.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Reasonable but context-specific: the database migration timing and the cross-service schema-consistency mechanism. Migrating from SQLite to MySQL only after a measured concurrency failure, rather than provisioning for productionscale concurrency from project initiation, was a reasonable response to genuine time and resource constraints, but a production system would more typically provision its production database technology from initiation rather than migrating reactively; we report this as a transparent account of an academic-context trade-off rather than as a recommended general practice. The schema-consistency mechanism (manually maintained naming and type conventions across two independently managed databases, Section V) is a knownfragile pattern relative to either a shared schema registry or a contract-testing approach, and the schema-drift risk it introduces is a specific, identified item for the future-work list below.

Required before production-readiness: CI/CD integration, containerisation, and load-testing-tool-based validation. Consistent with the testing-process findings of a companion case study examining this project's Agile and risk-management practice, the absence of automated CI/CD pipelines, the use of informal rather than dedicated loadtesting tools, and the absence of containerisation (Docker) for environment consistency are the three concrete changes that would be required to move this architecture from a validated academic prototype to a production-deployable system. None of these gaps reflects an architectural design flaw; each reflects time and resource constraints explicitly acknowledged during the project's own retrospective process.

CONCLUSION

This paper has documented the architecture, security design, and database evolution of a two-tier IoT telemetry and eCommerce system, evaluating its design decisions against REST architectural-style, microservice-design, and JWT-authentication literature. The two-tier service decomposition and token-based API authentication are well supported by established principles and would be expected to generalise to comparable systems; the database migration timing and the manually maintained cross-service schema-consistency mechanism are reasonable, time-constrained academic-context compromises rather than recommended general practice; and CI/CD integration, containerisation, and load-testingtool-based validation are identified as the specific, concrete changes required before the architecture could be considered production-ready. We have reported the specific evidence (a measured SQLite concurrency failure under simulated bulk ingestion) that motivated the most consequential architectural change made during development, providing a level of artefact-level detail that purely descriptive system reports frequently omit.

Limitations

This paper documents a single system built by a single small academic team; no comparative architecture (e.g. an alternative service-decomposition or database choice applied to the same requirements) was built or evaluated, and the productionreadiness assessment in Section VI is the project team's own critical evaluation rather than an independent architectural audit. The performance findings reported in Section III-B derive from informal load-simulation scripts rather than a dedicated load-testing tool, a limitation explicitly carried into the production-readiness assessment rather than treated as resolved.

REFERENCES

1. R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
2. S. Newman, Building Microservices: Designing Fine-Grained Systems. Sebastopol, CA: O'Reilly Media, 2015.
3. M. Jones, J. Bradley, and N. Sakimura, "RFC 7519: JSON web token (JWT)," Internet Engineering Task Force (IETF), 2015.
4. Django Software Foundation, "Django security documentation: Cross site request forgery protection," https://docs.djangoproject.com/en/stable/ref/ csrf/, 2024.
5. OWASP Foundation, "OWASP top ten 2021," https://owasp.org/Top10/, 2021.
6. M. Stonebraker, "SQL databases v. NoSQL databases," Communications of the ACM, vol. 53, no. 4, pp. 10–11, 2010.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

7. M. Fawzy, A. Tahir, M. Galster et al., “Exploring data management challenges and solutions in agile software development: A literature review and practitioner survey,” Empirical Software Engineering, vol. 30, p. 106, 2025.
8. OpenAPI Initiative, “OpenAPI specification v3.1.0,” https://spec. openapis.org/oas/v3.1.0, 2021.
9. M. Widenius and D. Axmark, MySQL Reference Manual. Sebastopol, CA: O’Reilly Media, 2002.