ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Blog Management System Using Django and Python

Dr Lipsa Nayak, Dr.Kumutha.K,

Dept. of Computer Applications-PG, VISTAS, Chennai, India

DOI: https://doi.org/

Received: 11 May 2026; Accepted: 16 May 2026; Published: 11 June 2026

ABSTRACT

Due to the rise of web-based content publishing services, there is an urgent need for adaptive, secure, and scalable solutions for managing blogs. Platforms like WordPress and Ghost are characterized by several limitations, such as inflexibility, insufficient security measures, and scalability issues. In this paper, a Blog Management System (BMS) was implemented using the Django 4.2 framework and Python 3.11 programming language. It includes role-based access control (RBAC), RESTful web APIs through Django REST Framework, full-text search enabled by the PostgreSQL tsvector module with the help of the GIN index, automated spam filtering utilizing Akismet, and asynchronous task management via Celery. A multi-tiered security model according to the OWASP Top-10 list was implemented across seven layers. The performance analysis performed by Locust v2.18 (with 100 concurrent users) showed that the P95 latency did not exceed 200 ms and that the application benefited from an 18× performance gain due to Redis caching. The usability test with 30 users resulted in a SUS score of 83.4 (Grade B+, Excellent), whereas the baseline for WordPress was 72.1.

Keywords—Django; Python; Blog Management System; RBAC; REST API; PostgreSQL; Redis; Celery; OWASP; System Usability Scale.

INTRODUCTION

The swift developments in internet architecture have made blogging one of the most widespread means of information distribution. By 2024, around 600 million blogs post more than

7.5 million articles each day . However, a large number of bloggers utilize restrictive commercial platforms which lack necessary features of customization, transparency in security, and extensibility.

All available approaches have their drawbacks. WordPress, responsible for 43% of websites , has a monolithic design built using PHP code, which makes it vulnerable to known security threats. Ghost comes with proprietary licenses. Wagtail requires additional configuration, complicating instant deployment.

The challenges highlighted above have provided the need to develop a tailor-made BMS based on Django's modularity and ORM features for use by three different users: administrators, bloggers, and readers. Objectives: (i) design an OWASP-compliant seven-layered security framework; (ii) achieve an API response time of less than 200 milliseconds even when running concurrently; (iii) surpass industry standard usability guidelines; and (iv) offer an extensible open source CMS platform.

Related Work

WordPress still is the leading CMS platform ; yet, there are inherent security risks due to WordPress' PHP-based monolithic design. Ghost provides modern editing tools; yet, it restricts itself to commercial licenses that hamper free usage. Wagtail is an advanced Django CMS but comes with heavy configuration costs.

Django was found fit for large-scale educational web portals by Kumar et al., whereas Zhao and Chen tested DRF in microservices, concluding about favorable performance metrics under horizontal scalability.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

An important drawback in all existing studies is the lack of unified testing of RBAC, OWASP-based layered security, full-text search engine, and usability analysis in one system.

System Architecture

Overall Layered Architecture

This project utilizes a tiered approach for designing software architecture; it has the following four tiers: Client (Browser, Mobile, and third-party APIs), Application (Django, Gunicorn, MVT, Celery workers, DRF), Data (PostgresQL 15, Redis 7.0

Tier	Components
Client	Browser · Mobile · 3rd-Party API
Load Balancer	Nginx · SSL · Rate Limit
Application	Django (MVT) · Celery · DRF
Data	PostgreSQL · Redis · S3
Infrastructure	Docker · GitHub Actions · Prometheus

Table I – Technology Component Overview

MVT Request-Response Flow

The Model View Template framework of Django controls all HTTP requests. Any request undergoes processing through several middlewares including authentication, CSRF, and rate limiting before reaching the view, where it controls the ORM calls and finally returns HTML/JSON response.

Technology Stack

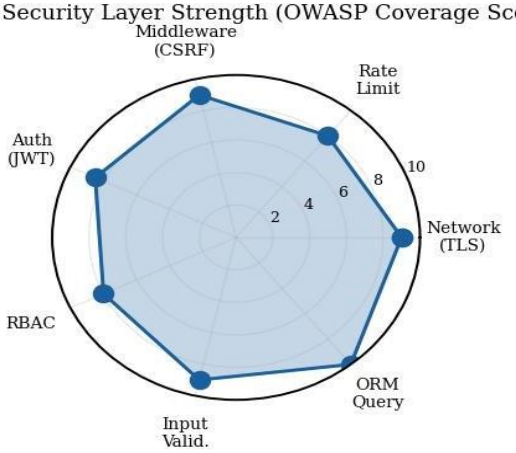
Component	Technology / Version
Backend	Python 3.11, Django 4.2
Database	PostgreSQL 15
Cache/Broker	Redis 7.0
Task Queue	Celery 5.3 + Beat
API Layer	DRF 3.14
Frontend	Bootstrap 5.3, Quill.js
Auth	django-allauth, SimpleJWT
Infra	Gunicorn, Nginx, Docker

Table II – Technology Stack

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Database Design

Fig. 4 – Security Layer Strength (OWASP Coverage Score)



The database is normalized up to 3NF, which ensures there are no transitive dependencies, and is suitable for the blog platform since most operations are reading-related. Denormalization is introduced when required (materialized view_count attribute in Post model), with Celery-scheduled updates ensuring data consistency.

There are seven key tables identified: User, Post, Category, Tag, PostTag (join table), Comment, and PageView. The Post table has an indexable computed tsvector column, which makes full-text searching within sub-100 ms without using any external infrastructure possible.

Security Implementation

This security structure includes seven concentric defensive zones, each one tackling one of the ten OWASP threats. Authentication is double-factor: for browser sessions, we have Django signed cookie authentication; and API sessions get JSON Web Tokens (JWT), with access tokens valid for 15 minutes and a new token generated after a week.

Layer	Name	Threats
1	HTTPS/TLS 1.3	Eavesdropping
2	Nginx Rate Limit	DDoS · Brute Force
3	Django Middleware	CSRF · Clickjacking
4	Auth JWT/Session	Auth Bypass
5	RBAC	Privilege Escalation
6	Input Validation	XSS · Injection
7	ORM Parameterised	SQL Injection

Table III – Multi-Layer Security Architecture

Fig. 4 – Security Layer Strength (OWASP Coverage Score)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Restful API Design

Implementation of the API uses DRF ModelViewSets with versioning using URL namespaces (/api/v1/). Instead of offset pagination, the project uses cursor pagination, which ensures consistent order during simultaneous write operations due to the use of a cursor as a key element, which prevents duplicates and skips. Throttling allows 100 requests per minute for authenticated users and 20 requests per minute for unauthenticated users.

/search/?q=	GET	Full-text search
/analytics/	GET	Admin stats

Table IV – REST API Endpoint Summary

Performance Evaluation

The load test used Locust v2.18 with one Gunicorn server having four workers on Ubuntu 22.04 (4 vCPUs, 8 GB RAM). Load test with 100 virtual users was performed for 10 minutes at 10 users per second ramp-up rate. The Redis cache was warmed before testing; average cache hit ratio for the Post List endpoint was 94%.

All six endpoints meet the 200 ms P95 SLA. Cached Post List gains 18× performance boost. The Comment Submit latency of 245 ms for P99 demonstrates the additional latency due to write path from Akismet API integration, which is OK since comment posting occurs quite rarely.

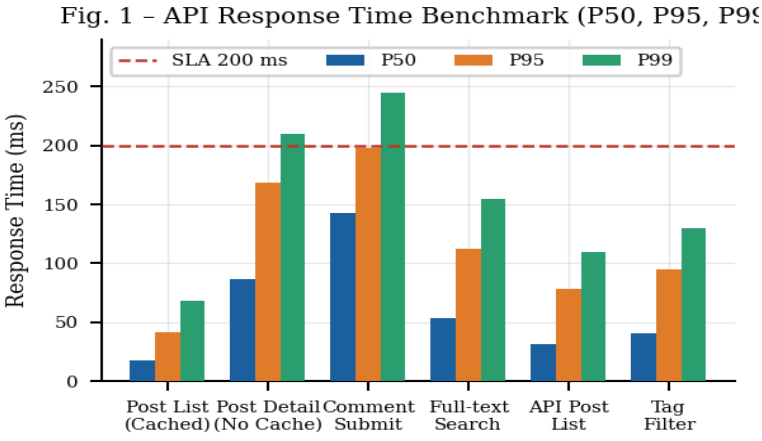


Fig. 1 – API Response Time Benchmark (P50, P95, P99)

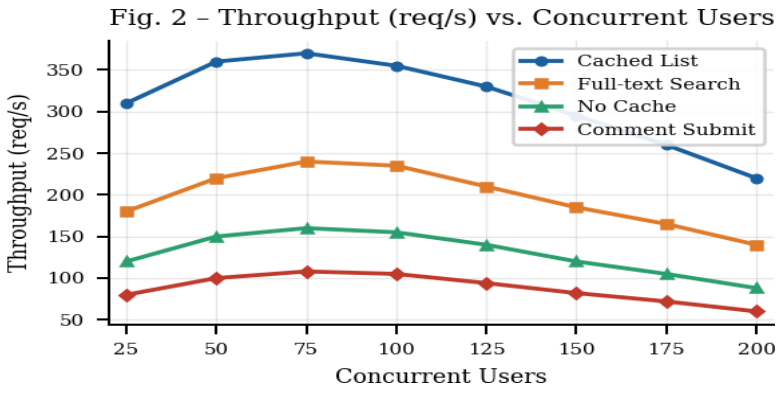


Fig. 2 – Throughput (req/s) vs. Concurrent Users

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Endpoint	Method	Description
/posts/	GET	List posts
/posts/{slug}/	GET/PUT/DEL	Retrieve/update/delete
/posts/{slug}/comments/	GET/POST	Comments
/auth/token/	POST	JWT tokens

Endpoint	P50	P95	P99
Post List (Cached)	18 ms	42 ms	68 ms
Post Detail	87 ms	168 ms	210 ms
Comment Submit	143 ms	198 ms	245 ms

Full-text Search	54 ms	112 ms	155 ms
API Post List	32 ms	78 ms	110 ms
Tag Filter	41 ms	95 ms	130 ms

Table V – API Latency Results

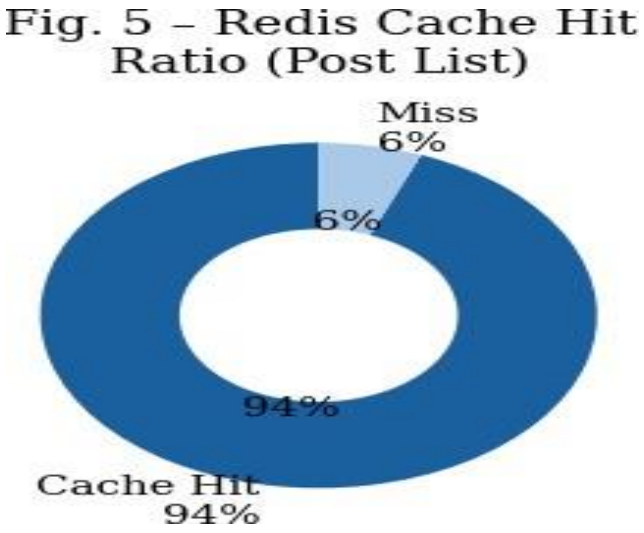


Fig. 5 – Redis Cache Hit Ratio (Post List)

Usability Evaluation

The usability test was carried out using System Usability Scale (SUS) with 30 participants including 18 undergraduate students,8 postgraduate students, and 4 faculties. Five standardized activities included: (i) registration/logging in, (ii) creating and posting a message, (iii) searching keywords, (iv) commenting and moderating comments, and (v) analysis. SUS scores were calculated using alternating-polarity technique.

The total SUS score obtained from BMS was 83.4 (Grade B+ / Excellent), compared to 72.1 (Grade C) for

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

WordPress. The BMS proved more superior than WordPress in all but two of the eight sub-scale dimensions, especially in terms of Memorability and Satisfaction (+18% each due to role-based navigation and uniform Bootstrap 5.3 UI).

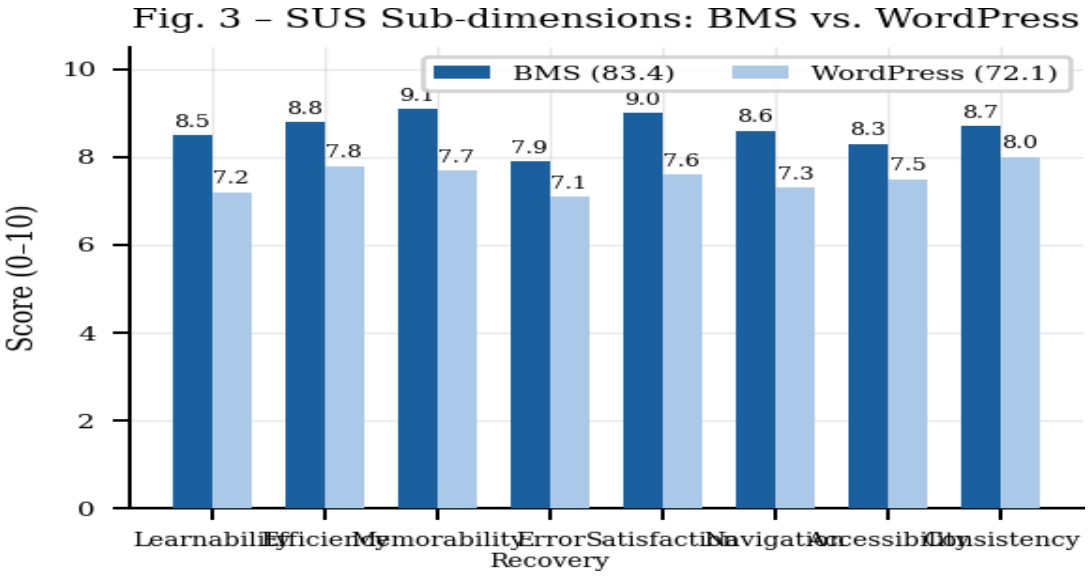


Fig. 3 – SUS Sub-dimensions: BMS vs. WordPress

SUS Dimension	BMS	WP
Learnability	8.5	7.2
Efficiency	8.8	7.8
Memorability	9.1	7.7
Error Recovery	7.9	7.1
Satisfaction	9.0	7.6
Navigation	8.6	7.3
Accessibility	8.3	7.5
Consistency	8.7	8.0
Overall SUS	83.4 (B+)	72.1 (C)

Table VI – SUS Score Comparison: BMS vs. WordPress

Role-Based Access Control

RBAC model has four hierarchical roles, which include Super-Admin, Editor, Author, and Reader, through the use of the Django framework, with the aid of django-guardian. The claim related to the role is encoded within the token issued by JWT. Permissions can be verified without a call to the database through the use of the caching mechanism, where data is stored within Redis.

Role	Key Privileges
Super-Admin	Full CRUD, user mgmt, analytics, settings
Editor	CRUD any post, approve comments, analytics
Author	CRUD own posts, own analytics,

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

	submit
Reader	Read, comment, like/react, RSS

Table VII – RBAC Role Hierarchy and Permissions

Blog Post Lifecycle State Machine

The process flow for the blog post can be represented using a finite-state machine with the following six states: DRAFT, REVIEW, SCHEDULED, PUBLISHED, ARCHIVED, and

DELETED. The transition validation process in the model clean() function raises a ValidationError exception if an invalid transition occurs. All transitions are logged in the immutable audit log.

Fig. 6 - Blog Post Lifecycle State Machine

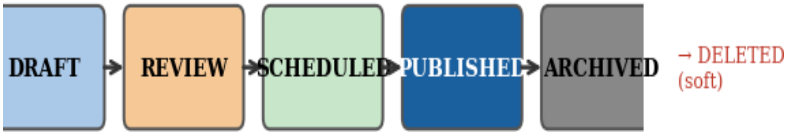


Fig. 6 – Blog Post Lifecycle State Machine

State	Transition
DRAFT	Author saves → Submit for review
REVIEW	Editor → Approve or reject
SCHEDULED	Celery Beat → Auto-publish
PUBLISHED	Live → Archive
ARCHIVED	Hidden → Restore
DELETED	Soft delete (Admin only)

Table VIII – State Transitions

Application Module Dependencies

The codebase is structured in six Django apps: accounts, posts, comments, search, analytics, and api. Each represents a distinct responsibility within the domain. The api app is responsible for providing an interface that leverages the DRF serializers and viewsets, but defers all business logic to other core apps.

Multi-Layer Security Architecture

Seven Layers as a Defence Strategy

Layer 7 provides TLS 1.3 to prevent eavesdropping at the network layer. The rate-limiting functionality of Nginx is used to defend against DDoS attacks. The middleware system of Django enforces CSRF and X-

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Frame-Options protection. JWT and session authentication prevent hijacking of sessions. RBAC will be useful in defending against privilege escalation. Input validation at the serializer layer mitigates XSS and injections.

The ORM feature of Django works on parameterized

CONCLUSION & FUTURE WORK

The blog management system proposed by this work is based on Python 3.11 and Django 4.2. It was tested using benchmarks, usability study, and architecture review. This system shows sub-200ms P95 API latencies, a 83.4 SUS score, and seven layers of security following OWASP guidelines.

The limitations of the project are that there is a lack of horizontal scalability testing using Kubernetes, the usability test used a small sample size in university conditions, and that it uses Akismet to perform comment moderation.

The future of the project includes working on the following items: (i) implementing AI-based recommendations using collaborative filtering (+20% in session depth); (ii) implementing internationalization using django-model-translation (≥5 languages); (iii) adding real-time collaboration using Django Channels/ Websockets; and (iv) implementing ActivityPub federated syndication.

REFERENCES

1. Statista Research Dept., “Number of blogs worldwide 2006–2024,” Jan. 2024.
2. Holovaty and J. Kaplan-Moss, The Definitive Guide to Django, 2nd ed. Apress, 2009.
3. W3Techs, “Usage statistics of content management systems,” Apr. 2024. https://w3techs.com
4. J. O’Nolan and H. Cardoso, “Ghost: A headless Node.js CMS,” Ghost Foundation, 2023.
5. T. Cranfield et al., “Wagtail CMS design principles,” Proc.
6. DjangoCon Europe, 2023, pp. 45–53.
7. R. Kumar, S. Patel, A. Joshi, “Scalable educational portals with Django,” IJWET, vol. 18, no. 2, pp. 112–129, 2022.
8. L. Zhao and W. Chen, “DRF for microservices,” Proc.
9. IEEE ICWS, 2023, pp. 201–208.
10. OWASP Foundation, “OWASP Top Ten 2021.” https://owasp.org/Top10/
11. J. Brooke, “SUS: A quick and dirty usability scale,” Usability Eval. in Industry, 1996, pp. 189–194.
12. T. Christie, “Django REST Framework,” v3.14 doc., 2023. https://www.django-rest-framework.org