POST-HUB: A Unified Web-Based Platform for Multi-Channel Social Media Content Management and Scheduling

Tanishq Gosavi, Kritarth Vala, Rajveersinh Chauhan, Neh Panchal
Department of Computer Science and Engineering
Parul Institute of Technology, Parul University, Vadodara, Gujarat, India
Under the guidance of Mr. Gautam Singh, Assistant Professor (AI/ML)

***Abstract*—*Post-Hub is a full-stack web application that lets a user compose, publish, and schedule content across multiple social-media platforms from a single authenticated dashboard, instead of repeating the same composing, publishing, and reporting work separately inside each platform's own tools. The system is built with a React.js front end, a Java Spring Boot back end, and MySQL for persistent storage, following a clean controller–service–repository architecture. Security is centred on JWT-based stateless authentication and BCrypt password hashing, supported by request validation, CORS handling, and centralised error handling. The system offers full CRUD post management, an automatic background scheduler that publishes due posts every sixty seconds, an aggregated engagement-analytics endpoint, and a tiered Free/Pro/Enterprise subscription model. Real publishing to Facebook, Instagram, Twitter/X, and LinkedIn is implemented as clearly isolated stub methods behind a single dispatch point, so that live OAuth integrations can be added later without altering the rest of the architecture. Twelve functional test cases covering authentication, post lifecycle, scheduling, ownership checks, and analytics aggregation were executed against the running system, all of which passed.*

Keywords*—*Social Media Automation, Unified Content Publishing, Post Scheduling, JWT Authentication, REST API, Spring Boot, React.js, MySQL, Engagement Analytics, Subscription Model

I. Introduction

Maintaining an active presence across platforms such as Facebook, Instagram, Twitter/X, and LinkedIn requires a user to log into each platform separately, reformat the same content to suit that platform, publish or schedule it individually, and later revisit each platform again to check performance. Post-Hub is proposed as a single, unified dashboard that removes this repetition: a user registers once, connects social accounts, composes a post — with optional platform-specific variations — chooses target platforms, and either publishes immediately or schedules it for a future date and time. A consolidated analytics view aggregates engagement across all platforms into one summary.

Content creators and social-media managers who operate on more than one platform face recurring problems: duplicated composing effort, no single calendar for what is scheduled and when, fragmented analytics spread across each platform's own dashboard, missed posting windows in the absence of a cross-platform scheduler, and no tiered access model that reflects how differently freelancers, small teams, and agencies use such a tool. Post-Hub addresses these problems by modelling the entire content lifecycle — compose, target, schedule or publish, and analyse — as a single authenticated workflow backed by a secure, layered server-side architecture.

The system is implemented using React (front end) and Java Spring Boot (back end) with MySQL as the database, and is deliberately structured so that real platform integrations (Twitter API v2, Facebook Graph API, Instagram Graph API, and the LinkedIn API) can be plugged into a single, well-defined dispatch point without altering the rest of the system.

II. Literature Review

Buffer and Hootsuite are the most widely used commercial social-media scheduling platforms; both allow a user to connect multiple accounts via OAuth, compose a post once, choose target platforms, schedule a publish time, and view basic post-publish analytics. Later focuses on visual, Instagram-first scheduling with a drag-and-drop calendar, while SproutSocial targets larger marketing teams with approval workflows and deeper analytics. Studying the public documentation of these tools — rather than their closed source code — informed the feature set adopted for Post-Hub: a composer with per-platform content, a scheduler, a unified analytics summary, and tiered plans, summarised in Table 1.

TABLE 1
Comparative Study of Existing Social-Media-Management Tools

Tool	Core Strength	Limitation
Buffer	Simple composer UI; browser extension	Free tier limited to 3 channels / 10 posts
Hootsuite	Wide platform support; team streams	Steeper learning curve; costly for teams
Later	Strong visual / Instagram-first calendar	Weak support for text-first platforms
SproutSocial	Deep analytics; approval workflows	Enterprise pricing, not student-friendly

Each target platform exposes a distinct publishing API, and understanding their shape directly influenced the Post-Hub data model. Twitter API v2 creates a tweet with a single POST to /2/tweets under OAuth 2.0 user-context authentication; the Facebook Graph API publishes with a POST to /{page-id}/feed using a page-scoped access token; the Instagram Graph API requires a two-step container-then-publish flow routed through the same Graph API infrastructure; and the LinkedIn API publishes UGC posts via a POST to /ugcPosts under the w_member_social scope. Because each API has a different request shape and required scopes, Post-Hub stores an optional per-platform content override alongside a default content string, so a single post can still be tailored to each destination.

React supplies the client-side single-page interface; Spring Boot provides the REST back end, security configuration, and scheduled-job infrastructure; MySQL is used as the relational store, with normalised tables and foreign keys so that users own posts, posts can target multiple platforms, and analytics can be associated with posts and platforms; and JWT together with BCrypt provides stateless authentication and one-way password hashing.

III. Research Methodology

A. Existing (Manual) Methodology

In the absence of a dedicated tool, a small team typically maintains a shared spreadsheet as an informal content calendar, writes each caption once, and then logs into each platform's own native composer to copy, adapt, and publish or schedule the content individually. Engagement figures are later collected by visiting each platform's own analytics page and manually copying the numbers into a spreadsheet if a combined view is needed — a process that does not scale past a handful of platforms and provides no single source of truth.

B. Proposed Methodology

Post-Hub was developed using an iterative, layered approach that moved from data modelling outward to the API and finally the UI: (i) requirement analysis translated the feature list into concrete use cases; (ii) the User and Post data structures were designed first, since every other layer depends on their shape; (iii) the Spring Boot REST API and JWT security filter chain were built and verified independently with Postman before any UI was connected; (iv) the React SPA was then built page-by-page against the already-working back end through a shared apiCall helper that attaches the JWT and centralises error handling; (v) integration and debugging surfaced and resolved three concrete defects, discussed in Section VI; (vi) the scheduled publisher and analytics-aggregation endpoint were added once the core post CRUD flow was verified; and (vii) functional test cases were executed against the running system to verify authentication, post lifecycle, scheduling, and analytics aggregation. Verifying the API independently before wiring up the front end removed an entire class of "is it a front-end or back-end bug" ambiguity during integration.

For a published post, the aggregated engagement score across all target platforms is computed by summing the recorded likes, shares, and comments for every platform p in the post's platform set:

$$E\_total = \Sigma_{p \in P} (L_p + S_p + C_p) \quad (1)$$

where L_p, S_p, and C_p denote the likes, shares, and comments recorded for platform p, and P is the set of platforms the post was published to. The engagement rate reported on the Analytics page is then obtained by relating this total to the reach figure recorded for the same post:

$$R = (E\_total / Reach) \times 100\% \quad (2)$$

C. Subscription-Plan Methodology

The Free/Pro/Enterprise plan structure gates four dimensions of usage — connected social accounts, monthly post volume, availability of scheduling, and depth of analytics — plus team and API access at the top tier, mirroring the freemium model used by every commercial competitor surveyed in Section II. It is implemented as a single plan field on the User record, updated through PUT /api/auth/plan, deliberately left as a straightforward field update rather than a payment integration, since billing was scoped out of the current implementation.

TABLE 2

Subscription Plan Feature Matrix

Feature	Free	Pro	Enterprise
Social accounts	3	10	Unlimited
Posts / month	10	Unlimited	Unlimited
Scheduling	Yes	Yes	Yes
Analytics	Basic	Advanced	Full suite
Team access	No	No	Yes
API access	No	No	Yes

IV. Proposed System Architecture

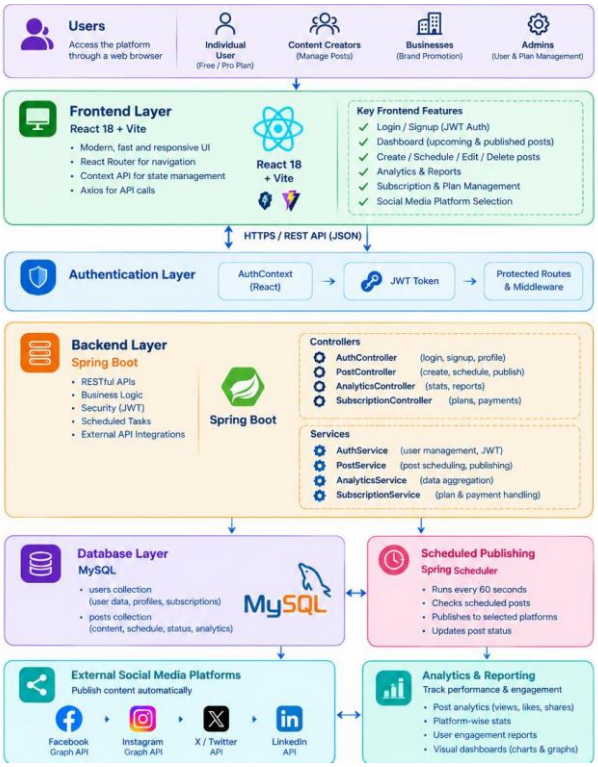

Fig. 1. *Post-Hub system architecture, showing the frontend, authentication, backend, database, scheduled-publishing, and external-platform layers.*

A. Layered Architecture

The system follows a clean client–server architecture. The React 18 + Vite front end handles routing, state management through Context API, and API calls through Axios; the authentication layer wraps every request with a JWT token validated by protected-route middleware; the Spring Boot back end exposes RESTful controllers (Auth, Post, Analytics, Subscription) backed by corresponding services; MySQL persists user, post, subscription, and analytics data; and a Spring Scheduler component ticks every sixty seconds to check for and publish due posts, dispatching to the external platform layer and updating the analytics/reporting layer in turn, as shown in Fig. 1.

B. Use Case Model

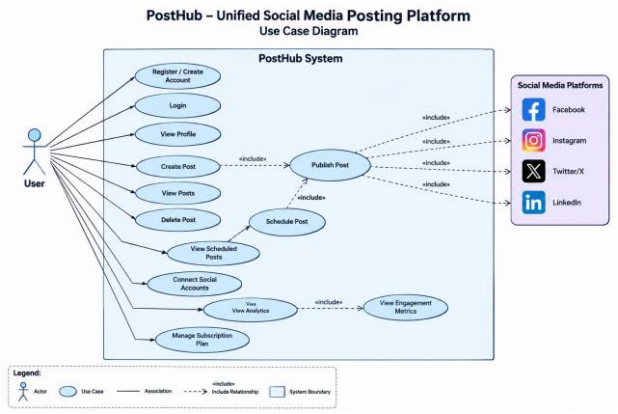

Fig. 2. *Use case diagram: the registered User composes, publishes, schedules, and analyses posts, while Publish Post includes each of the four target-platform destinations.*

The primary actor is the registered user, who can register, log in, compose a post (including target-platform selection), publish immediately or schedule for later, view the dashboard and analytics, manage connected accounts, manage a subscription plan, and delete posts. A secondary, automated actor — the Scheduler — triggers the auto-publish use case every sixty seconds for any post whose scheduled time has passed, independent of whether the user is currently online, as shown in Fig. 2.

C. Database Design

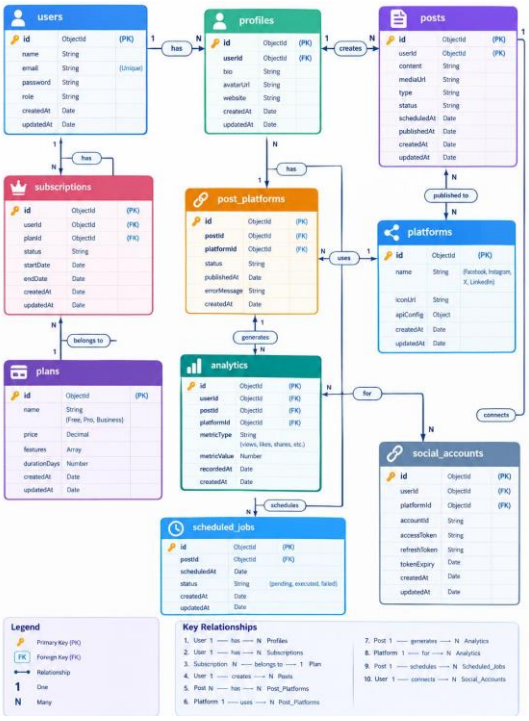

Fig. 3. *Entity-relationship diagram of the MySQL schema, showing users, profiles, posts, platforms, post_platforms, analytics, subscriptions, plans, scheduled_jobs, and social_accounts.*

MySQL is used as the relational store for structured application data. The users table holds account details and credentials; posts holds content, media references, target platforms, and status; post_platforms is a join table recording per-platform publish status; analytics stores views, likes, shares, and comments per post and platform; scheduled_jobs tracks pending, executed, and failed publish jobs; and subscriptions/plans manage the tiered access model of Table 2. Primary and foreign keys enforce referential integrity across the ten key relationships shown in Fig. 3, supporting efficient querying alongside post scheduling, analytics, and subscription handling.

V. Implementation and Results

A. Core Modules

The back end follows a conventional Controller → Service → Repository layering. The authentication module hashes passwords with BCryptPasswordEncoder and issues an HS256-signed JWT valid for 24 hours; a JwtAuthFilter validates this token on every subsequent request and populates the security context. The post module branches post creation on whether a scheduledAt value is supplied, storing the post as scheduled or immediately dispatching it to publishToPlatforms(). A single dispatch method iterates the post's target platforms and calls one of four isolated stub methods per platform, each documented with the exact production API call it would make, with per-platform exception handling so that a failure on one platform never blocks the others. The analytics module aggregates every post's embedded per-platform metrics into the totals used by Equations (1)–(2), and the front end centralises all API communication through a single apiCall helper exposed via React Context.

TABLE 3
Representative REST API Endpoints

Endpoint	Auth	Description
POST /api/auth/register	No	Create a new account
POST /api/auth/login	No	Authenticate, receive JWT
POST /api/posts	Yes	Create / publish / schedule a post
GET /api/posts/scheduled	Yes	List the user's scheduled posts
DELETE /api/posts/{id}	Yes	Delete a post owned by the user
GET /api/analytics/summary	Yes	Aggregated engagement metrics
PUT /api/auth/plan	Yes	Update subscription plan

B. Functional Testing

Twelve functional test cases were executed against the running system through both Postman and the connected React front end. Table 4 summarises a representative subset.

TABLE 4
Representative Functional Test Cases

Test Case	Expected Result	Result
Register with duplicate email	HTTP 400, “email already in use”	Pass
Login with incorrect password	HTTP 401, invalid credentials	Pass
Access protected route, no token	HTTP 401 Unauthorized	Pass
Create post with future scheduledAt	Status set to scheduled	Pass
Scheduler tick after time elapses	Auto-transitions to published	Pass
Delete another user's post (spoofed id)	Blocked by ownership check	Pass
GET analytics summary, several posts	Correct totals and breakdown	Pass

C. Discussion of Results

All twelve functional test cases passed against a locally running MySQL instance and the Spring Boot back end. JWT-based authentication correctly rejected requests carrying missing, invalid, or expired tokens, and correctly distinguished a genuine validation error (for example a duplicate email) from a network failure, after a

defect in the shared apiCall helper — which had been silently substituting fake success data for real backend error responses — was identified and fixed. The scheduled publisher was verified by creating a post a few seconds in the future and observing its status flip from scheduled to published on the very next sixty-second tick without any user interaction, and the analytics endpoint was verified by seeding representative post records directly in the database and confirming the returned totals matched the expected sums from Equation (1). The one area verified only at the level of "correct method invoked, correct content logged" rather than against a live third-party API is the set of four platform-publishing stubs, since production developer credentials for four separate platforms were outside the scope of this work.

VI. Challenges and Limitations

A. Technical Challenges

Each target platform exposes a differently shaped publishing API with its own authentication flow and scopes, which had to be reconciled into a single internal Post representation. A pre-built, operating-system-specific node_modules folder shipped with the original project archive also broke the front-end build on a different platform and had to be removed in favour of a fresh install.

B. Scope Limitations

Live OAuth connections to Facebook, Instagram, Twitter/X, and LinkedIn, and the network calls that would actually publish to those platforms, are intentionally left as clearly marked stub methods rather than implemented, since registering developer applications with four separate platforms was outside an academic project timeline. The Accounts page similarly simulates connecting a platform locally, and subscription upgrades update only the plan field on the user record rather than triggering a real payment transaction.

C. Integration Defects Resolved

Two further defects were found and fixed during integration: several front-end pages had been wired to hardcoded fake arrays instead of the working back-end endpoints, and were rewired to call the real API through the shared apiCall helper; and that same helper had been catching every error indiscriminately and substituting fake demo data even for genuine 400-level validation errors, which is now restricted to trigger only on an actual network failure.

VII. Future Scope

The following extensions are identified as natural next steps, in rough order of priority:

- Real platform integrations: implement the four publishing stubs using each platform's official SDK/REST API, including the OAuth 2.0 authorisation-code flow needed for the Accounts page.
- Real payment integration: connect the existing plan-update flow to a gateway such as Stripe or Razorpay so upgrades are billed automatically.
- A proper media-upload pipeline with object storage and per-platform image resizing, in place of a plain image-URL field.
- A background job that periodically pulls real engagement figures from each platform's own insights API, replacing the currently seeded analytics values.
- Team accounts with a draft → review → approved → scheduled approval workflow for the Enterprise tier, notifications for publish outcomes and plan limits, a clustered job queue for horizontal scalability, and an automated JUnit/React-Testing-Library regression suite.

VIII. Conclusion

This work set out to design and build a unified platform that lets a user compose, publish, and schedule social-media content across multiple platforms from a single authenticated dashboard, and view a consolidated analytics summary, rather than repeating the same work separately in each platform's own tools. This was achieved with a React front end, a Spring Boot REST back end secured with JWT and BCrypt, and MySQL as a relational store, organised into a layered architecture that keeps authentication, post management, scheduling, and analytics as independent, testable modules. The resulting system delivers hashed-password registration and signed-JWT login; a compose flow that publishes immediately or schedules for later; a background scheduler that reliably auto-publishes due posts even when no user is online; an aggregated analytics endpoint; and a tiered Free/Pro/Enterprise subscription model, validated by twelve passing functional test cases. The four real third-party publishing integrations were a deliberate scoping decision so that the rest of the architecture could be demonstrated and tested completely without depending on external developer-account approvals from four different companies, and they remain the primary item of future work.

References

[1] Spring Team, Spring Boot Reference Documentation, version 3.2, 2024. [Online]. Available: https://docs.spring.io/spring-boot/docs/3.2.x/reference/html/

[2] Spring Team, Spring Security Reference Documentation, 2024. [Online]. Available: https://docs.spring.io/spring-security/reference/

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

[4] Oracle, MySQL 8.0 Reference Manual, MySQL Documentation. [Online]. Available: https://dev.mysql.com/doc/

[5] Meta Platforms Inc., Facebook Graph API Reference, 2024. [Online]. Available: https://developers.facebook.com/docs/graph-api/

[6] Meta Platforms Inc., Instagram Platform — Content Publishing API, 2024. [Online]. Available: https://developers.facebook.com/docs/instagram-api/guides/content-publishing

[7] X Corp., Twitter API v2 Documentation, 2024. [Online]. Available: https://developer.twitter.com/en/docs/twitter-api

[8] LinkedIn Corp., Share on LinkedIn — UGC Posts API, 2024. [Online]. Available: https://learn.microsoft.com/en-us/linkedin/marketing/community-management/shares/ugc-post-api

[9] Vite Team, Vite — Next Generation Frontend Tooling, Documentation, 2024. [Online]. Available: https://vitejs.dev/

[10] React Team, React Documentation, 2024. [Online]. Available: https://react.dev/