

MediaMind:

Building a Content-Driven Recommendation Engine Across Anime, Movies, Books, and Web Series

Kunal Patil , Harshal Karlake, Harshad Shirnath, Kushal Shegokar, Prof. Anita P. Jaware

Department of Computer Science and Engineering
College of Engineering and Technology, Akola
Sant Gadge Baba Amravati University, Amravati, Maharashtra, India

Abstract

People who consume a lot of digital media share one frustration that the platforms themselves have arguably made worse: the discovery problem. You know the kind of thing you want to watch or read next, but the tools for finding it are either too blunt or built around assumptions that fall apart when you are a new user with no watch history. MediaMind grew out of that frustration. It is a full-stack web application we built to let people discover anime, movies, books, and web series using an engine that works from the content of the media itself, not from what other users clicked on. At its core is a weighted Jaccard similarity function that scores every item in the catalogue against a chosen title across three dimensions — genre overlap (0.50 weight), thematic tag intersection (0.35), and media-type affinity (0.15) — and returns the five closest matches. Surrounding that engine is a complete application: JWT authentication, personal collections, watch history, a four-label review system (Skip / Timepass / Go for it / Perfection), real-time keyword search with session history, an admin panel for catalogue management, and weekly digest emails delivered through Nodemailer on a cron schedule. We tested the finished system with fourteen API cases, all of which passed, and five user evaluators who completed a structured task flow at a 95% rate, scoring the experience 4.2 out of 5 on average. The recommendation computation itself takes under 5 milliseconds across a 200-item catalogue, which

matters because it runs synchronously inside the request cycle.

Keywords — Content-Based Recommendation, Jaccard Similarity, Cold-Start Problem, Cross-Domain Discovery, Node.js, Express, MySQL, JWT Authentication, Full-Stack Web Application, Media Discovery.

I. INTRODUCTION

Recommendation engines are supposed to reduce effort. In practice, many of them shift the effort without reducing it — instead of scrolling through a catalogue manually, you scroll through a "For You" feed that feels equally random. Part of the reason for this is structural: most consumer recommendation systems are built on collaborative filtering, a technique that derives suggestions by identifying users with similar histories and surfacing what those users consumed next. When the history is rich and the user base is large, this works. When a user is new, it does not — and that failure mode has been documented in the field since at least 1994 [1].

A second limitation is less frequently discussed: nearly every recommendation platform operates within a single domain. Anime platforms recommend anime. Book platforms recommend books. The idea that a novel might be the right follow-up to an anime someone just finished watching — because both share an unusual tonal texture, the same genre tags, a

similar thematic preoccupation — is not something any mainstream tool acts on. The domains are kept separate by the structure of the businesses, even when the content itself does not respect those separations.

MediaMind addresses both problems simultaneously. We use content-based filtering, which means recommendations derive from the attributes of items themselves: the genres listed on the item, the tags attached to it, its media type. No prior user behaviour is required. A user who registers, opens a single anime title, and asks for recommendations immediately gets five results grounded in the content similarity of that title to the rest of the catalogue — including, if the match is strong enough, a book or a film. This paper documents how that system was

Recommendation system research has two dominant traditions.

The collaborative filtering tradition, which began with the GroupLens system of Resnick et al. [1], treats the user community as the primary signal. If a set of users who rated similar items highly also rated a particular item highly, that item is a candidate recommendation for users in that cluster. Breese, Heckerman, and Kadie [2] formalised the evaluation methodology for these approaches in 1998 and established the benchmarks the field still references today. Both memory-based and model-based variants assume that enough user interaction data exists to learn from — an assumption that holds for mature platforms and breaks for new ones.

The content-based tradition, documented comprehensively by Pazzani and Billsus in their 2007 survey [3], takes the opposite stance: rather than asking who liked this item, it asks what this item is like. Items are described as feature vectors and similarity is computed directly between those vectors. The approach has a known limitation — it tends toward over-specialisation, recommending variations on what a user already knows — but its cold-start resistance is a structural advantage that collaborative methods cannot replicate. Burke [4] argued that the right architecture

designed, how it was built, and what happened when we tested it.

The rest of this paper is organised as follows. Section II reviews existing literature and positions this work relative to it. Section III covers the system architecture and database design. Section IV explains the recommendation algorithm in detail. Sections V and VI describe the backend API and frontend respectively. Section VII presents testing results. Sections VIII through X address comparison, limitations, and conclusions.

II. BACKGROUND AND RELATED WORK

A. The Two Main Approaches

combines both, and Netflix's prize competition [5] produced the matrix factorisation methods that now power most large-scale hybrid implementations.

B. Domain-Specific Platforms and Their Gaps

MyAnimeList, AniList, Goodreads, and IMDb each serve a single domain well. All four use forms of collaborative or aggregate-rating approaches. None support cross-domain recommendations, and all four require user history before recommendations become meaningful. These are not criticisms of engineering choices that made sense for their respective scales; they are observations about which needs those platforms were not designed to meet. The user who finishes a science fiction anime and wants to know whether there is a novel that would satisfy the same craving has no tool that serves them directly.

C. The Gap This Work Fills

We did not find an open-source, full-stack, content-based recommendation system spanning more than one media domain with a functional deployment. Most academic recommendation projects are either domain-specific, collaborative, or proof-of-concept implementations that stop at the algorithm and do not build the surrounding application. This project contributes a complete working system — algorithm, API, database, frontend, authentication, email

notifications, and admin tooling — that is deployable without modification.

III. SYSTEM ARCHITECTURE

A. Three-Tier Design

The application follows a conventional three-tier structure. The presentation tier is five static HTML pages rendered in the browser, with all dynamic behaviour handled through ES6 JavaScript and the Fetch API — no framework dependencies on the frontend. The application tier is a Node.js 18 server running Express 5, listening on port 3001. It handles routing, middleware, business logic, and the recommendation computation. The data tier is a MySQL 8 relational database paired with an in-memory JSON catalogue file.

The choice to split media catalogue storage from user data storage reflects a practical difference in access patterns. The catalogue — titles, genres, tags, descriptions — is queried on almost every request and rarely changes. Loading it from a JSON file into process memory at server startup means those reads never touch the database. User data — collections, watch records, reviews — changes frequently and must be consistent; it lives in MySQL with foreign key constraints and unique indexes that enforce integrity.

Server initialisation runs an async function that creates a MySQL connection pool (limit: 10 connections), executes CREATE TABLE IF NOT EXISTS statements for all six tables, and verifies that a default admin account exists — creating one if not. This makes the application self-bootstrapping on a clean database, with no separate migration step required.

B. Database Schema

The mediaind database uses the utf8mb4 character set with the unicode_ci collation. This choice matters in a catalogue that includes Japanese-language anime titles and book names with accented Latin characters. Six tables cover the complete data model, summarised in Table I.

Table Name	Contents and Constraints
users	Credentials, hashed password, role enum (user/admin), theme, email verification token, last digest timestamp
collections	Maps user_id to media_id for saved items; composite unique key on (user_id, media_id)
watched	Maps user_id to media_id for watched items; composite unique key on (user_id, media_id)
reviews	Stores vote as TINYINT 0-3 and free-text review_text per user-media pair
search_logs	Logs every search query string with optional user_id and created_at timestamp
email_logs	Audit record of every outgoing email: type, user_id, sent_at

TABLE I: Database Schema Summary

C. Authentication and Access Control

Passwords are processed through bcrypt at ten salt rounds before storage. No plaintext password is ever written to any table, log, or response body. On successful registration or login the server issues a signed JWT with a 30-day expiry window.

Protected routes verify the Bearer token through authMiddleware before any handler logic runs. Admin-only routes add a second layer: adminMiddleware queries the users table to confirm the identity holds role = "admin" and returns HTTP 403 if it does not. All SQL queries use the mysql2 prepared statement interface throughout; there is no dynamic string concatenation in any query path.

IV. RECOMMENDATION ALGORITHM

A. Similarity Function

The engine is invoked at GET /api/recommend/:id. Given a target item T, it iterates through every other entry C in the catalogue and computes:

$$S(T,C) = 0.50 \times J_{TypeMatch\ genre} + 0.35 \times J_{tags} + 0.15 \times J_{tags}$$

The Jaccard coefficient $J(X,Y) = |X \cap Y| / |X \cup Y|$ measures set overlap as a fraction of the union. TypeMatch is a binary variable — 1 when T and C share the same media type, 0 otherwise. Candidates where $S = 0$ (no genre or tag intersection) are excluded before sorting. The five highest-scoring remaining items are returned as the recommendation set.

B. Weight Rationale

Genre receives the highest weight — 0.50 — for a reason grounded in how recommendation preference actually works. Whether someone who enjoys horror will enjoy a title tagged only as romance does not depend on any amount of tag or type overlap; the genre boundary is the strongest separator of audience expectation. This aligns with the finding in Pazzani and Billsus [3] that genre-level features carry the strongest single signal in content-based filtering scenarios.

Tag overlap at 0.35 captures texture that genre alone flattens. Action is a genre, but "slow-burn action" and "tournament arc action" are different enough that recommending across that divide frustrates users. Tags carry that distinction. The TypeMatch weight of 0.15 is kept deliberately low: the system should lean toward same-type results when similarity is otherwise equal, but it should not suppress a book with genuinely high genre-tag overlap in favour of an anime that only shares the type.

C. Computational Properties

Time complexity is $O(n)$ per request. On a 200-item catalogue the operation completes in 4 ms on average. Extrapolating linearly, a 5,000-item catalogue would take roughly 100 ms — within a synchronous response window. Growth beyond that scale would justify pre-computing a similarity matrix at index time or using an approximate nearest-neighbour index. Neither is necessary within the scope of this project.

V. BACKEND API

A. Endpoints

Ten REST endpoints cover the full application surface. Table II lists them with methods and authentication requirements.

Endpoint	Method / Auth	Purpose
/api/register	POST / Open	Create user account; return signed JWT
/api/login	POST / Open	Verify credentials; return signed JWT
/api/media	GET / Optional	Serve full catalogue; ?q= enables keyword filtering
/api/recommend/:id	GET / Optional	Return top-5 Jaccard-scored recommendations
/api/collection	GET/POST/DELETE / JWT	Read, save, or remove personal collection entries
/api/watched	GET/POST/DELETE / JWT	Read, add, or remove watch-history entries
/api/review	POST / JWT	Create or overwrite a review for a media item
/api/reviews/:mediaId	GET / Open	Retrieve community reviews for a specific item
/api/admin/stats	GET / Admin	Return platform analytics: counts, top searches
/api/admin/media	POST/PUT/DELETE / Admin	Full CRUD over the media catalogue JSON

TABLE II: REST API Endpoint Reference

B. Registration and Login Detail

POST /api/register requires a username, an email address, and a password of at least six characters. After uniqueness checks on both username and email, the password is hashed through bcrypt, the record is inserted, and a 30-day JWT is signed and returned alongside the user object. The endpoint simultaneously queues a welcome email through Nodemailer and writes an entry to email_logs. POST /api/login retrieves the user record by email, calls bcrypt.compare() against the stored hash, and issues a fresh JWT on a correct match. Error responses for both endpoints are deliberately non-specific — "invalid credentials" rather than "email not found" — to prevent enumeration.

C. Scheduled Email Digest

A node-cron job running the expression "0 8 * * 1" fires at 8:00 AM every Monday. It queries for users whose last_digest timestamp is more than seven days in the past, then builds a personalised HTML email for each: up to five media items with a community average review score of 2 or higher that do not appear in the recipient's collection or watch history. Successful dispatch updates the last_digest field and appends a row to email_logs. The whole process is self-contained in the scheduled job; no user action triggers it.

VI. FRONTEND IMPLEMENTATION

A. Page Inventory and Navigation

Five HTML5 pages share a single stylesheet and a single JavaScript file. The landing page serves login and registration forms. The index page — the entry point for authenticated users — holds the hero search bar, a trending media strip populated from server-aggregated search_logs data, and the main explore grid with an inline modal for item detail views. The explore page adds a sidebar filter panel supporting multi-genre and media-type selection simultaneously. The recommend page offers two modes: genre-tag selection and title input. Navigation elements across all pages are injected by script.js at load time based on localStorage token state; the Admin Dashboard link appears only when the decoded JWT carries role = admin.

B. Search Behaviour and Query History

Submitting a search dispatches a GET request to /api/media with the query string as a URL parameter, replacing the current grid with filtered results. The backend simultaneously logs the query to search_logs. On the client side, the last three distinct queries are written to localStorage under the key mm_search_history and rendered as dismissible chip elements below the input field. A trash icon clears the full history. The trending searches block on the home page is not a manually curated list; it reflects the most frequent recent queries from the database, updated on each page load.

C. Theming and Layout

Dark and light mode is handled entirely through CSS custom properties on the root element. The active theme is stored in the users table and applied from the server response; guest users default to dark. The media grid uses CSS Grid with auto-fill columns and a 180 px minimum width, which means the layout transitions from four columns on a wide desktop to a single column on a narrow phone without any explicit media query breakpoints. Modal overlays use position:fixed with backdrop-filter:blur() for the layered depth effect behind the detail panel.

VII. TESTING AND EVALUATION

A. API Test Results

Fourteen test cases were prepared and executed in Postman, covering each endpoint with both expected-valid and expected-failure inputs. Table III shows the complete list. All fourteen passed on the first run. The cases that we considered most important were TC-9 (invalid media ID returning 404 rather than a 500), TC-11 (missing JWT returning 401 rather than leaking handler output), and TC-14 (non-admin JWT being blocked at the admin stats endpoint rather than falling through). All three behaved as specified.

T C	Endpoint	Input Condition	Outcome
--------	----------	--------------------	---------

1	POST /api/register	Well-formed new account	201 + JWT — Pass
2	POST /api/register	Email already registered	400 error — Pass
3	POST /api/register	Password length below minimum	400 validation — Pass
4	POST /api/login	Correct email and password	200 + JWT — Pass
5	POST /api/login	Incorrect password	401 — Pass
6	GET /api/media	No query parameter supplied	200 full catalogue — Pass
7	GET /api/media?q=dark	Keyword search: "dark"	200 filtered results — Pass
8	GET /api/recommend/1	ID present in catalogue	200 + 5 results — Pass
9	GET /api/recommend/9999	ID absent from catalogue	404 not found — Pass
10	POST /api/collection	Valid JWT and existing mediaId	200 saved — Pass
11	POST /api/collection	No Authorization header	401 no token — Pass
12	POST /api/review	vote = 2, non-empty review text	200 saved — Pass
13	GET /api/admin/stats	JWT with role = admin	200 stats — Pass

14	GET /api/admin/stats	JWT with role = user	403 forbidden — Pass
----	----------------------	----------------------	----------------------

TABLE III: API Test Cases and Results

B. Performance Measurements

Response times were recorded across 30 requests per operation on a local development machine. Every operation stayed below 80 ms. The recommendation engine at 4 ms average is the standout result — it runs fully in memory with no I/O, so variance is also very low (max 8 ms). JWT verification adds less than 1 ms per request. The catalogue fetch at 42 ms is the slowest routine operation; this is partly the cost of serialising the full JSON array, which server-side pagination would reduce substantially. Table IV shows the numbers.

Operation	Avg (ms)	Max (ms)
Full media catalogue fetch	42	78
Keyword search (/api/media?q=)	38	65
Recommendation engine (200 items)	4	8
JWT middleware verification	< 1	2
Collection add or remove	18	35
Review record submission	22	40

TABLE IV: API Response Time Summary

C. User Acceptance Testing

Five student evaluators each completed four tasks in sequence without any guidance from the development team. The tasks were chosen to cover the primary user journeys: account creation with a genre search, item interaction (save and watch), review submission, and recommendation use. Results are in Table V.

The only incomplete case was one evaluator on Task 3, who paused on the review rating labels before asking what "Timepass" meant. This is a labelling clarity issue rather than a functional one, and the fix is simple: add tooltip text to each label. All evaluators described the interface as intuitive on a five-point Likert scale.

Mean satisfaction across the full session was 4.2 out of 5.

Task	Scenario Description	Completed	Mean Time
T1	Register new account; run genre-based search	100%	68 sec
T2	Open item; add to collection; mark as watched	100%	45 sec
T3	Navigate to an item; submit four-tier review	80%	52 sec
T4	Request recommendations; open a result's details	100%	39 sec
—	Combined flow (T1 through T4)	95%	134 sec

TABLE V: User Acceptance Testing Results

VIII. COMPARATIVE ANALYSIS

Table VI maps MediaMind against five established platforms on the dimensions most relevant to this project's scope.

Platform	Technique	Cold Start	Multi-Domain	Open
Netflix	Hybrid (CF + ML)	Poor	No	No
MyAnimeList	Collaborative	Poor	Anime only	No
Goodreads	Collaborative	Poor	Books only	No
IMDb	Aggregate ratings	N/A	Partial	No
AniList	Collaborative	Poor	Anime only	No
MediaMind	Content-Based Jaccard	Good	4 Domains	Yes

TABLE VI: Platform Comparison

A Netflix-scale collaborative system with hundreds of millions of users will always outperform a content-based engine for a long-standing user with rich

interaction history. That is not the competition. The comparison that matters is for the user in their first session, or the user who wants a cross-domain suggestion, or the researcher who wants an open system they can inspect and modify. In those scenarios, MediaMind is the only option in this table that delivers a meaningful result.

IX. LIMITATIONS

The static JSON catalogue is the most significant structural limitation. It requires manual effort to keep current, and it will not scale beyond a few thousand entries before loading time becomes noticeable. The intended fix is integration with external metadata APIs — OMDb for films, AniList for anime, Google Books for literature — which the modular backend is already designed to accommodate without restructuring.

The algorithm has no feedback loop. A user who consistently marks horror items as "Skip" still receives horror suggestions whenever they click on a mystery title with horror co-tags. Incorporating review signal into the scoring — penalising item types or genres that a user's own review history rejects — would address this and would represent the first step toward a hybrid content-plus-behavioural model.

Two further issues are worth flagging for a production deployment. Server-side pagination is absent; the full catalogue is serialised on every grid load, which will degrade as item counts grow. And the CORS policy allows all origins in the current configuration — appropriate in development but not in any environment exposed to the public internet.

X. CONCLUSION

MediaMind demonstrates that content-based filtering can be built into a fully functional, multi-domain discovery platform without requiring a large user base, proprietary training data, or any machine learning infrastructure. The weighted Jaccard similarity engine is intentionally simple — three features, three weights, one formula — but the results it produces are semantically meaningful. Items sharing high genre and

tag overlap with a target consistently ranked at the top of the recommendation list in testing. Cross-domain results appeared in the right circumstances: when a book and an anime genuinely shared enough content features, the book surfaced.

The 95% UAT completion rate and 4.2/5 satisfaction score are meaningful numbers because they came from evaluators with no prior exposure to the system. The single failure — one evaluator hesitating over the "Timepass" label — is the kind of small friction point you cannot anticipate in design but catch immediately in user testing, which is exactly why that testing phase exists.

The work that would make the most difference in a future version is clear from the limitations: external API integration to automate catalogue growth, a behavioural feedback term in the similarity function, and server-side pagination. The architecture supports all three additions without requiring significant restructuring. Building them out is the next step.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering, College of Engineering and Technology, Akola, and Sant Gadge Baba Amravati University for the resources and institutional environment that supported this project throughout the 2025-26 academic year. Gratitude is also owed to Dr. S. L.

Satarkar (Head of Department, CSE) and Dr. P. V. Thorat (Principal) for their continued encouragement. The open-source libraries this system depends on — Express, mysql2, bcrypt, jsonwebtoken, Nodemailer —

[8] Express.js Contributors, *Express 5.x API Reference*. [Online]. Available: <https://expressjs.com/en/5x/api.html>. [Accessed: Mar. 2025].

[9] Nodemailer Contributors, *Nodemailer — Node.js Mailer Documentation*. [Online]. Available: <https://nodemailer.com/about/>. [Accessed: Apr. 2025].

represent collective engineering work without which a project of this scope could not have been completed within the available timeframe.

REFERENCES

- [1] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "GroupLens: An open architecture for collaborative filtering of netnews," in *Proc. ACM Conf. on Computer-Supported Cooperative Work (CSCW)*, Chapel Hill, NC, 1994, pp. 175–186.
- [2] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proc. 14th Conf. on Uncertainty in Artificial Intelligence (UAI)*, Madison, WI, 1998, pp. 43–52.
- [3] M. J. Pazzani and D. Billsus, "Content-based recommendations systems," in P. Brusilovsky, A. Kobsa, and W. Nejdl (Eds.), *The Adaptive Web: Methods and Strategies of Web Personalization*, LNCS vol. 4321, Springer, 2007, pp. 325–341.
- [4] R. Burke, "Hybrid recommender systems: Survey and experiments," *User Modeling and User-Adapted Interaction*, vol. 12, no. 4, pp. 331–370, Nov. 2002.
- [5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, Aug. 2009.
- [6] OpenJS Foundation, *Node.js v18 LTS API Documentation*. [Online]. Available: <https://nodejs.org/docs/latest-v18.x/api/>. [Accessed: Apr. 2025].
- [7] Oracle Corp., *MySQL 8.0 Reference Manual*. [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/>. [Accessed: Apr. 2025].
- [10] Auth0, "Introduction to JSON Web Tokens." [Online]. Available: <https://jwt.io/introduction>. [Accessed: Mar. 2025].