E-Commerce Portal with Intelligent Recommendation System

Krushna Peherkar[1], Tejas Mahajan[2], Prathmesh Tarte[3], Prof.S.R. Yendole[4]
[1,2,3,4]*International Centre of Excellence in Engineering and Management, Chhatrapati Sambhajinagar, Maharashtra, India – 431136*

***Abstract*—Online retailers must help shoppers find relevant products among very large catalogues while keeping the storefront fast, secure and easy to administer. This paper presents the design and implementation of an E-Commerce Portal with an Intelligent Recommendation System built on the MERN stack (MongoDB, Express.js, React.js and Node.js). The portal provides user registration and login using password hashing and JSON Web Tokens (JWT), a searchable and filterable product catalogue, cart and checkout, inventory control, order management and a role-based administration panel. An interaction-tracking module records views, clicks, ratings, cart additions and purchases in MongoDB, and a recommendation service applies collaborative filtering with relevance scoring and ranking to produce personalised product suggestions, while popular and category-based fallbacks serve new users. A Redis caching layer with time-to-live (TTL) policies and cache invalidation reduces database workload and response time. The system was tested with Postman across authentication, shopping, caching and recommendation scenarios and prepared for deployment on hosted services. The work demonstrates a scalable, modular approach for combining core commerce functions with data-driven personalisation.**

***Index Terms*—E-commerce, recommendation system, collaborative filtering, MERN stack, Redis caching, JWT authentication**

I. INTRODUCTION

Electronic commerce has become a primary channel for buying and selling goods. As catalogues grow into thousands of items, shoppers face information overload: finding a suitable product through search and category browsing alone takes time, and many visitors leave without buying. Recommender systems address this problem by predicting which items a particular user is likely to value, and they are now a standard part of large retail platforms [3], [7]. For a business, relevant recommendations raise engagement and conversion; for a customer, they shorten the path from intent to purchase.

Building a complete portal, however, involves more than a recommendation algorithm. A shopper needs secure accounts, a responsive storefront, reliable cart and checkout flows and accurate stock information, while the business needs tools to manage products, categories, inventory and orders. Many academic and prototype portals implement only the basic cart and omit personalisation, caching or administration, which limits their value as realistic, scalable systems.

This paper describes an E-Commerce Portal with Intelligent Recommendation System developed on the MERN stack. The aim is to deliver shopping, administration, analytics and personalised recommendation capabilities in one integrated and scalable portal that serves diverse business and customer needs. The main objectives are: (i) to build a responsive React.js storefront; (ii) to expose commerce functions through a structured Node.js/Express.js REST API backed by MongoDB; (iii) to secure access using hashed passwords, JWT and role-based authorisation; (iv) to capture user behaviour and generate collaborative-filtering recommendations; and (v) to improve responsiveness with Redis caching.

The contributions of this work are a layered reference architecture for an intelligent e-commerce portal, a behaviour-driven recommendation pipeline with a feedback loop and cold-start handling, a cache strategy that serves popular data and recommendation results quickly, and a systematic testing and deployment approach. The remainder of the paper is organised as follows: Section II reviews related work, Section III states the problem and requirements, Section IV gives the system overview and technology stack, Section V presents the architecture, Sections VI to VIII describe

implementation of the commerce platform, recommendation engine and caching layer, Section IX covers testing and deployment, Section X discusses outcomes and limitations, and Section XI concludes.

II. RELATED WORK

Collaborative filtering. Collaborative filtering (CF) predicts a user's interest from the preferences of similar users or the similarity between items. The GroupLens system was an early open architecture that used user ratings to recommend netnews articles [1]. Item-based algorithms, which compare items rather than users, were shown to scale better and to give high-quality predictions [2], and the item-to-item approach was applied in production at Amazon.com [3]. Surveys classify CF techniques into memory-based and model-based methods and discuss sparsity, scalability and cold start as the principal challenges [5], [6].

Model-based and implicit feedback methods. Matrix factorisation maps users and items to latent factors and was prominent in large-scale rating prediction [4]. In real shops, explicit ratings are scarce, whereas clicks, views, cart events and purchases are abundant; CF for implicit feedback datasets treats these events as signals of preference with different confidence levels [10]. The present portal follows this view and records all such interactions.

Hybrid and content-based approaches.

Content-based methods recommend items similar in attributes to those a user liked, and hybrid systems combine content and collaborative evidence to reduce cold-start problems [5], [7]. Our system keeps CF as its core and uses popularity and category-based fallbacks for new users.

Web architecture, security and caching. Representational State Transfer (REST) provides a stateless client-server style suited to web APIs [9]. JSON Web Tokens give a compact, signed way to carry claims between client and server and enable stateless authentication [8]. In-memory stores such as Redis are widely used to cache frequently requested data to reduce latency and database load. This work integrates these ideas with a MongoDB-backed MERN application.

III. PROBLEM STATEMENT AND REQUIREMENTS

Existing small-scale portals typically suffer from four gaps: (a) product discovery depends on manual search and category browsing, (b) behavioural data is not captured, so nothing can be learned from customers, (c) every request goes to the database, so latency grows with traffic, and (d) administration is limited or absent. The proposed portal is therefore specified against the following requirements.

Table I. Functional and Non-Functional Requirements

ID	Requirement	Type
R1	Secure registration, login and role-based access	Functional
R2	Catalogue with search, sorting and filters	Functional
R3	Cart, checkout, inventory control and order tracking	Functional
R4	Admin management of products, categories, stock, orders	Functional
R5	Capture of views, clicks, ratings, cart and purchase events	Functional
R6	Personalised recommendations with new-user fallback	Functional
R7	Fast responses under repeated, high-demand requests	Non-functional
R8	Protection of credentials, secrets and sensitive data	Non-functional
R9	Modular, maintainable and deployable code base	Non-functional

Two principal actors interact with the system. The customer registers, browses, searches, adds items to the cart, checks out, tracks orders and receives recommendations. The administrator manages the catalogue, inventory and order fulfilment. The recommendation service acts as an internal actor that reads interaction data and writes ranked results to the cache.

IV. SYSTEM OVERVIEW AND TECHNOLOGY STACK

The project vision can be summarised by three questions. Why: to create scalable, intelligent product discovery that connects customers with relevant

experiences while empowering teams with trusted data. How: by combining MongoDB, Express.js, React.js and Node.js with collaborative filtering, Redis caching and JWT for secure, responsive performance. What: a scalable, integrated portal delivering shopping, administration, analytics and personalised recommendations.

Table II. Technology Stack and Role of Each Component

Layer	Technology	Role in the portal
Frontend	React.js (Vite), HTML, CSS, JavaScript	Responsive pages, routing, reusable components
Backend	Node.js, Express.js	REST APIs, controllers, middleware, services
Database	MongoDB	Users, products, carts, orders, reviews, interactions
Cache	Redis	Popular items, recommendation results, TTL policies
Security	JWT, one-way password hashing	Authentication and role-based authorisation
Tooling	VS Code, Git/GitHub, Postman	Development, collaboration, API testing

MongoDB was chosen because product catalogues and interaction logs are document-shaped and evolve over time; Express.js keeps the API layer thin and modular; React.js supports component reuse and fast client-side navigation; and Redis offers sub-millisecond in-memory reads for hot data.

Development methodology. The project was carried out in staged increments, each producing a working result that the next stage builds on: (1) learning the foundation technologies; (2) setting up the development, database, caching and project-structure environment; (3) building the React frontend; (4) creating the Node.js/Express.js backend and REST APIs; (5) implementing authentication; (6) developing product management and shopping features; (7) tracking user activity and building recommendations; (8) integrating Redis; (9) building the administration panel; (10) testing; (11) deployment; and (12) documentation with UML diagrams, test evidence and presentation material. This ordering keeps the platform usable at every stage and allows the intelligent features to be added without disturbing the core commerce flows.

V. SYSTEM ARCHITECTURE

The portal follows a layered client-server architecture (Fig. 1). The presentation layer is a React single-page application. It communicates over HTTPS with an application layer built with Node.js and Express.js, which exposes REST endpoints and groups logic into authentication, catalogue-cart-order and recommendation services. The data layer combines MongoDB as the system of record with Redis as a fast cache that sits in front of expensive reads.

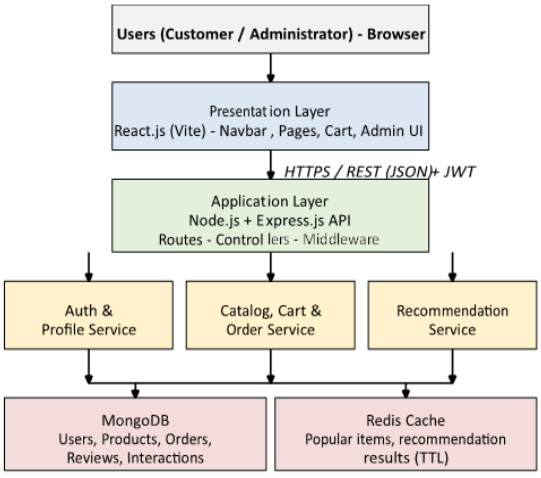


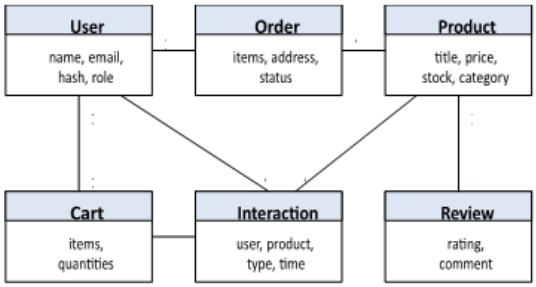
Fig. 1. Layered architecture of the E-Commerce Portal

The code base is separated into frontend, backend and configuration directories to clarify responsibilities, simplify maintenance and give a scalable foundation for new features. In the backend, routes, controllers, middleware and services are kept apart so that each can be tested and changed independently. Table III lists the principal MongoDB collections and the information they hold.

Table III. Principal Data Collections

Collection	Principal information stored
Users	Name, email, hashed password, role (customer/admin), profile and preference signals
Products	Title, description, category, price, images, stock quantity, ratings

Carts / Orders	Items, quantities, delivery and payment details, order status history
Reviews	Product reference, user, rating, comment
Interactions	User, product, event type (view, click, cart, purchase, rating), timestamp



1 = one, * = many

Fig. 2. Entity relationships among the principal collections

A user owns a cart, places orders that reference products, writes reviews and generates interaction records. Interaction records link users and products and are the only input the recommender needs, which keeps the recommendation logic independent of the order and cart schemas.

VI. IMPLEMENTATION OF THE CORE COMMERCE PLATFORM

A. Foundation and environment setup

Development began with the foundation technologies: HTML, CSS, JavaScript, Git and GitHub, followed by responsive interface design, asynchronous programming and APIs, and team practices such as branches, clear commits and pull requests. The environment comprises Node.js for runtime support, Visual Studio Code for editing and debugging, a local MongoDB service whose connectivity was validated before use, and a local Redis instance whose connection was verified to provide a dependable caching layer.

B. Frontend

The frontend was initialised as a React project with Vite to obtain a fast development environment, folder structure and baseline configuration. A navigation bar and client-side routing connect the Home, Products, Login and Register screens and leave room for future portal pages. Responsive design, reusable components and consistent styling provide a uniform experience across devices and make later feature development quicker.

Table IV lists the main screens and their purpose.

Table IV. Main User Interface Screens

Screen	Purpose and main components
Home	Navigation bar, featured and popular products, personalised recommendations
Products	Product cards, keyword search, sorting, category / price / rating filters
Product detail	Specifications, price, availability, reviews, related suggestions
Login / Register	Validated forms that call the authentication API
Cart / Checkout	Basket review, delivery and payment details, order summary
Orders	Order history and status tracking
Admin	Product, category, inventory and order management

C. Backend and REST APIs

The server environment is configured with Node.js and Express.js. The API is structured into organised routes, controllers, middleware and services. REST endpoints are provided for users, products, carts, orders and reviews, enabling consistent data exchange across all commerce workflows. Reliability is strengthened through input validation, centralised error handling, structured logging and secure configuration of secrets.

D. User authentication and authorisation

Registration validates submitted details, enforces required fields and prevents duplicate or malformed accounts. Passwords are hashed with a strong one-way algorithm before being stored in MongoDB, so credentials stay protected even if database records are exposed. At login, the submitted password is verified against the stored hash and a JWT access token is issued [8]; the token authenticates later requests, giving stateless sessions. Middleware protects private routes by checking the token and separates customer and administrator permissions so that each role can reach only approved resources and actions.

E. Product management and discovery

The product catalogue presents cards with images, current prices and availability. Keyword search matches names, descriptions, brands and related terms. Sorting controls and filters for category, price range,

rating and availability help shoppers narrow large collections, and detailed product pages describe features, specifications, pricing, reviews and purchasing options before checkout.

F. Shopping features

Cart management lets shoppers add, update, remove and review items. Checkout captures customer information, validates delivery and payment details and shows the purchase summary before the order is submitted. Inventory control synchronises stock levels, reserves items during purchase and blocks unavailable products, which reduces overselling. Order management supports placement, status tracking, notifications and an accessible order history.

G. Security measures

Security is treated as a cross-cutting concern. Passwords are never stored in plain text; they are hashed with a one-way algorithm before storage. Access tokens are signed and checked on every protected request [8], and administrative operations require an additional role check. Input validation at the API boundary rejects malformed or duplicate data early, and centralised error handling avoids leaking internal details. Configuration values such as database URLs, JWT secrets and API keys are kept in environment variables rather than in source code, with least-privilege access and periodic secret rotation, and CORS policies restrict which origins may call the API in production.

H. REST API design

Table V lists the main resource groups exposed by the Express.js API. Protected routes pass through authentication middleware; administrative routes pass through an additional role check.

Table V. Main Api Resource Groups

Resource	Typical operations	Access
/Users	Register, login, profile	Public / user
/Products	List, search, filter, view, create, update, delete	Public / admin
/Cart	Add, update, remove, view	User
/Orders	Place, track, history, update status	User / admin
/Reviews	Add rating, list reviews	User
/Interactions	Record view, click, cart, purchase	User
/Recommendations.	Get ranked items for a user	User

Every endpoint validates its input, returns consistent JSON and delegates failures to a centralised error handler, so clients receive predictable status codes and messages.

I. Administration panel

The admin panel allows authorised staff to create, update, delete and publish products; add categories and define hierarchies; monitor stock in real time and identify shortages; review incoming orders and update fulfilment status; and enforce role-based access so that only authorised users can manage products, orders, inventory, settings and sensitive operational data.

VII. INTELLIGENT RECOMMENDATION ENGINE

The recommendation subsystem turns observed behaviour into ranked product lists. Its stages are shown in Fig. 3 and described below.

Cold-start users receive popular / category-based fallbacks until enough data exists

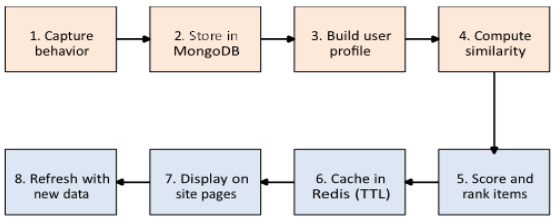


Fig. 3. Recommendation pipeline with feedback loop

A. Tracking user activity

Every product view, click, rating, cart addition and purchase is recorded and persisted in the Interaction collection of MongoDB. This gives the recommender a reliable, queryable history of customer intent signals. Observed behaviour is converted into preference signals that enrich the user profile, and newly captured interactions are fed back into later calculations so that profiles and rankings adapt as interests change.

B. Preference signal and similarity

Because explicit ratings are sparse, each user-item pair is given an implicit preference value that aggregates the number of events of each type (view, click, cart

addition, purchase, rating) multiplied by an event-specific weight, in line with the implicit-feedback view [10]. The weights are configurable and increase with the strength of the signal, so a purchase counts more than a view.

Similarity between two users is then computed with cosine similarity over the items both have interacted with [2]. Users whose preference vectors point in a similar direction are treated as neighbours, and the same idea can be applied to compare products. Comparing users and products in this way reveals shared preferences and similar entities and narrows the candidate set efficiently at scale.

C. Candidate scoring and ranking

For an unseen item, a collaborative score is predicted from the nearest neighbours of the user: the preferences of each neighbour for that item are averaged, weighted by how similar the neighbour is to the user and adjusted for each person's own rating tendency.

Relevance scores are then calculated by combining three ingredients: the collaborative score, the popularity of the item, and the match between the item's category and the user's inferred interests. Together with business rules, these are combined using tunable weights, and items are ranked so that the most useful one's surface first. Items already purchased or out of stock are filtered out before the top-K list is returned.

D. Recommendation algorithm

Algorithm 1 summarises the request-time procedure used to produce recommendations for a user.

Algorithm 1. Recommend (u, K)

1. key ← "rec:" + u; if cache has key, return cached list.
2. If u has fewer than m interactions, list ← top-K popular / category items; go to 8.
3. Load interactions of all users from MongoDB and build each user preference value from event counts and weights.
4. Compute cosine similarity between u and other users; keep the nearest neighbours.
5. For each candidate item i not yet purchased by u and in stock, predict the collaborative score from the neighbours.
6. Combine collaborative score, popularity and category match into a final score.
7. Sort by score and keep the top K.
8. Store list in Redis with a TTL; return list.

The complexity of steps 3 to 5 grows with the number of users and items, which is why the result of the procedure is cached (Section VIII) and why neighborhood size and candidate set size are limited in practice.

E. Delivery and cold start

Recommendations are displayed on the home, product and account pages. A new user has no behavioural history, so the system serves popular or category-based items until enough personal behaviour is available for learning. This fallback addresses the cold-start problem identified in the literature [5], [6].

VIII. REDIS CACHING STRATEGY

Recommendation calculation and catalogue queries are comparatively expensive, so a Redis layer is placed in front of MongoDB. The strategy follows the cache-aside pattern and has five elements: (1) popular products and recommendation results are stored in Redis for reuse; (2) incoming requests first check the cache, so cached responses are returned before MongoDB is queried or recommendations are recomputed; (3) tailored TTL policies give frequently changing information shorter lifetimes, balancing freshness with efficiency; (4) cached entries are invalidated and refreshed whenever products, inventory or recommendations change, preventing stale data from influencing purchase decisions; and (5) the cycle repeats continuously, lowering response time and database workload during high-demand periods.

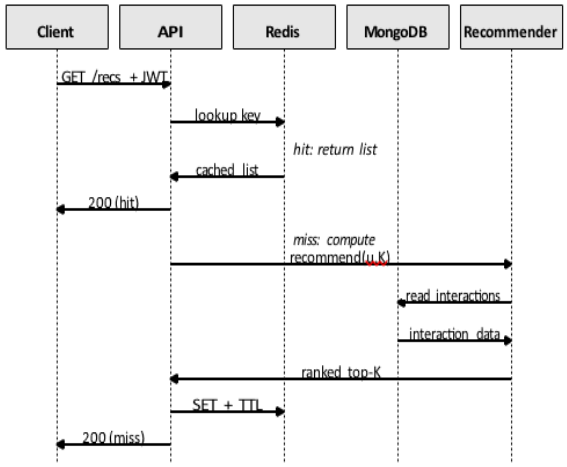


Fig. 4. Recommendation request: cache hit and miss

Table VI. Cache Policy by Data Category

Cached data	Lifetime (TTL)	Invalidated when
Popular products.	Short to medium	Product, price or stock changes
Recommendation lists	Short	New purchase or interaction by the user
Category listings	Medium	Category or catalogue edited by admin
Product details.	Medium	Product record updated

Lifetimes are expressed qualitatively because suitable values depend on traffic and how quickly the catalogue changes; they are configuration parameters of the cache service rather than fixed constants. When Redis is unavailable, the service falls back to reading MongoDB directly, so caching improves speed without becoming a single point of failure.

Scalability considerations

The architecture scales along three axes. The stateless API (JWT rather than server-side sessions) can be replicated behind a load balancer. MongoDB can be scaled with indexing on frequently queried fields and, when required, replication and sharding. The recommendation computation can be moved from request time to scheduled background jobs that pre-fill Redis, trading a little freshness for much lower latency.

IX. TESTING AND DEPLOYMENT

A. Application testing

Testing follows a four-stage cycle of planning, execution, analysis and optimisation. Test cases were defined to cover API behaviour, authentication flows, shopping journeys, recommendation outputs, validation rules, error paths and expected user outcomes. Postman was used to run API requests, authentication sequences, cart operations, checkout workflows, order creation, caching checks and recommendation scenarios. Results were reviewed to identify recurring failures, compare response times and refine assertions, after which defects were diagnosed and the tests repeated. Table VII summarises the coverage.

Table VII. Test Coverage

Area	What was verified
Authentication	Registration validation, hashed storage, login, JWT checks, role permissions
Shopping	Cart operations, checkout, order creation, stock reservation
Recommendation.	Output for existing and new users, use of fresh interactions
Caching	Cache hit/miss behaviour, TTL expiry, invalidation on change
Robustness	Input validation, error paths, response accuracy, workflow consistency

B. Deployment

The frontend is published to a production hosting platform with build settings and domain configured, and delivery of assets is verified. The Node.js/Express.js API is deployed with runtime settings and secure endpoints, and communication with the frontend and external services is confirmed. Hosted MongoDB and Redis services are configured with network access and production databases, and persistence, caching, availability and recovery are tested. Database URLs, JWT secrets and API keys are held as protected environment variables with least-privilege access and secret rotation. Finally, domains, CORS policies, logging, monitoring and health checks are validated before launch.

X. RESULTS AND DISCUSSION

The implemented portal delivers the complete shopping journey, from registration and login through search, filtering, cart, checkout and order tracking, together with an administrator console for catalogue, inventory and order control. Postman-based testing confirmed that protected routes accept only valid tokens, that customer and administrator roles are separated, that cart and order workflows remain consistent, and that cached and recomputed responses agree after invalidation.

Table VIII. Comparison With a Basic Portal

Aspect	Basic portal	Proposed portal
Discovery	Manual search and browsing	Search, filters, personalised lists
Behaviour data	Not captured	All key events stored

New users	Same static page	Popular / category fallback
Read performance	Every request hit DB	Redis cache-aside with TTL
Administration	Limited or absent	Role-based admin panel
Security	Often basic	Hashing, JWT, role checks

From an engineering viewpoint, three design choices were most significant. First, separating routes, controllers, middleware and services kept the codebase maintainable as features were added. Second, recording every interaction in a dedicated collection decoupled data capture from recommendation logic, so the algorithm can evolve without changing the storefront. Third, the Redis layer allows popular products and recommendation results to be served without repeated database queries, which is expected to matter most under heavy traffic.

Design trade-offs. Several choices balance competing goals. Neighborhood CF is simple and explainable but less scalable than factorization models [4]. Longer cache lifetimes lower latency but risk showing outdated stock or rankings, which is why invalidation on change is combined with TTL. Stateless JWT authentication simplifies scaling, but a token cannot be withdrawn before expiry without extra measures such as short lifetimes or a deny-list. Finally, strict inventory reservation prevents overselling at the cost of briefly holding stock for carts that are later abandoned.

Privacy and ethics. Behavioural tracking is sensitive. Only data needed for personalisation (product events and ratings) should be stored, access to interaction records should be limited to the recommendation service and administrators, and users should be informed about how their activity is used. Recommendation lists should also avoid over-promoting a narrow set of items, because pure popularity or similarity can reduce the diversity of what shoppers see.

Proposed evaluation protocol. To evaluate recommendation quality quantitatively, interactions can be split chronologically into training and test sets, with the latest purchases of each user held out. The personalised ranking is compared with a popularity-only baseline using the metrics in Table IX. System performance is measured separately, by recording response times for the same endpoints with and without the Redis layer under concurrent load.

Table IX. Metrics For Future Quantitative Evaluation

Metric	What it measures
Precision@K / Recall@K	Share of top-K items that are relevant / share of relevant items retrieved
NDCG@K	Ranking quality, rewarding relevant items placed higher
RMSE / MAE	Error of predicted preference values
Median and 95th-percentile latency	Endpoint response time with and without cache
Cache hit ratio	Fraction of requests served from Redis
Click-through, conversion rate	Online effect of recommendations on behaviour

Limitations. This paper concentrates on design and implementation. The evaluation was functional; a large-scale quantitative study using real traffic or public datasets was not performed. Neighbourhood-based CF also suffers from sparsity and its cost grows with the number of users and items, and the event and scoring weights were set by design judgement rather than learned. Future experiments should follow the protocol above and report the listed metrics.

XI. CONCLUSION AND FUTURE SCOPE

This paper presented an E-Commerce Portal with an Intelligent Recommendation System built on the MERN stack. The system integrates secure authentication, a searchable catalogue, cart, checkout, inventory and order management, an administration panel, behaviour tracking, collaborative-filtering recommendations and Redis caching in a modular architecture. The result is a scalable and maintainable foundation that connects customers with relevant products while giving business teams trusted operational data.

Future work includes hybrid recommenders that combine content-based features and matrix factorisation [4], learning the scoring weights from data, offline and online evaluation with A/B testing, integration of real payment gateways, explainable recommendations, privacy controls for behavioural data, and a mobile application client.

ACKNOWLEDGMENT

The authors thank the project guide and the department for their guidance and support during the development of this project.

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. Computer Supported Cooperative Work (CSCW), 1994, pp. 175–186.

[2] Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in Proc. 10th Int. World Wide Web Conf. (WWW), 2001, pp. 285–295.

[3] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," IEEE Internet Computing, vol. 7, no. 1, pp. 76–80, 2003.

[4] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," Computer, vol. 42, no. 8, pp. 30–37, 2009.

[5] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 6, pp. 734–749, 2005.

[6] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," Advances in Artificial Intelligence, vol. 2009, Art. no. 421425, 2009.

[7] Ricci, L. Rokach, and B. Shapira, Eds., Recommender Systems Handbook. New York, NY, USA: Springer, 2011.

[8] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015.

[9] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, CA, USA, 2000.

[10] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in Proc. 8th IEEE Int. Conf. Data Mining (ICDM), 2008, pp. 263–272.