Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Design and Development of a Curated E-Commerce Platform for Technology Products

Aviral Mehrotra
Bachelor of Technology Computer Science and Engineering
Shri Ramswaroop Memorial University
Lucknow, India

Dipanshu Pandey
Bachelor of Technology Computer Science and Engineering
Shri Ramswaroop Memorial University
Lucknow, India

Abstract **- The rapid spread of digital commerce has necessitated the development of web-based platforms that are scalable, secure, and maintainable simultaneously. This paper presents the design, development, and evaluation of Stannum, a comprehensive Full Stack E-Commerce application constructed using the MERN (MongoDB, Express.js, React.js, Node.js) technology stack. The primary objective of the project is to develop a production ready commercial platform that incorporates end-to-end functionality, spanning product catalogue management, user authentication, shopping cart operations, multi-address shipping management, and an integrated payment processing pipeline via the Razorpay API.**

The system architecture is predicated upon a clear separation of concerns, realised through a React.js Single Page Application frontend bootstrapped with Vite-React and styled with Tailwind CSS communicating with a stateless RESTful API backend built on Node.js and Express.js. Application state is managed through a combination of Redux Toolkit and React Context API, facilitating predictable and centralised state transitions across asynchronous data flows. MongoDB, accessed via the Mongoose Object Document Mapper, serves as the persistence layer, employing a document-referencing strategy to emulate relational integrity within a NoSQL paradigm.

Security is enforced through JSON Web Token (JWT) authentication stored in HTTPOnly cookies, Bcrypt-based password hashing, CORS configuration, and the segregation of sensitive credentials within server-side environment variables. Role-Based Access Control (RBAC) ensures that administrative privileges including product creation, order management, and sales analytics are isolated from customer operations. Media assets are managed through the Cloudinary API, and performance is optimised via Vite's code-splitting, lazy loading, and MongoDB indexing strategies.

The results of this project demonstrate that the MERN stack, when combined with third-party APIs and disciplined architectural patterns, creates a feasible and efficient model for the development of modern E-Commerce systems. The Stannum platform achieved a measurable reduction in frontend build times, seamless payment verification, and maintainable codebase modularity. Future enhancements may include microservices decomposition, AI-driven product recommendations, and mobile application porting via React Native.

Keywords **-** ***MERN Stack, E-Commerce, Single Page Application, RESTful API, Role-Based Access Control, JWT Authentication, Razorpay, Cloudinary, Redux Toolkit, MongoDB.***

I. INTRODUCTION

The global e-commerce sector has undergone a transformative evolution over the past two decades, transitioning from static HTML catalogues to highly dynamic, data-driven platforms capable of serving millions of concurrent users. According to prevailing industry analyses, global E-Commerce revenues continue to exhibit sustained double-digit growth, compelling software engineers and system architects to prioritise scalability, performance, and security at the earliest design stages. The increasing prevalence of mobile browsing and the rising expectations of end-users for instant, responsive interfaces have further mobilized the adoption of modern JavaScript based frameworks.

Traditional web application architectures, particularly those built upon the LAMP (Linux, Apache, MySQL, PHP) stack, while historically dominant, they have demonstrated many limitations in the context of real-time interactivity, non-blocking I/O operations, and rapid frontend iteration cycles. The emergence of the MERN stack consisting of MongoDB, Express.js, React.js, and Node.js has addressed many of these limitations by offering a unified, JavaScript-centric development environment that allows code reuse, asynchronous event-driven programming, and a document-oriented data model well-suited to the requirements of modern applications.

This paper presents Stannum, a full-stack e-commerce platform developed to investigate the practical viability of the MERN stack for the construction of a production-grade commercial application. The platform implements a complete commerce workflow, encompassing user registration and authentication, product browsing and filtering, cart management, checkout with payment gateway integration, and a dedicated administrative control panel. The primary objectives of the project are articulated as follows:

- To design and implement a modular, RESTful API backend capable of supporting role-based access control and separation of concerns.
- To develop a performant, component-based React frontend employing Redux Toolkit for predictable state management.
- To integrate third-party APIs specifically Razorpay for payment processing and Cloudinary for media management within a secure and maintainable architecture.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

- To evaluate the security posture of the system through the application of JWT-based authentication, Bcrypt hashing, and environment variable protection.

II. LITERATURE REVIEW

A. Evolution of Single Page Applications

The architectural model of the Single Page Applications was developed as a direct response to reduce latency and user experience limitations which were present in traditional multi-page web applications, where each navigational event required a full server-rendered page reload. Garrett came up with the concept of Asynchronous JavaScript and XML also known as AJAX, which provided the technical base for client-side page updates without full document refreshes [5]. The following development of Model View Controller JavaScript frameworks, mainly the AngularJS, enabled the systematic management of application state on the client side.

React.js was introduced by Facebook in 2013, it presented a major shift in UI development with the help of its virtual DOM diffing algorithm, component-based programming model. Unlike Full Stack MVC frameworks, React's design philosophy clearly restricts its scope to the view layer, providing the developers with the flexibility to create useful libraries for routing e.g. React Router, state management e.g. Redux, and data fetching e.g. Axios. This capability, while also carrying a degree of architectural decision making upon the developer, has proven highly advantageous for the development of large-scale applications such as Stannum, where the domain specific state requirements require fine-grained control.

B. Shift from Monolithic to Modular Architectures

The academic and industrial discussions on software architecture have increasingly favored the conversion of monolithic systems into modular or microservices-based designs, particularly within the context of scalable web applications. A monolithic architecture, in which all applications contain authentication, business logic, data access, and views are lumped within a single deployable unit, this presents challenges in the domains of independent scaling, fault isolation, and continuous deployment.

Our Stannum platform, while not being a full-fledged microservices system, adopts the architectural model of modular divisions within its backend by partitioning routing and controller logic across distinct domains like authentication (/auth), administrative operations (/admin), and user-facing operations (/shop). This design pattern aligns with the concept of a "modular monolith" as described by Martin Fowler; where the clear domain boundaries are established to enable a future migration to microservices if required.

C. MERN Stack Versus Traditional Technology Stacks

The LAMP stack (Linux, Apache, MySQL, PHP) was the dominant stack for web application development throughout the early 2000s, it offered a well-documented and a stable mix of technologies. However, this tack presented us with multiple issues when working on modern applications like the synchronous blocking I/O model of PHP is not well suited to manage high concurrency workloads, MySQL's rigid schema administration imposes overhead during iterative development; and the branching of server-side (PHP) and client-side (JavaScript) languages introduces complex overhead and code duplication [12].

Node.js is built upon the V8 JavaScript engine it addresses the concurrency limitation through an event-driven, non-blocking I/O model, enabling a single server process to manage thousands of simultaneous connections with minimal resource overhead. [8] MongoDB's document-oriented model, have nothing to do with a fixed schema in favour of flexible documents, and it is particularly open to the variable and evolving data structures encountered in e-commerce contexts such as products with varied attribute sets. The use of JavaScript across both the frontend (React) and backend (Node.js/Express) further reduces context-switching costs and enables the sharing of validation schemas, utility functions, and data models.

D. Role of Third-Party APIs in Modern Web Development

The contemporary approach to web application development is increasingly characterized by the strategic delegation of non-core functionality to specialized third-party APIs, a practice consistent with the Service-Oriented Architecture (SOA) philosophy [10]. Payment processing, in particular, involves regulatory compliance, cryptographic security, and banking infrastructure concerns that are beyond the scope of a typical development team. The integration of payment gateways such as Razorpay, Stripe, or PayPal as black-box services greatly reduces these complexities and transfers the compliance burden to the service provider.

Similarly, media management by using the services like Cloudinary or AWS S3 eliminates the infrastructure load of the server file storage, offering a scalable, globally distributed Content Delivery Network with image transformation pipelines [2]. The adoption of these APIs within Stannum demonstrates a practical engineering trade-off: accepting a degree of external dependency in exchange for reduced development time, enhanced security, and improved performance.

III. SYSTEM ARCHITECTURE AND METHODOLOGY

A. MERN Stack Architectural Overview

The Stannum platform uses a client-server architectural pattern, in which the frontend and backend are kept as separate applications and interact with each other through a specified API. The frontend acts as a React.js single-page application running entirely within the client's web browser, making asynchronous HTTP requests to the backend server and displaying the user interface in accordance with the responses received. The backend acts as an Express.js application running on Node.js, offering a set of RESTful API endpoints, processing incoming requests, and interacting with the MongoDB database.

This architecture discusses several notable advantages. The frontend may be independently developed, tested, and deployed without necessitating modifications to the backend, and vice versa. Also, the same backend API can also serve multiple clients: a web browser, a native mobile application, or a third-party integration without any major modifications. This complete technology stack is combined using JavaScript,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

enabling developers to operate across the entire application with a consistent programming language.

B. Data Flow Between Frontend and Backend

The following operations are executed when a user interacts with the platform and that data requires data from the server side, like a product search, checking out a cart, etc:

- A user interaction triggers an event handler within the React component.
- Then the event handler dispatches a Redux thunk action, which encapsulates an asynchronous Axios HTTP request to the selected Express.js endpoint.
- Then the Express.js router receives this request and applies applicable middleware (authentication verification, input validation), and assigns it to the appropriate controller function.
- Then the controller function invokes the Mongoose model methods to query or alter the MongoDB database and constructs a JSON response object.
- The response is then returned to the Redux slice, which updates the state store's application, triggering a re-render of the relevant React components.

This unidirectional flow of data is consistent with the Flux architecture promoted by Facebook and implemented by Redux, which ensures that application state transitions are predictable, traceable, and testable [1].

C. Database Schema Design

MongoDB is a schema-less NoSQL database by nature, the Mongoose ODM applies to a schema at the application level, providing data validation, type casting, and middleware hooks. The Stannum database consists of the following primary collections, whose relationships are managed through document by referencing the ObjectId, rather than hard coding, this strategy helped to minimise data duplication and allowed independent document updates:

1) User Model:

It stores the user credentials, i.e. username, email, password, their roles, there exists only two roles user and admin, and a reference to a collection of associated Address documents.

2) Product Model:

This encapsulates the product metadata, including its title, description, category, brand, pricing, sale pricing, stock level, and a Cloudinary-hosted image URL. Indexing is applied on the category and brand fields to improve the experience on the admin's panel.

3) Order Model:

This model references the purchasing user and encapsulates an array of ordered products here, each containing a product reference, quantity, and price snapshot, a shipping address snapshot, the payment gateway details, and order and payment status.

4) Cart and Address Models:

The Cart model maintains a per-user array of cart items, each referencing a product ObjectId with a corresponding quantity. The Address model stores multiple shipping address documents, each belonging to a user, enabling persistent multi-address management.

D. Razorpay Payment Integration Pipeline

The payment processing workflow in Stannum follows the recommended server-side order creation pattern supported by Razorpay's documentation, which ensures that the order amount and currency are defined on the trusted server rather than on the client, reducing the risk of client-side tampering and malicious attempts. The integration pipeline proceeds through the following stages:

- Order Initiation: Upon checkout confirmation, the React frontend dispatches a request to the /shop/order/create endpoint. Then the express controller calculates the total payable amount from the server-side cart state and invokes the Razorpay Orders API to create a payment order which in return provides a unique razorpayOrderId.
- Client-Side Payment: The frontend receives the razorpayOrderId along with the public Razorpay key, which initialises the Razorpay payment modal. The user completes payment within the secure portal, which yields a razorpayPaymentId and a signature upon successful payment.
- Server-Side Verification: The frontend transmits the payment credentials to the /shop/order/verify endpoint. The Express controller reconstructs the expected SHA256 signature using the Razorpay secret key and compares it against the received signature. If there is a mismatch it results in a failed payment status else a successful match updates the order status to confirmed.

E. System Design

The system design of Stannum encompasses a comprehensive set of structural and behavioural models that collectively define the architecture, data organisation, and interaction patterns of the platform. The design phase translates the functional and non-functional requirements identified during system analysis into concrete blueprints, ensuring that the implemented system adheres to principles of modularity, scalability, and security. The following subsections detail the use case model, entity-relationship structure, data flow architecture, and class-level organization of the system.

1) Use Case Diagram

The use case diagram presented in Figure 3.1 provides a high-level behavioural model of the Stannum platform, illustrating the interactions between the two primary actors the User and the Admin and the functional capabilities exposed by the system. This diagram is particularly significant in the context of the platform's Role-Based Access Control (RBAC) architecture, as it visually demarcates the boundary between user-accessible operations and administratively restricted functionality.

From the user's perspective, the system supports a unified set of operations spanning the complete e-commerce workflow: account registration and authenticated login, product browsing and search, viewing detailed product information, managing the shopping cart, placing

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

orders, and session termination via logout. These use cases collectively represent the shopper-facing application layer, governed by standard user-level privileges enforced through JWT-based middleware on the backend.

The admin actor, by contrast, interacts with a distinct and privileged subset of system functionality. Administrative use cases include adding, editing, and deleting products from the catalogue, viewing the complete product inventory, managing registered user accounts, and monitoring order activity across the platform. Access to these operations is restricted at the API level through role-verification middleware, ensuring that no user-role principle can invoke administrative endpoints regardless of client-side state.

The intersection of both roles at the authentication boundary where both users and admins must pass through the login and credential verification process highlights the unified authentication mechanism underpinning the system, while the divergence of accessible use cases post-authentication reflects the RBAC model in practical operation.

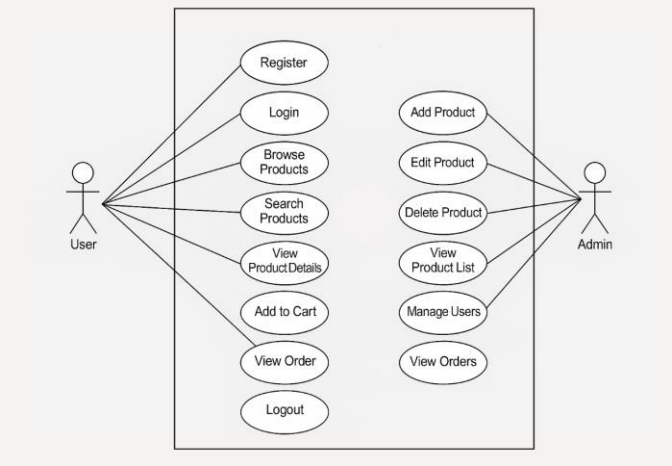

Figure 3.1: Use Case Diagram

2) Entity-Relationship Diagram

The Entity-Relationship (ER) diagram presented in Figure 3.2 provides a formal conceptual model of the data architecture underlying the Stannum platform. Although MongoDB is a document-oriented NoSQL database that does not natively enforce relational constraints, the application-level schema design implemented via the Mongoose ODM deliberately replicates the structural rigour of a relational model using ObjectId-based document referencing. This approach preserves referential integrity while retaining the schema flexibility and horizontal scalability inherent to the MongoDB paradigm.

a) Entities and their Attributes

The data model is organised around six primary entities, each corresponding to a distinct MongoDB collection:

• Customer: (Customer ID, Name, Email, Password, Address, Phone)

• Product: (Product ID, Name, Description, Price, Category, Stock)

• Cart: (Cart ID, Customer ID, Product ID, Quantity)

• Order: (Order ID, Customer ID, Order Date, Total Amount, Status)

• Order Item: (Order Item ID, Order ID, Product ID, Quantity, Subtotal)

•Payment: (Payment ID, Order ID, Payment Method, Payment Status, Transaction ID)

b) Relationship between Entities

• Customer places multiple Orders → (1:M between Customer & Order)

• An Order consists of multiple Order Items → (1:M between Order & Order Item)

• Each Order Item is linked to a Product → (M:1 between Order Item & Product)

• A Customer adds multiple Products to Cart → (1:M between Customer & Cart)

• Each Order is associated with a Payment → (1:1 between Order & Payment)

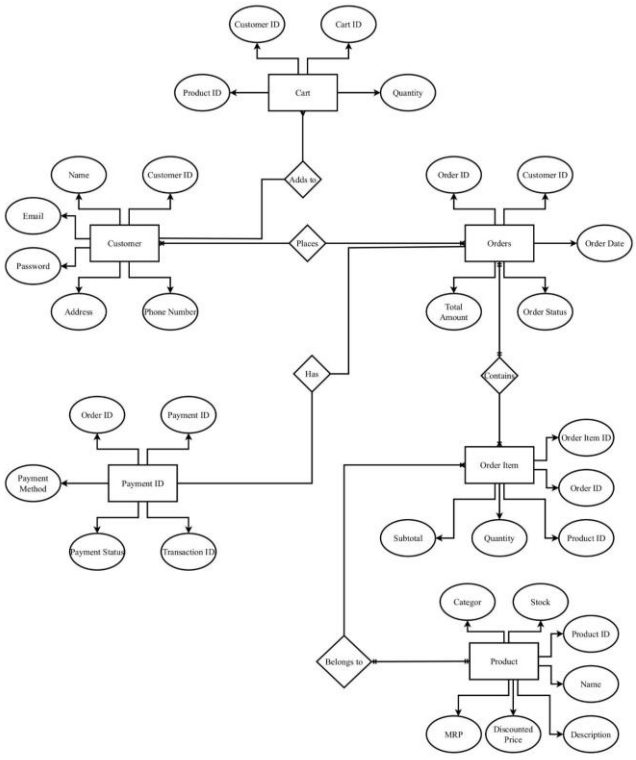

Figure 3.2: Entity Relationship Diagram

3) Data Flow Diagram

A data flow diagram (DFD) is a visual representation of how data flows within our platform. It maps out the sequence of information, actors, and steps within a system, using defined symbols to represent people and processes.

a) Level 0 DFD

This Level 0 Data Flow Diagram (DFD) provides a high-level overview of our website, representing the interaction between the external entities and the system. It primarily includes two main external entities: the Customer

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

and the Payment Gateway. The Customer interacts with the system to browse products, place orders, and make payments.

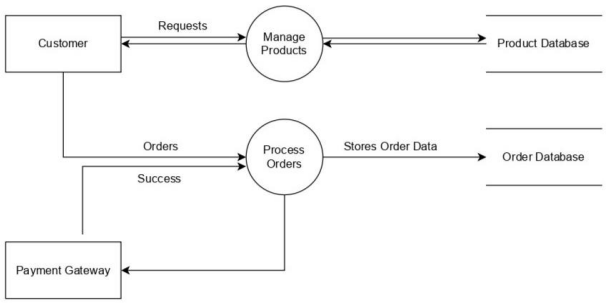

Figure 3.3: Level 0: DFD

b) Level 1 DFD

The Level 1 DFD breaks down the e-commerce system into its major processes while providing a structured overview of data flow between entities, processes, and data stores. It includes essential functions such as Browsing Products, Adding Items to Cart, Placing an Order, and User Authentication (Login/Register) to ensure smooth user interaction.

The Product Database stores product details, enabling efficient retrieval for search and filtering. The Order Database keeps track of customer purchases, while the Customer Database manages user registration details and authentication credentials. The Payment Gateway securely processes transactions and sends confirmation back to the system, ensuring seamless order completion.

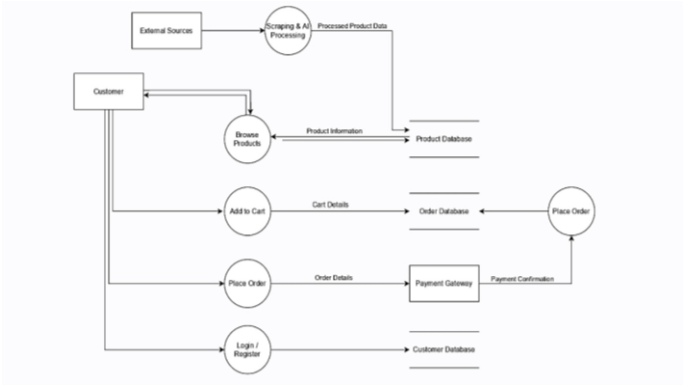

Figure 3.4: Level 1: DFD

c) Level 2 DFD

This Level 2 DFD further decomposes the main processes from Level 1 into more detailed sub-processes, offering a granular view of data flow within the system. For example, the User Authentication process is broken down into Registering a New Account, Validating Credentials, and Managing User Sessions, ensuring secure and seamless access. Similarly, the Order Placement process includes Order Verification, Processing Payment, and Updating Order Status, streamlining transaction handling. Additional processes such as Inventory Management (Updating Stock Levels, Managing Product Listings) and Customer Support (Handling Queries, Resolving Issues) further refine system functionality.

This level provides a structured roadmap for system design and implementation, ensuring clarity in data processing, storage, and interactions between different components.

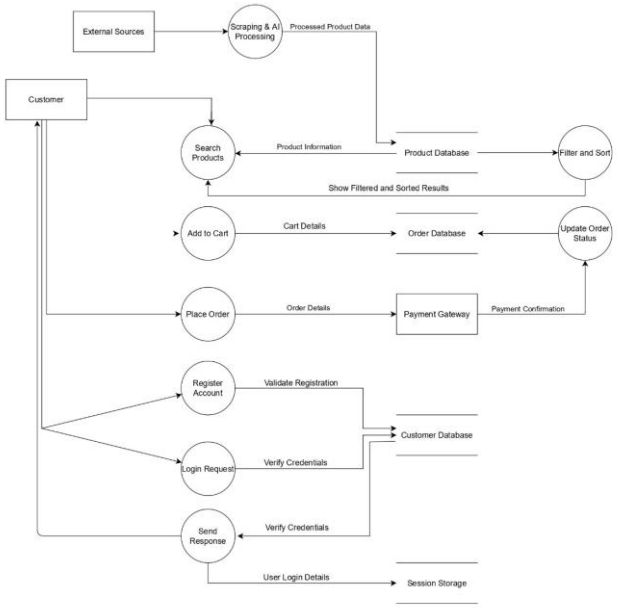

Figure 3.5: Level 2: DFD

4) Class Diagram

In the below class diagram, each class is represented as a rectangular box divided into three compartments: the top compartment shows the class name (e.g., Customer, Cart, Orders), the middle compartment lists the class attributes such as customerID, email, or orderStatus, and the bottom compartment includes any methods, such as addToCart() or placeOrder().

Lines connecting the classes represent associations and relationships between entities. For example, a one-to-many relationship connects Customer to Orders, indicating that one customer can place multiple orders. Similarly, Orders is linked to OrderItem and Payment, demonstrating how each order can include multiple items and is associated with a single payment. These associations help define how the components of the e-commerce system interact with one another.

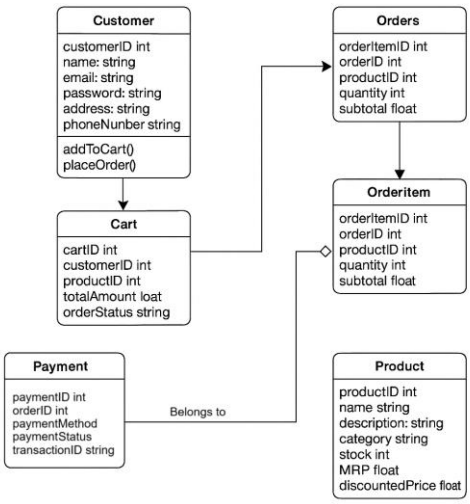

Figure 3.6:Class Diagram

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

IV. IMPLEMENTATION DETAILS

A. Frontend Implementation

1) Component-Based Architecture and Routing

The Stannum frontend is structured as a hierarchical tree of React components. The root component of the application establishes the top-level routing configuration using the React Router library which defines the protected routes that conditionally render components based on the authenticated user's role. A custom AuthGuard component is created which wraps sensitive routes which query the Redux authentication state to determine whether a redirect to the login page is warranted or not.

The application is organised into a layout-based routing where it has the ShoppingLayout component which renders the navigation header and footer, serving as a consistent component for all shopper-facing routes, the AdminLayout similarly provides a consistent sidebar navigation for administrative views. This pattern minimises redundant renders and promotes layout consistency across page navigations without full remounts.

2) State Management with Redux Toolkit

The Redux Toolkit is a solid primary state management solution for cross-cutting application flaws, which includes the authentication state, product listings, shopping cart data, and order history. The toolkit's createSlice API minimizes the boilerplate code associated with Redux by sending lifecycle actions (pending, fulfilled, rejected) to the extraReducers in the corresponding slice.

The withCredentials: true flag here is critically required to ensure that the cookie containing the web token is transmitted with CORS requests. The React Context API is used for localised UI state like toast notification management and modal visibility, where the overhead of a Redux slice would be uneven.

3) UI Design with Tailwind CSS

Tailwind CSS is by far the most efficient and easily implemented CSS framework which is employed for all visual styling and responsive layout. Its atomic class-based approach eliminates the need for a lengthy CSS file hierarchy, and allows the CSS to be added directly within the component's JSX code. This approach is particularly encouraging in the scalability and component isolation characteristics of a complex UI such as an e-commerce platform, where multiple independent styled components must be maintained properly.

Interactive UI primitives like dialog modals, dropdown menus, and accessible form controls are sourced from Shadcn/ui. It is a headless component library that provides fully accessible, styled as well as unstyled components following the most modern design patterns. This combination of shadcn/ui's semantics and Tailwind CSS's visual styling utilities enables the construction of a polished, accessible interface without the restrictions imposed by other component libraries.

B. Backend Implementation

1) RESTful API Design and Routing

The Express.js backend is structured around a three-domain routing architecture. The /api/auth namespace handles user registration, login, logout, and session verification. The /api/admin is protected by a role-verification middleware which exposes endpoints for product CRUD operations, order status management, and image upload. Whereas the /api/shop provides endpoints for product browsing, cart management, address management, order placement, and payment verification.

Each of the domains is implemented as a self-contained Express Router module, imported and mounted within the central application entry point.

2) RESTful API Design and Routing

A custom Express middleware function, authMiddleware, is applied to all protected routes. This middleware extracts the web tokens from the incoming request's cookie header, then verifies its signature using the server-side secret key, and attaches the decoded user payload which contains the userId and role, to the request object for the usage by further controller functions. If token verification fails due to expiry or any other method, a 401 Unauthorized response is returned immediately.

Role verification is then applied as a secondary middleware on administrative routes, by inspecting user role to confirm administrative privileges before allowing access to product or order management operations.

3) Cloudinary Media Upload Pipeline

Product images are uploaded using a middleware named Multer. This middleware keeps the uploaded file in the server's memory for a short time, instead of saving it directly to the server. After the image arrives, it's immediately sent to Cloudinary for secure storage. Cloudinary then gives us a URL and a unique ID for the image. This URL is stored alongside the Product data, and the frontend uses it to display the image quickly, thanks to Cloudinary's content delivery network.

C. Security Implementation

1) JWT Authentication and HttpOnly Cookies

JSON Web Tokens are used here as an authentication method for the Stannum platform. Upon successful credential verification, the server issues a signed JWT containing the user's ID and role, with a defined expiry period. This token is then transmitted to the client exclusively via a response header with the HttpOnly flag enabled, this prevents the Client-side JavaScript from accessing the contents of the response cookie and thereby minimizing the risk of Cross-Site Scripting or token exfiltration [9].

The Secure flag is additionally enabled in production deployments, restricting cookie transmission to HTTPS connections.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

2) Password Hashing with Bcrypt

User passwords are hashed before storing to the database, using the Bcrypt algorithm with a 10+ salt round factor, then a value is selected to add a computationally rigorous hashing process that completely avoids brute-force and dictionary attacks while it being achievable for typical server hardware.

During the authentication, the Bcrypt compare function is used to compare the plaintext password submission with the stored hash without requiring the storing or retrieval of the original password.

3) CORS Configuration and Environment Variables

Cross-Origin Resource Sharing (CORS) is configured on the Express.js server to clearly permit requests originating from the defined frontend origin, with credentials support enabled to allow cookie sharing across origins.

All the sensitive configuration values including the MongoDB connection string, JWT secret, Razorpay API keys, and Cloudinary credentials are saved in an .env file and can only be accessed locally or only via deployment platform. This practice ensures that no credentials are exposed within the source code repository and isn't hardcoded, which aligns with the twelve-factor application methodology [13].

V. RESULTS AND DISCUSSION

A. Build Performance and Development Tooling

The adoption of Vite as the frontend build tool produced multiple improvements in development process as compared to the default provided Create React App toolchain, which relies on Webpack for bundling. The Vite's native ES module server uses the browser's built-in module support during development, serves the source files without bundling, resulting in almost instant Hot Module Replacement capability even as the application's component count grew. When in production, Vite's Rollup-based bundler generates optimised, hierarchical output with automatic code splitting along with dynamic importing, greatly reducing the initial page loading times.

Lazy loading of route-level components are implemented via React.lazy and Suspense functions, which in turn further delays the loading of administrative and checkout components until they are explicitly navigated to, this prevents unnecessary resource usage.

B. State Management Efficiency

The integration of Redux Toolkit introduced a degree of architectural boilerplate that, while initially imposing, produced significant benefits in terms of state predictability and debuggability. The Redux DevTools browser extension enabled real-time inspection of the action dispatch sequence and state transitions throughout the development lifecycle, facilitating the rapid identification and resolution of synchronisation defects between the cart state and the server-persisted cart data.

The cartItems state, managed in the Redux store, was designed to remain synchronised with the MongoDB-persisted cart document via explicit fetch operations triggered on authenticated session initialisation and following each cart mutation. This approach ensured consistency between the client's temporary data and the server's permanent data. This is a crucial aspect in stateful e-commerce applications, where maintaining cart integrity is essential for business.

C. Payment Processing Reliability

The Razorpay integration underwent assessment through a series of test transactions, encompassing both successful payment instances and simulated failure scenarios. The server-side HMAC-SHA256 signature verification process demonstrated its efficacy in rejecting compromised payment confirmation payloads, thus safeguarding the integrity of the order confirmation pipeline. Furthermore, the latency inherent in the two-step API pattern (order creation followed by payment verification) was deemed inconsequential from a user experience standpoint, given that the Razorpay payment modal's loading phase effectively concealed the server roundtrip.

D. Scalability and Maintainability Assessment

The Stannum platform's horizontal scalability potential is facilitated by its stateless backend design: because authentication state is encoded within a self-contained JWT rather than stored in server-side sessions, multiple instances of the Express application can operate behind a load balancer without the need for shared session storage. MongoDB's distinguishing support for horizontal scaling further accommodates growth in data volume without requiring an architectural redesign.

Maintainability is encouraged through the modular routing architecture, the strict separation of controller logic from route definitions, and the consistent application of the single responsibility principle across both frontend and backend components. The use of Mongoose schemas provides an implicit data contract that serves as living documentation of the expected data structure for each collection.

VI. RESULTS AND DISCUSSION

A. Conclusion

This paper has shown the design, architecture, and implementation of Stannum, a comprehensive full-stack e-commerce platform developed using the MERN tech stack. A scalable, secure, and maintainable e-commerce system has been developed through the systematic application of established software engineering principles. These principles encompass separation of concerns, role-based access control, RESTful API design, and stateless authentication.

This study has verified the suitability of the MERN stack for the construction of production-grade commercial web applications, demonstrating that its essential technologies complement one another effectively: React's declarative UI model integrates naturally with Redux's unidirectional data flow, Node.js's non-blocking runtime aligns with the high-concurrency demands of web server workloads, and MongoDB's flexible document model accommodates the variable data structures encountered in e-commerce domains. The strategic integration of Razorpay and Cloudinary as specialised third-party APIs further validated the worth of the

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

API-first development paradigm in reducing development complexity while enhancing functional capability.

The security architecture implemented, which incorporated HttpOnly cookie-resident JSON Web Tokens (JWTs), bcrypt for password hashing, Cross-Origin Resource Sharing (CORS) configurations, and the externalization of environment variables, was observed to mitigate the primary vulnerability vectors outlined by OWASP concerning web application authentication systems [9]. The modular backend architecture exhibits potential for incremental expansion without compromising established functionalities, thereby aligning with the open-closed principle.

B. Future Scope

Several enhancements are identified as deserving future study:

- Mobile Application Development: Porting the frontend to React Native would enable the delivery of a native mobile experience for iOS and Android platforms, sharing business logica and API integration code with the existing web frontend and thereby reducing duplicated development effort.
- Microservices Migration: Dividing the huge Express backend into independently deployable microservices such as separate services for authentication, catalogue management, order processing, and notification dispatch would enhance fault isolation, independent scalability, and continuous deployment cadence.
- Real-Time Notifications: The integration of WebSocket communication (via Socket.io) would enable real-time order status notifications, inventory alerts, and administrative dashboards, enhancing the responsiveness of the platform's operational monitoring capabilities.
- Automated Testing Pipeline: The systematic execution of unit tests (Jest), integration tests, and end-to-end tests would reinforce code quality assurance and facilitate safe refactoring as the codebase evolves.
- GraphQL API Layer: Replacing or supplementing the RESTful API with a GraphQL layer would enable clients to request precisely the data fields required for each operation, reducing over-fetching and improving the efficiency of network utilisation.

In conclusion, the Stannum project serves as a demonstration of the practical applicability of modern JavaScript-based full-stack development methodologies for the construction of scalable, secure, and commercially functional e-commerce systems. The architectural decisions, implementation patterns, and security measures documented herein collectively contribute to the body of knowledge pertaining to full-stack web application engineering with the MERN stack.

ACKNOWLEDGMENT

We would like to express our heartfelt gratitude to all those who supported and guided us throughout this project. First, A special thanks to our project guide, Dr. Mohd. Nadeem for their valuable suggestions, continuous guidance, and encouragement. Their patience and insightful advice played a key role in shaping this project.

We also thank our project coordinators, Mr. Ravi Prakash Vishwakarma, Assistant Professor Department of Computer Science & Engineering, for their help in resolving both technical and non-technical issues. Despite their busy schedules, they were always ready to guide us whenever we faced any difficulties, and their support made our work much easier.

We are also grateful to the faculty and staff who supported us directly or indirectly during this project. This project would not have been possible without the contribution and support of all these individuals, and we sincerely appreciate their efforts.

REFERENCES

[1] Abramov, D. (2015). *Redux: Predictable state container for JavaScript apps* [Computer software]. GitHub. https://github.com/reduxjs/redux

[2] Cloudinary. (2023). *Cloudinary developer documentation: Image and video upload, storage, optimization and delivery*. https://cloudinary.com/documentation

[3] Fedosejev, A. (2015). *React.js essentials*. Packt Publishing.

[4] Fowler, M. (2018). *Patterns of enterprise application architecture* (2nd ed.). Addison-Wesley Professional.

[5] Garrett, J. J. (2005). Ajax: A new approach to web applications. Adaptive Path.

[6] Laudon, K. C., & Traver, C. G. (2021). *E-commerce 2021: Business, technology, society* (17th ed.). Pearson.

[7] Newman, S. (2015). Building microservices: Designing fine-grained systems. O'Reilly Media.

[8] Node.js Foundation. (2019). *Node.js documentation: About*. https://nodejs.org/en/about/

[9] OWASP. (2021). *OWASP top ten web application security risks*. Open Web Application Security Project. https://owasp.org/www-project-top-ten/

[10] Papazoglou, M. P., & Georgakopoulos, D. (2003). Service-oriented computing. *Communications of the ACM, 46*(10), 25-28. https://doi.org/10.1145/944217.944233

[11] Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.

[12] Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing, 14*(6), 80-83. https://doi.org/10.1109/MIC.2010.145

[13] Wiggins, A. (2017). *The twelve-factor app*. Heroku. https://12factor.net/