International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

# Design and Development of a MERN Stack-Based Multi-Vendor Group Buying Platform with Dynamic Bulk Discount Mechanism

**Gowrishangar S[1], Dr. K. Sangeetha[2], Mathivathanan M[1], Bharath Kumar K[1], Basava Kumar S[1]**
[1]Department of Computer Science and Engineering, SNS College of Engineering, Coimbatore, India
[2]Assistant Professor, CSE Department, SNS College of Engineering, Coimbatore, India

---

**Abstract -** Group buying enables customers to purchase products collectively and benefit from quantity-based discounts. GroupBuy is a MERN stack-based multi-vendor e-commerce platform designed to connect buyers and sellers through a centralized marketplace. The proposed system integrates product management, user authentication, group-buying activities, discount mechanisms, cart management, and order processing. Buyers can participate in group purchases to access discounted prices, while sellers can manage products and offers. The platform uses React.js for the user interface, Node.js and Express.js for backend services, and MongoDB for data storage. This paper presents the system architecture, key modules, and group-buying workflow. The project aims to encourage collaborative purchasing, improve seller reach, and make products more affordable. Further functional, security, and performance testing is required to evaluate the system's reliability and effectiveness.

**Key Words:** Group Buying, MERN Stack, Multi-Vendor E-Commerce, Bulk Discounts, Collaborative Purchasing, Online Marketplace**.**

## I. INTRODUCTION

Cloud-based social commerce is transforming how digital marketplaces aggregate consumer demand to secure wholesale pricing. Traditional e-commerce models isolate buyers with retail markups, while merchants incur high acquisition costs. Furthermore, high-velocity group buying often suffers from inventory race conditions, overselling, and manual deal reconciliation.

GroupBuy is presented as a centralized demand-aggregation marketplace built on a decoupled MERN architecture. The platform orchestrates merchant deal configurations, tiered pricing thresholds, participant pools, and automated fulfillment. Deals require aggregating a target buyer quorum before a strict deadline to unlock discounted pricing tiers. To guarantee transactional integrity, inventory decrement executes through atomic database operations (findOneAndUpdate with $gte guards) to eliminate race conditions, while an automated background scheduler (node-cron) continuously resolves deal states and performs stock rollbacks without administrative overhead.

This paper makes three design contributions: (1) unifying the collective-purchasing journey from pool aggregation to order fulfillment within a single decoupled platform; (2) demonstrating how atomic database constraints, cron-based lifecycle engines, and global authentication interceptors prevent overselling and data anomalies; and (3) defining state-validation safeguards that maintain transaction and session integrity under high concurrent loads.

## II. LITERATURE REVIEW

Research in social commerce underscores the efficacy of collective-purchasing models in aggregating demand, reducing merchant customer acquisition costs (CAC), and unlocking tiered wholesale pricing [1], [2]. However, sustained operational viability requires real-time synchronization of shared deal metrics, as uncoordinated buyer aggregation and ambiguous quorum boundaries undermine transaction integrity and consumer trust.

Studies on high-concurrency transaction processing highlight critical vulnerabilities regarding race conditions and overselling during traffic surges [3]. Conventional read-modify-write routines (find then save) fail under simultaneous loads by reading stale stock values, resulting in negative inventory balances [4]. Mitigating these anomalies demands database-level atomic, conditional operations that evaluate stock availability and execute decrements within a single atomic cycle.

Literature on distributed marketplace architectures emphasizes deterministic state handling, automated reconciliation, and resilient session management [5]. Unmonitored group pools risk severe inventory fragmentation if manual intervention is required to resolve deal outcomes; automated background workers guarantee deterministic inventory rollbacks for unmet quorums and instant settlement for successful pools [6]. Furthermore, coupling stateless token authentication with client-side response interceptors prevents session invalidation errors and eliminates submission deadlocks.

Systems-oriented research on distributed marketplace architectures emphasizes the necessity of deterministic state machines, automated reconciliation, and resilient session handling [5]. In quorum-dependent group buying, unfulfilled or stalled pools present a severe risk of merchant inventory fragmentation if administrative intervention is required to close deals. Automated background schedulers and idempotent lifecycle workers ensure that expired deals trigger immediate inventory rollbacks to merchant catalogs, while successful pools transition reliably into bulk order queues [6]. Furthermore, maintaining stateless authentication architectures supported by active client-side response interceptors ensures that credential expirations and transient session faults are intercepted and handled cleanly without causing corrupted order submissions or application deadlocks.

## III. METHODOLOGY

### *A. Study Scope and Design Approach*

This study is organized as an architectural systems-design case study of a cloud-native, high-concurrency collective commerce marketplace. The research evaluates how demand aggregation, tiered pricing models, and time-delimited purchasing pools can be deployed reliably on decoupled web architectures without data corruption. The analytical framework defines the actors and system services, formalizes the state machine governing deal lifecycles, and outlines the precise architectural layers responsible for policy enforcement, atomic transactions, persistent storage, and automated reconciliation.

The primary design objective is to deliver transparent, collective volume discounts to end consumers while eliminating inventory anomalies and operational overhead for merchants. The system model formally partitions operational boundaries across three primary actors:

All permissions and mutations are validated server-side using Role-Based Access Control (RBAC). Client applications are treated as untrusted presentation interfaces; application state, tier qualification, and inventory counts rendered in the browser are considered strictly advisory until verified against persistent database record. A buyer can inspect deal metrics and initiate checkout for authorized items, but cannot access another user's order records or tamper with pricing structures; a merchant manages only their assigned product catalog and deal lifecycles; and client applications cannot bypass server-side business rules by manipulating browser state or client-side storage keys.

### *B. Functional Model and Candidate Data*

The functional model contains several connected modules. Deal management stores the product description, base pricing, and tier policy. Participant management stores buyer details and links the user to a pool or purchase campaign. Deal ingestion validates a submitted catalog, stores it in a protected collection, and invokes checks to verify available stock, deadlines, targets, quotas, and pricing. Rendered metrics are presented for buyer review because network latency, concurrent submissions, and incomplete data can produce discrepancies. The primary record should remain accessible to authorized merchants so that visual counts can be checked against the source.

Tier-requirement comparison relates the captured information to quorum rules. A valid transaction should verify reserved inventory and active deadlines, not trust only a client-side calculation. Checkout management assigns sessions, routing, addresses, pricing rules, and status. Concurrency control executes atomic database updates and stock decrements when a purchase mutation is triggered. Scheduler management records cron triggers, deadline checks, rollbacks, and settlements. Reporting consolidates the volume and the pool status, while an audit trail records significant changes to inventory, access rights, deal status, and final settlements.




*Fig. 1. Proposed system overview of Aegis Recruit.*

### *C. Four-Phase Deal and Checkout Progression Workflow*

*The workflow begins when a merchant creates a deal listing and configures product stock or tier thresholds. Catalog browsing supports initial discovery, but it does not itself reserve inventory or generate an order. Buyers who participate in the pool complete the configured phases in the following order:*

- ***Phase 1 — Tier Selection and Routing:*** *Buyers select a desired discount tier and target pool. The system transfers pricing targets, tier identifiers, timestamps, and deal metadata into ephemeral route state. An active user token makes the checkout request eligible for Phase 2 under the platform policy.*

- ***Phase 2 — Delivery Capture and Re-Validation:*** *Authenticated buyers provide shipping addresses and contact details. When submitted, a server-side controller recalculates pricing tiers against persistent database records. Verification guards ensure the requested quantity, active status, and address integrity are satisfied. Validated transaction parameters make the order eligible for Phase 3.*

- ***Phase 3 — Atomic Inventory Reservation:*** *The backend executes a single-operation atomic query to decrement stock and add the buyer. Transactions are evaluated through conditional filters according to remaining inventory counts. Any order record or pool entry is committed under a pending status and remains auditable for merchant fulfillment.*

- ***Phase 4 — Automated Pool Resolution:*** *An authorized background daemon evaluates the deal's deadline, participant progress, stock levels, and relevant exceptions. The scheduler records the final outcome, such as completed, expired, or cancelled, with an inventory rollback where appropriate.*

*The pool progression gates are business rules, not visual labels. The backend should calculate or verify the price and stock from persisted catalog data, apply the configured boundary rule consistently, and store the resulting transaction state. A transaction with depleted stock, invalid pricing, or an expired deal deadline should not be silently advanced. The application should use explicit states for pending-pool, completed, expired, cancelled, and rollback-required outcomes, with error handling and state recovery defined by the marketplace policy.*




*Fig. 2. Proposed workflow.*

### *D. System Architecture and Data Flow*

The proposed architecture separates the presentation layer, application workflow, background scheduling engine, persistent data storage, ephemeral state management, and external cloud services. The presentation layer provides merchant, administrative, and buyer experiences. The application layer validates requests, checks permissions, coordinates deal and product records, processes checkouts, calculates or verifies tier prices, applies deal transitions, and triggers inventory rollbacks. These responsibilities should remain authoritative on the server so that users cannot alter pricing, stock, or eligibility by changing client-side state

The background scheduling engine supports deadline tracking, quorum verification, automated pool resolution, stock rollback triggers, dynamic tier updates, and fulfillment dispatching. A decoupled cron-based worker interface isolates the checkout workflow from scheduled polling overhead. Before being committed or broadcast, deal state transitions should be checked for quorum compliance and evaluated for inventory consistency. Where practical, the system should retain enough context—such as participant counts, threshold targets, and execution timestamps—to help an authorized merchant understand the resolution outcome. An optimistic client-side counter or mock status must not be represented as an authoritative transaction state.

The data layer holds structured records such as users, roles, deals, products, pricing tiers, pool participants, orders, delivery addresses, stage transitions, and audit events. Product catalog images and merchant assets should be stored in protected cloud storage, with asset metadata linked to database schemas. The server should check authorization before serving order documents or merchant reports




*Fig. 3. Proposed system architecture and external-service boundaries.*

### *E. Security, Privacy, and Responsible AI*

Marketplace records include personal details, credentials, shipping addresses, payment preferences, pool reservations, and financial transactions. Access should be restricted by role and ownership, connections should use secure transport, and stored records should not be publicly accessible by default. Input validation is needed for forms, address fields, checkout submissions, and API calls. Audit events should record important status changes and inventory updates with timestamps and authorized-user information.

Buyers should receive clear notice before any financial hold or pool commitment begins, including what is charged, how quorums operate, how long deals run, and how rollbacks occur. Accessible fallback mechanisms should be provided where network instability interrupts the checkout flow. Client-derived discounts and tier summaries should be advisory, catalog-linked, and server-verified. Client-side price tampering or forged cart parameters should not be treated as legitimate purchase evidence

## IV. RESULT

### *A. Prototype and Design Outcomes:*

The available project artifacts define the main journey across merchant and buyer functions. The merchant-facing prototype includes a dashboard, catalog and inventory navigation, deal configuration, pool quorum monitoring, order fulfillment, reports, and settings. Buyer-facing activities are represented as a sequence of deal discovery, tier selection, decoupled checkout, and order status tracking.

*Fig. 4. Illustrative recruiter dashboard prototype.*




*Fig. 5. Illustrative interface with sample.*

The deal-settlement screen brings pool participation indicators and the final order fulfillment status into one view. This design is intended to help a merchant inspect buyer commitment tallies and stock reservation logs alongside the automated settlement record before initiating physical dispatch.




*Fig. 6. Illustrative interface with sample*

At the workflow level, the reported design outcomes are the explicit four-phase deal and checkout lifecycle, the proposed quorum milestones and threshold gates for tier activation, the separation of client route-state and atomic persistence responsibilities, and the automated reconciliation of pool commitments in a settlement record.

## V. DISCUSSION

The principal design benefit is continuity across the collective commerce lifecycle. A centralized deal and order record can reduce repeated transfer of state parameters and make the current transaction phase easier to identify, provided that the tier configuration, buyer commitments, atomic stock reservations, and settlement status remain linked to persistent database records. A consistent quorum policy can improve the predictability of group purchases, while a consolidated settlement ledger can make it easier for authorized merchants to inspect the transaction history behind a fulfillment batch. These benefits are plausible design goals; their magnitude has not yet been measured in the available materials.

The security model is not an optional addition to the workflow. A platform that collects resumes, code, interview responses, and hiring decisions needs authorization checks at the data-access layer, not only protected navigation in the interface. Organization-level isolation is essential in a multi-company system. Coding submissions must be treated as untrusted input and executed only in a suitably isolated service. Interview recording and monitoring need explicit candidate notice, purpose limitation, retention controls, and access records. The project should document which of these controls are implemented and which remain planned.

### *B. Limitations and Validation Plan*

*The main limitation is the absence of verified empirical evidence. Prototype images and architectural schemas do not demonstrate that authentication, route-state forwarding, atomic inventory reservation, cron-based lifecycle monitoring, automated stock rollbacks, order tracking, or merchant notifications operate reliably end to end under live traffic. Likewise, diagrams of external cloud boundaries do not establish that database clusters and hosting environments are configured, secured, or hardened against extreme load. The current paper therefore reports system design and prototype coverage, not production readiness or empirical performance superiority.*

*A reproducible evaluation should begin with unit tests for tier discount calculations, quorum threshold evaluations, and permitted deal stage transitions. API integration tests should verify that buyer reservations and order documents are persisted atomically and that direct API submissions cannot bypass server-side pricing or stock guards ({ stock: { $gte: quantity } }). Role-based access tests should cover buyer transaction privacy, merchant catalog management, and administrative system metrics. Payload-handling tests should examine malformed checkout bodies, boundary price manipulations, and unauthorized mutations. The background scheduler should be tested for timer drift, execution idempotency, recovery from thread stalls, and accurate state transitions during concurrent deal expirations.*

.

## VI. CONCLUSION AND FUTURE WORK

GroupBuy proposes an integrated collective-purchasing marketplace that connects merchant catalog management, tiered pricing configurations, decoupled checkout routing, atomic inventory reservations, automated lifecycle scheduling, and final order fulfillment. Its core design contribution is a single traceable workflow with explicit concurrency guards, database-level atomic decrements, and consolidated deal settlement ledgers. The accompanying prototypes illustrate how merchants and consumers could review tier progression, active pool metrics, and transaction outcomes in a centralized workspace.

## ACKNOWLEDGMENT

The authors acknowledge the Department of Computer Science and Engineering, SNS College of Engineering, Coimbatore, for supporting the academic project. The authors also thank the project guide for guidance during the design and documentation process.

## REFERENCES

[1] R. J. Kauffman, L. Lai, and H. Wang, “Social commerce and group buying: Research trends, business models, and consumer dynamics,” *Electronic Commerce Research and Applications*, vol. 11, no. 2, pp. 85–92, 2012, doi: 10.1016/j.elerap.2011.12.005.

[2] D. Jing and F. Xie, “Group buying on the web: A tool for price discrimination and demand aggregation,” *Journal of Revenue and Pricing Management*, vol. 10, no. 3, pp. 244–258, 2011, doi: 10.1057/rpm.2010.5.

[3] X. Liu, J. He, and Y. Tan, “Mitigating inventory overselling under high concurrency in online flash sales,” in *Proc. IEEE Int. Conf. on E-Business Engineering (ICEBE)*, 2018, pp. 112–119, doi: 10.1109/ICEBE.2018.00025.

[4] B. Borisov and S. Rashkov, “Concurrency control mechanisms and transaction anomalies in document-oriented NoSQL databases,” in *Proc. Int. Conf. on Computer Systems and Technologies (CompSysTech)*, 2020, pp. 45–52, doi: 10.1145/3414592.3414601.

[5] J. Zhang, Y. Zhao, and G. Huang, “Architecture design and state synchronization for distributed collective-purchasing platforms,” *Journal of Systems Architecture*, vol. 115, p. 102012, 2021, doi: 10.1016/j.sysarc.2021.102012.

[6] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1987, pp. 249–259, doi: 10.1145/38713.38742.

[7] ISO/IEC/IEEE, *Systems and software engineering — Life cycle management — Risk management*, ISO/IEC/IEEE Standard 16085:2021, 2021, doi: 10.1109/IEEESTD.2021.9429712.

[8] OWASP Foundation, *OWASP Application Security Verification Standard (ASVS)*, Version 5.0.0, 2025. [Online]. Available: https://asvs.owasp.org.