



ARKA: Design, Implementation and Security Evaluation of a Secure E-Commerce Web Platform with Server-Authoritative Pricing and Real-Time Collaborative Shopping

Diksha Subhash Kumbhare¹, Nirjala Subhash Kumbhare²

¹ Dept. of Computer Science (MCA), Sardar Patel College, Balaghat, India

² Dept. of Commerce & Management (MCM), VMV College, Nagpur, India

Corresponding Author Email: dikshakumbhare10@gmail.com, nirjalakumbhare16@gmail.com

Abstract—

How to Cite this Article:

Kumbhare, D. S. & Kumbhare, N. S. (2026). ARKA: Design, Implementation and Security Evaluation of a Secure E-Commerce Web Platform with Server-Authoritative Pricing and Real-Time Collaborative Shopping. International Journal of Creative and Open Research in Engineering and Management, 10(10), 1-9.
<https://doi.org/10.55041/ijcope.v2i10.017>

License:

This article is published under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

© The Author(s). Published by International Journal of Creative and Open Research in Engineering and Management.



<https://doi.org/10.55041/ijcope.v2i10.017>

Business-logic flaws in checkout flows, where a server trusts prices, totals or roles asserted by the client, remain a leading cause of financial loss in web commerce, and security studies continue to find them in widely deployed platforms. At the same time, mainstream storefronts treat shopping as a single-user activity and offer no synchronous way for several people to build one cart together. This paper presents the design, implementation and security evaluation of **ARKA** (), a full-stack e-commerce web platform built with Next.js 16 (App Router), React 19, TypeScript, Prisma ORM and PostgreSQL. ARKA places four controls in sequence: (i) a sliding-window rate limiter and (ii) JSON Web Token (JWT) verification with role-based access control (RBAC), both executed in Edge Middleware before any route handler; (iii) Zod schema validation of every request body; and (iv) a server-authoritative Canonical Pricing Engine that accepts only product identifiers and quantities from the client and recomputes the subtotal, promotional discount, shipping charge and 18% GST from database prices. A Real-Time Collaborative Shopping Room lets several authenticated users share one server-held cart over WebSocket. In a 14-day controlled evaluation, none of 50 price-tampering attempts altered a persisted order total, none of 75 authorization-bypass attempts reached a protected handler, all 85 requests above the login rate limit were rejected with HTTP 429, and 200 collaborative cart operations across three concurrent users produced no state divergence, with median update latencies of 31–34 ms.

Keywords— Business Logic Vulnerabilities; Collaborative Shopping; E-Commerce Security; JWT; Next.js; Parameter Tampering; Prisma ORM; Rate Limiting; RBAC; Server-Authoritative Pricing; WebSocket; Zod.



I. INTRODUCTION

Online retail now carries a large share of everyday purchasing, and the security of a store is judged less by its cryptography than by whether its business rules can be bent. Attacks on e-commerce frequently target *logic* rather than memory-safety bugs: Wang et al. [1] showed that inconsistencies between merchant software and third-party payment services allowed purchases at arbitrary prices, or without payment at all, on real stores; Sun et al. [2] used static analysis to uncover 11 previously unknown logic vulnerabilities in widely deployed open-source e-commerce software; and a 2023 study by Pagey et al. [3] of 32 e-commerce platforms, together hosting about 10 million stores, reported 37 serious vulnerabilities, including weaknesses on six platforms that allowed customers to shop for free.

A common root cause is a misplaced trust boundary. Order totals, discounts and even user roles are computed or held in the browser and then accepted by the server as submitted. Anyone with browser developer tools or an intercepting proxy can edit such values before they leave the client. The OWASP Top 10:2021 ranks Broken Access Control as the most serious web application risk [4], and the OWASP API Security Top 10 (2023) separately lists broken object-property-level authorization, unrestricted resource consumption and unrestricted access to sensitive business flows [5], all of which apply directly to checkout and login endpoints.

Modern full-stack frameworks make the boundary easy to blur. In a React application it is natural to compute a cart total inside a client component and to forward it to a checkout endpoint, and it is equally natural to "protect" a page by hiding navigation links when a user is not logged in. Neither practice protects the underlying API: a direct HTTP request bypasses the user interface entirely.

There is also an experiential gap. Shopping is often social — families choosing an appliance, friends planning a purchase, a buyer and an approver agreeing an order — yet most storefronts offer only single-user carts or asynchronous shared wish-lists. Zhu et al. [6] showed empirically that the design of collaborative-shopping support measurably affects how co-shoppers coordinate and how they perceive the shared experience.

This paper describes ARKA, a full-stack e-commerce platform designed so that these failures are excluded by construction rather than by careful coding at each endpoint. Its contributions are:

1. a layered request pipeline that rejects abusive, unauthenticated and unauthorized traffic in Edge Middleware, before application code runs;
2. a server-authoritative pricing model, stated as Eqs. (2)–(6), in which no client-supplied financial field enters any term, and its implementation as a Canonical Pricing Engine;
3. a server-authoritative collaborative cart protocol over WebSocket that keeps every participant's view identical; and
4. a controlled evaluation across four test suites covering price tampering, authorization bypass, brute-force login and collaborative consistency.

Section II reviews related work. Section III states the problem and threat model. Section IV describes the system design, Section V the implementation, Section VI the experimental setup and Section VII the results and their limitations. Section VIII concludes.

II. RELATED WORK

A. Logic Vulnerabilities in E-Commerce

The studies above [1]–[3] are diagnostic: they find and measure logic flaws in existing systems. Wang et al. [1] analysed merchant applications integrated with Cashier-as-a-



Service providers and showed how state inconsistencies between the two parties could be exploited. Sun et al. [2] applied symbolic execution and taint analysis to detect payment-logic violations statically. Pellegrino and Balzarotti [7] proposed a black-box technique that infers behavioural patterns from a few recorded user traces of a web application and generates test cases that violate them, detecting logic flaws without access to source code. Pagey et al. [3] extended the analysis beyond checkout to merchant dashboards and storefront APIs, reporting store-takeover, data-exposure and free-shopping weaknesses. ARKA is a *constructive* counterpart: rather than detecting tampering after deployment, it removes the client's ability to influence financial values at all.

B. Access Control and Token-Based Authentication

Role-based access control was standardised by NIST [8], which defines users, roles, permissions and sessions as the core model. JSON Web Tokens [9] carry such claims in a signed, compact form suitable for stateless verification. The OWASP Top 10 recommends implementing access control once and reusing it throughout the application, denying by default [4]. Placing verification inside each API handler violates this principle: a single handler that omits the check exposes its resource. ARKA therefore centralises verification in middleware, which executes before every matched route.

C. Rate Limiting

HTTP status 429 (Too Many Requests) and its optional Retry-After header are defined in RFC 6585 [10]. Exact sliding-window limiting counts every request in the preceding window; at very large scale, providers approximate it by weighting the previous fixed window's count, as described by Cloudflare [11]. OWASP lists unrestricted resource consumption as an API risk, with

credential-guessing against login endpoints as a typical consequence [5].

D. Collaborative Shopping and Shared State

Beyond the user-centred evidence of Zhu et al. [6], keeping replicated state consistent is a long-studied problem. Operational transformation [12] and conflict-free replicated data types (CRDTs) [13] let replicas edit concurrently and still converge. Both target settings where clients may edit independently or offline. In commerce, every change must be validated against inventory and pricing on the server anyway, so ARKA adopts a simpler design in which the server is the single authority and clients only render its snapshots (Section IV-F).

TABLE I
SUMMARY OF RELATED WORK

Work	Focus	Gap addressed by ARKA
Wang et al. [1]	Logic flaws between merchants and payment services	Analysis only; no reference design
Sun et al. [2]	Static detection of payment-logic bugs	Detects flaws; does not prevent them
Pellegrino & Balzarotti [7]	Black-box logic-flaw testing	Testing method, not a platform
Pagey et al. [3]	Security of 32 e-commerce platforms	Measurement study
Zhu et al. [6]	Collaborative shopping support	User study; no secure implementation
OT [12], CRDT [13]	Convergence of replicated state	No commerce validation
ARKA (this work)	Prevention by design + shared cart	—

III. PROBLEM STATEMENT AND THREAT MODEL

The objectives of ARKA are: **O1**, the persisted value of every order is determined only by server-side data; **O2**, no protected operation executes for a caller without a valid token and a sufficient role; **O3**, automated clients cannot make unbounded attempts against sensitive endpoints; and **O4**, all participants in a shared cart always observe the same state.



We consider three adversaries. **A1** is an anonymous Internet client. **A2** is an authenticated customer who uses an intercepting proxy and can therefore modify, add, remove or replay any field of any request, including cookies. **A3** is an automated script issuing high-rate requests, for example password guessing. Compromise of the server, the database or the JWT signing secret, network attacks against TLS and volumetric denial of service are outside the scope of this work. Table II lists the threats and the control that addresses each.

TABLE II
THREATS AND CORRESPONDING CONTROLS

Threat	Adv.	ARKA control
T1 Tampered price or total	A2	Prices read from database (IV-E)
T2 Forged promo or discount	A2	Server-only promo table (IV-E)
T3 Access to admin endpoints	A1, A2	Middleware RBAC, 403 (IV-C)
T4 Missing, expired or forged token	A1	Signature + expiry check, 401 (IV-C)
T5 Brute-force login	A3	Sliding-window limit, 429 (IV-B)
T6 Malformed or extra fields	A2	Zod schema, 400 (IV-D)
T7 Personal data in logs	—	Redacting logger (IV-G)
T8 Divergent shared cart	—	Server-held room state (IV-F)

IV. SYSTEM DESIGN

A. Architecture Overview

Every API request passes through the same ordered pipeline (Fig. 1): Edge Middleware applies the rate limiter and then JWT/RBAC checks; the route handler parses the body against a Zod schema; the service layer performs business logic, including the Canonical Pricing Engine; and Prisma ORM reads and writes PostgreSQL. A request rejected at any stage receives an error response and never reaches later stages. Security-relevant events from every stage are written to a structured audit log with personal data redacted.

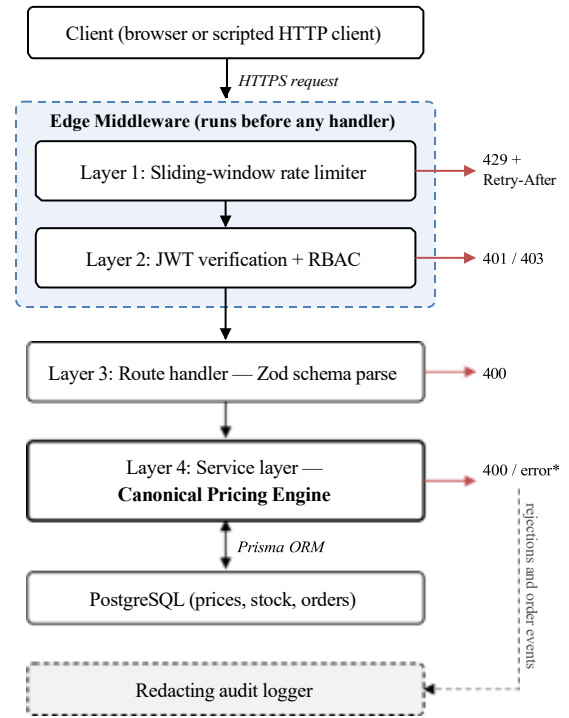


Fig. 1. ARKA request pipeline. A request rejected at any layer (red arrows, right) receives the status code shown and never reaches later layers. *An unknown product returns 400; insufficient stock raises InsufficientInventoryError.

B. Layer 1: Sliding-Window Rate Limiting

For each incoming request the middleware derives a key k from the client IP address and an endpoint class $c \in \{\text{read}, \text{write}\}$. It counts the requests with that key and class in the preceding window of $W = 60$ s:

$$N_{k,c}(t) = |\{\tau \in \log_{k,c} : t - W < \tau \leq t\}| \quad (1)$$

The request is admitted only if $N_{k,c}(t) < L_c$, with $L_{\text{write}} = 15$ and $L_{\text{read}} = 60$ requests per minute. Otherwise the middleware returns HTTP 429 with a Retry-After header [10], and no application code runs. Algorithm 1 states the rule in its exact (sliding-log) form.

Algorithm 1 Sliding-window admission

1. **input:** request r ; key $k \leftarrow \text{clientIP}(r)$; class c
2. $t \leftarrow \text{now}()$
3. discard from $\log[k,c]$ all $\tau \leq t - W$
4. **if** $|\log[k,c]| \geq L_c$ **then**
5. **return** 429, Retry-After: W
6. **end if**
7. append t to $\log[k,c]$; forward r to Layer 2



C. Layer 2: JWT Verification and RBAC

Session state is carried in a signed JWT [9] stored in an HTTP-only cookie named *auth-token*, so it is not readable by page scripts. The middleware verifies the token's signature against the server-held JWT_SECRET, and its expiry, with the *jwtVerify* function of the *jose* library, which runs in the Edge runtime without Node.js dependencies. On success, the decoded claims, including *role*, are forwarded to the handler as a request header; route requirements are then checked against that role. Table III gives the resulting access policy.

TABLE III
ACCESS POLICY ENFORCED IN MIDDLEWARE

Endpoint class	No / expired token	Customer	Admin
Public catalogue reads	allowed	allowed	allowed
Checkout, orders, rooms	401	allowed	allowed
/api/admin/*	401	403	allowed

D. Layer 3: Schema Validation

Every handler begins by parsing its body against a Zod schema. The order-creation schema (abridged) is:

```
OrderSchema = z.object({
  items: z.array(z.object({
    productId: z.string().min(1),
    quantity: z.number().int().positive().max(99),
    selectedColor: z.string().optional(),
    selectedSize: z.string().optional(),
  })).min(1).max(50),
  shippingAddress: z.object({
    name: z.string().min(2).max(100),
    email: z.string().email(),
    phone: z.string().regex(/^([6-9]\d{9})$/),
    address: z.string().min(5),
    city: z.string().min(2),
    zipCode: z.string().regex(/^\d{6}$/),
    country: z.string().min(2),
  }),
  paymentMethod: z.enum(['COD', 'CARD', 'UPI']),
  promoCode: z.string().max(20).optional(),
})
```

The schema bounds quantities (1–99 per line, 1–50 lines), validates Indian mobile numbers and six-digit PIN codes, and restricts payment methods to an enumeration. Two properties matter for security. First, the schema contains *no* price, subtotal or total field. Second, Zod's

z.object removes unrecognised keys during parsing by default, so a *total* or *price* field injected by an attacker is discarded before the service layer receives the data. A parse failure returns HTTP 400 with per-field messages, and no database query is made.

E. Layer 4: Canonical Pricing Engine

The engine receives only (*productId*, *quantity*) pairs and an optional promotional code. It loads each product from the database, checks stock, and computes every amount from server-side values (Algorithm 2).

Algorithm 2 Canonical order pricing

```
1. input: items = [(idi, qi)], promo code p
2. subtotal ← 0
3. for each (idi, qi) in items do
4.   prod ← getProductById(idi)
5.   if prod = ∅ then reject (unknown product)
6.   if prod.stock < qi then throw InsufficientInventory
7.   subtotal ← subtotal + prod.price × qi
8. end for
9. discount, freeShip ← lookup(p, VALID_PROMOS)
10. shipping, tax, total ← Eqs. (4)–(6)
11. return (subtotal, discount, shipping, tax, total)
```

With *price_i* the database unit price of product *i* and *q_i* its validated quantity:

$$subtotal = \sum_i price_i \times q_i \quad (2)$$

For a code *p* present in the server-resident table VALID_PROMOS, with percentage *pct_p* and cap *cap_p*:

$$discount = \min(subtotal \cdot pct_p / 100, cap_p) \quad (3)$$

and discount = 0 for any other code. The promo table is never sent to the client. Shipping, with threshold *T* = ₹999 and fee *F* = ₹149, is:

$$shipping = 0 \text{ if } subtotal \geq T \text{ or } freeShip_p; \text{ else } F \quad (4)$$

Goods and Services Tax at the standard rate *r* = 0.18 is applied to the discounted amount:

$$taxable = \max(0, subtotal - discount) \\ tax = \text{round}(taxable \times r) \quad (5)$$

$$total = \max(0, taxable + tax + shipping) \quad (6)$$



Integrity property. Write a checkout request as $q = (I, p, \phi)$, where I is the validated item list, p the promo code and ϕ every other client-controlled value (any injected price or total, headers, field order). By Eqs. (2)–(6), $\text{total}(q) = F(I, p; S)$, where S is server state (catalogue prices, stock, promo table and constants). Because ϕ is not an argument of F , $\text{total}(I, p, \phi) = \text{total}(I, p, \phi')$ for every ϕ, ϕ' . A client can choose *what* to buy and which code to try, but never *what it costs*. This is objective O1.

TABLE IV
SERVER-SIDE PRICING PARAMETERS

Parameter	Symbol	Value
Free-shipping threshold	T	₹999
Flat shipping fee	F	₹149
GST rate	r	18%
Max. quantity per line	—	99
Max. lines per order	—	50
Promo table	VALID_PROMOS	server only

F. Real-Time Collaborative Shopping Room

A room is created by one authenticated user, the host, who receives an 8-character alphanumeric room code and shares it with collaborators. Each collaborator joins from their own authenticated session. The server keeps one WebSocket connection per participant and holds the room's cart; clients never hold an authoritative copy.

When a participant adds, removes or changes the quantity of an item, the client sends an *intent*, not a new cart. The server authenticates the sender, checks room membership, validates the item, applies the change to the room cart, recomputes the canonical total and broadcasts a CART_UPDATE event carrying the complete snapshot to the other participants (Fig. 2):

```
CartUpdateEvent = {
  type: "CART_UPDATE",
  roomId: string,
  actorId: string,
  actorName: string,
  items: CartItem[],
  updatedAt: ISO8601 timestamp
}
```

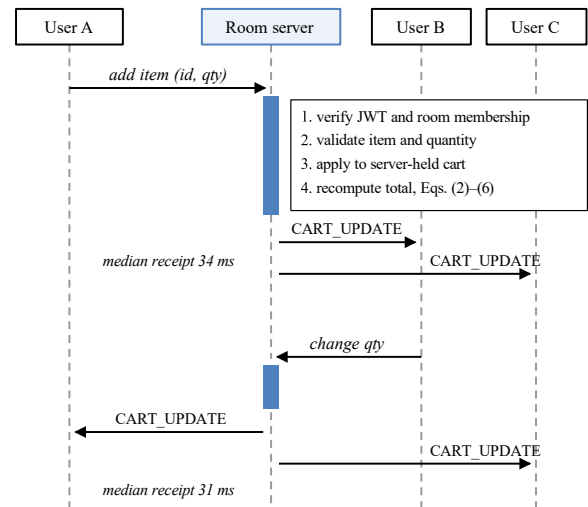


Fig. 2. Collaborative Shopping Room message flow. Clients send intents; the server validates, updates the single authoritative cart and broadcasts a full snapshot to the other participants. Latencies are the medians measured in Section VII-D.

Consistency. Because all changes to a room pass through one server-side handler, they are applied in a single total order, and each broadcast carries the full cart rather than a delta. A participant that misses or receives a message late is corrected by the next snapshot, so all views converge to the server state (objective O4). This avoids the transformation or merge logic of OT [12] and CRDTs [13], at the cost of requiring the server to be reachable — an acceptable trade-off for checkout, which needs the server in any case.

Room codes. If codes are drawn from 36 symbols (A–Z, 0–9), an 8-character code has $36^8 \approx 2.8 \times 10^{12}$ possible values. Guessing is further limited because joining requires an authenticated session and passes through the rate limiter.

G. Redacting Audit Logger

ARKA writes a structured log entry for every order creation, authentication event and security violation (rate-limit, 401, 403 and validation rejections). Before an entry is written, a redaction pass replaces email addresses, phone numbers and full names with anonymised tokens, so logs can be shipped to external monitoring services



without carrying customers' personal data. This supports the data-minimisation intent of India's Digital Personal Data Protection Act, 2023 [14].

V. IMPLEMENTATION

ARKA is implemented in TypeScript end to end. Table V lists the technology stack. Pages and API route handlers use the Next.js App Router [15]; the request-interception layer is a Next.js middleware function running on the Edge runtime. Validation uses Zod [16], database access uses Prisma ORM [17] over PostgreSQL, token verification uses *jose* [18], and the shopping rooms use the WebSocket protocol [19].

TABLE V
TECHNOLOGY STACK

Concern	Technology
Framework, routing	Next.js 16 (App Router) [15]
User interface	React 19
Language	TypeScript
Request interception	Next.js Edge Middleware
Input validation	Zod [16]
Data access	Prisma ORM [17]
Database	PostgreSQL
Authentication	JWT [9] via <i>jose</i> [18], HTTP-only cookie
Real-time rooms	WebSocket [19]
Payment	Mock flow: COD, card, UPI

The principal data entities are shown in Table VI. Orders store the five amounts produced by the pricing engine together with the promotional code, so every persisted order can be re-verified against Eqs. (2)–(6).

TABLE VI
PRINCIPAL DATA ENTITIES

Entity	Key fields
User	id, name, email, credentials, role
Product	id, name, price, stock, colour and size options
Order	id, user, subtotal, discount, tax, shipping, total, promoCode, paymentMethod, shippingAddress
Order item	order, productId, quantity, selectedColor, selectedSize
Audit entry	event type, time, redacted details

VI. EXPERIMENTAL SETUP

The security architecture was evaluated in a controlled test over 14 days against a test deployment of ARKA. Four scripted suites were run:

S1, price tampering (A2). 50 checkout requests were intercepted and their *total* field replaced with ₹1 before submission. In three of them the product identifiers were also replaced with values that do not exist in the catalogue. Pass criterion: no persisted order reflects a client-supplied amount.

S2, authorization bypass (A1, A2). 75 direct HTTP requests were sent to the administrative endpoints */api/admin/products* and */api/admin/orders* using valid customer-role tokens, expired tokens or no token. Pass criterion: no administrative handler executes.

S3, brute-force login (A3). A script sent 100 sequential POST requests to */api/auth/login* within 60 seconds from one client. Pass criterion: at most $L_{\text{write}} = 15$ requests reach the handler.

S4, collaborative consistency. Three authenticated users joined one room and performed 200 cart operations. After each operation the participants' cart snapshots were compared, and the time until the other participants received the *CART_UPDATE* event was recorded. Pass criterion: zero divergence.

VII. RESULTS AND DISCUSSION

Table VII summarises the outcomes; all four suites met their pass criteria.



TABLE VII
SUMMARY OF EVALUATION RESULTS

Suite	Trials	Observed outcome
S1 Price tampering	50	47 persisted with server-computed total; 3 rejected (unknown product); 0 influenced by client total
S2 Auth. bypass	75	75 blocked in middleware (401 or 403); 0 admin operations
S3 Login rate limit	100	15 reached handler; 85 rejected with 429, Retry-After: 60
S4 Collaborative room	200	0 divergence; median 34 ms (add) and 31 ms (quantity change)

A. Price Integrity

In all 50 tampered requests, the client-supplied total had no effect. The 47 requests with valid products were priced by the engine from database values and persisted with the server-computed total. The three requests that also named non-existent products were rejected with HTTP 400 and created no order. This confirms the integrity property of Section IV-E in practice: the injected *total* was stripped during schema parsing and could not enter Eq. (6). Table VIII reproduces one order from the evaluation, which used the code ARKA10 (10% discount, cap not reached).

TABLE VIII
WORKED EXAMPLE: SERVER-COMPUTED ORDER

Step	Amount (₹)
Subtotal, Eq. (2)	11,400
Discount, 10% of 11,400, Eq. (3)	1,140
Taxable amount, $11,400 - 1,140$	10,260
GST, $\text{round}(10,260 \times 0.18 = 1,846.8)$, Eq. (5)	1,847
Shipping (subtotal $\geq \text{₹}999$), Eq. (4)	0
Total, $10,260 + 1,847 + 0$, Eq. (6)	12,107

B. Access Control

All 75 requests in S2 were stopped by the middleware before any route handler executed. Requests without a token or with an expired token received HTTP 401; requests carrying a valid customer-role token received HTTP 403. No administrative operation was performed. Because enforcement happens once, in middleware, the result does not

depend on each administrative handler repeating the check.

C. Rate Limiting

Fig. 3 shows the outcome of S3. The first 15 requests reached the login handler and received ordinary responses (200 or 401 depending on the credentials). Requests 16 to 100 were rejected in middleware with HTTP 429 and Retry-After: 60, so the 85 excess requests placed no load on the handler or the database.

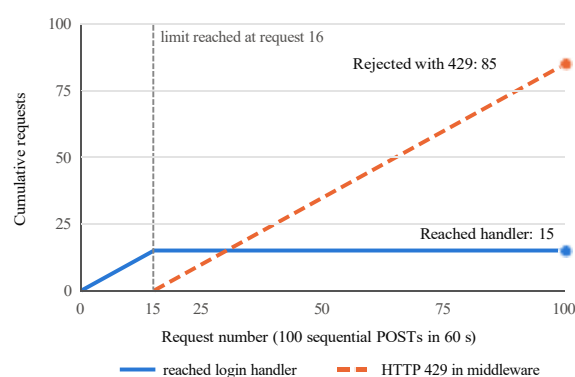


Fig. 3. Brute-force login test (S3). Cumulative requests reaching the login handler (solid) and rejected by the middleware with HTTP 429 (dashed) over 100 sequential requests from one client within 60 s; write-endpoint limit $L_{\text{write}} = 15$.

D. Collaborative Consistency

When user A added a product, users B and C received the `CART_UPDATE` event after a median of 34 ms; when user B changed a quantity, users A and C received the update after a median of 31 ms. Across the 200 operations, no divergence between the three participants' carts was observed. Because every snapshot is generated from the same server-side cart, any participant could also check out from that cart without its total differing from what the others saw.

E. Discussion

The results follow from two structural decisions rather than from defensive code in individual endpoints. Financial values are a function of server state alone, and the schema both omits and strips any client-supplied amount. Authorization is decided once, at the



edge, before any handler can run. A developer who adds a new administrative route under `/api/admin` inherits the policy of Table III without writing any check.

The cost of the pricing design is one catalogue read per order line. For large carts this can be reduced to a single query that fetches all requested products at once, without weakening the integrity property.

F. Limitations

Several limitations qualify these results. (i) The evaluation is small — four suites with 50 to 200 trials each, run by the developers against one deployment — and it did not include concurrent checkouts of the same low-stock item. To prevent overselling, the stock check and decrement must be made atomic, for example within one database transaction or as a conditional update. (ii) An exact sliding-window counter held in middleware memory is local to each running instance; in a multi-instance or multi-region deployment the counts must be kept in a shared store, or approximated as in [11]. Keying limits on IP address also groups together users behind shared carrier-grade NAT, which is common on mobile networks. (iii) Stateless JWTs cannot be revoked before they expire unless lifetimes are kept short or a revocation list is consulted. (iv) Forwarding the verified role as a request header is safe only if the middleware first discards any client-supplied header of the same name. (v) Next.js 16 deprecates the *middleware.ts* convention in favour of *proxy.ts*, which runs on the Node.js runtime; the Edge middleware used here is still supported but will need to migrate [15]. (vi) Payments use a mock flow, so payment-provider state inconsistencies of the kind studied in [1] were not evaluated.

VIII. CONCLUSION AND FUTURE WORK

This paper presented ARKA, a secure e-commerce web platform in which the client can choose what to buy but cannot influence

what it costs or which operations it may perform. A four-layer pipeline — sliding-window rate limiting and JWT/RBAC in Edge Middleware, Zod schema validation, and a server-authoritative Canonical Pricing Engine — addresses the price-tampering, access-control and brute-force threats identified in Section III, and a server-authoritative Collaborative Shopping Room lets several users build one cart in real time. In the controlled evaluation, 0 of 50 tampering attempts changed a persisted total, 0 of 75 bypass attempts reached a protected handler, 85 of 85 excess login requests were rejected at the edge, and 200 collaborative operations produced no divergence.

Future work will (1) integrate a payment aggregator authorised under the Reserve Bank of India's guidelines [20] for UPI and card settlement, and evaluate the merchant–payment-provider consistency problems identified in [1]; (2) replace the per-instance rate limiter with a shared, distributed store; (3) add atomic inventory reservation and evaluate it under concurrent load; (4) migrate request interception to the Next.js 16 *proxy* convention; and (5) add a personalised recommendation engine based on collaborative filtering over anonymised order data.

REFERENCES

- [1] R. Wang, S. Chen, X. Wang, and S. Qadeer, “How to Shop for Free Online — Security Analysis of Cashier-as-a-Service Based Web Stores,” in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2011, pp. 465–480, doi: 10.1109/SP.2011.26.
- [2] F. Sun, L. Xu, and Z. Su, “Detecting Logic Vulnerabilities in E-Commerce Applications,” in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2014.
- [3] R. Pagey, M. Mannan, and A. Youssef, “All Your Shops Are Belong to Us: Security Weaknesses in E-commerce Platforms,” in *Proc. ACM Web Conference (WWW '23)*, Austin, TX, USA, 2023, doi: 10.1145/3543507.3583319.



- [4] OWASP Foundation, "OWASP Top 10:2021," 2021. [Online]. Available: <https://owasp.org/Top10/>
- [5] OWASP Foundation, "OWASP API Security Top 10 – 2023," 2023. [Online]. Available: <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>
- [6] L. Zhu, I. Benbasat, and Z. Jiang, "Let's Shop Online Together: An Empirical Investigation of Collaborative Online Shopping Support," *Information Systems Research*, vol. 21, no. 4, pp. 872–891, Dec. 2010.
- [7] G. Pellegrino and D. Balzarotti, "Toward Black-Box Detection of Logic Flaws in Web Applications," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2014.
- [8] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transactions on Information and System Security*, vol. 4, no. 3, pp. 224–274, 2001, doi: 10.1145/501978.501980.
- [9] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015.
- [10] M. Nottingham and R. Fielding, "Additional HTTP Status Codes," IETF RFC 6585, Apr. 2012.
- [11] J. Desgats, "How we built rate limiting capable of scaling to millions of domains," *Cloudflare Blog*, Jun. 7, 2017. [Online]. Available: <https://blog.cloudflare.com/counting-things-a-lot-of-different-things/>
- [12] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," in *Proc. ACM SIGMOD International Conference on Management of Data*, 1989, pp. 399–407.
- [13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Proc. 13th Int. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 6976, 2011, pp. 386–400.
- [14] Government of India, "The Digital Personal Data Protection Act, 2023 (No. 22 of 2023)," Ministry of Law and Justice, Aug. 2023.
- [15] Vercel, "Next.js 16," *Next.js Blog*, Oct. 21, 2025. [Online]. Available: <https://nextjs.org/blog/next-16>
- [16] C. McDonnell, "Zod: TypeScript-first schema validation with static type inference." [Online]. Available: <https://zod.dev>
- [17] Prisma Data, Inc., "Prisma ORM Documentation." [Online]. Available: <https://www.prisma.io/docs>
- [18] F. Skokan, "jose: JWA, JWS, JWE, JWT, JWK, JWKS library," GitHub repository. [Online]. Available: <https://github.com/panva/jose>
- [19] I. Fette and A. Melnikov, "The WebSocket Protocol," IETF RFC 6455, Dec. 2011.
- [20] Reserve Bank of India, "Guidelines on Regulation of Payment Aggregators and Payment Gateways," Mar. 17, 2020.

ABOUT THE AUTHORS

Diksha Subhash Kumbhare holds a Bachelor of Computer Applications (BCA) from Raison College, Nagpur, and a Master of Computer Applications (MCA) from the Department of Computer Science, Sardar Patel College, Balaghat. She is a full-stack web developer with hands-on experience in Next.js, React, TypeScript, PostgreSQL, Prisma ORM, and security-first web application architecture.

Nirjala Subhash Kumbhare holds a Bachelor of Commerce (B.Com) from Bhivgade National College, Nagpur, and is currently pursuing a Master of Commerce and Management (MCM) at VMV College, Nagpur. Her research interests include e-commerce business processes, digital payment systems, online transaction security, and consumer behaviour in digital retail platforms.