International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

Shopmicro Ai microservices ecommerce platform

Garikapudi Nagalohitha
PG MCA Scholar, Department of Computer Science, Sir C R Reddy College, Eluru, Andhra Pradesh, India-534005
garikapudinagalohitha@gmail.com

Mehaboob Karishma
Assistant Professor, MCA Department, Sir C R Reddy College Autonomous PG Courses, Eluru, Andhra Pradesh, India-534005
mehaboob.karishma@gmail.com

Musunuru Ratnakar
HOD, Department of Computer Science, Sir C R Reddy College, Eluru, Andhra Pradesh, India-534005

Abstract*—The rapid expansion of online commerce requires platforms that remain responsive, secure, maintainable, and personalized as user traffic and business functions grow. This paper presents ShopMicro, an AI-enabled e-commerce platform developed with a microservices architecture. A React and TypeScript frontend communicates through a centralized API Gateway with independently deployable Authentication, Product, Order, Payment, and Inventory services implemented using Node.js and Express.js. Each service maintains an isolated PostgreSQL database, reducing coupling and improving fault isolation. The platform integrates the Groq large-language-model API as an intelligent shopping assistant and Pinecone vector search for semantic product discovery beyond exact keyword matching. JWT-based authentication, bcrypt password hashing, protected routes, and role-based access control secure customer and administrator workflows. Experimental observations reported by the project show API Gateway response times of approximately 80–150 ms, product-service responses of 100–180 ms, order processing of 350–600 ms, payment processing of 300–500 ms, inventory updates below 100 ms, and service availability of 99.8%. The results indicate that the proposed architecture provides a practical foundation for scalable, cloud-ready, and intelligent e-commerce applications while preserving modularity and independent service evolution.

Keywords—Microservices, e-commerce, API Gateway, semantic search, large language model, Pinecone, Groq AI, PostgreSQL, JWT authentication.

I. INTRODUCTION

E-commerce systems must coordinate product discovery, customer identity, shopping carts, payments, orders, and inventory while supporting high availability and frequent feature updates. In a monolithic application these functions share one deployment unit and usually one data model. As the platform grows, a local change can require complete redeployment, and a failure in one module can affect unrelated functions. Microservices address this problem by separating business capabilities into independently developed and deployable services [1]–[6].

ShopMicro applies this architectural style to a complete online-shopping workflow. The customer interface supports registration, product browsing, semantic search, cart operations, checkout, payment, and order history. An administrator interface supports product, category, order, payment, and inventory management. All requests pass through an API Gateway, which presents one client-facing endpoint while routing each request to the responsible service. The five backend services operate on separate ports and maintain isolated PostgreSQL databases.

A. Motivation and Problem Definition

Traditional product search depends on lexical overlap between the customer query and stored product text. This can fail when a user expresses intent rather than an exact product name. Static support pages also provide limited assistance during product comparison. ShopMicro adds an LLM-based shopping assistant and vector retrieval so that users can describe requirements in natural language and retrieve semantically related products. The design problem is therefore not only to implement shopping functions, but to connect AI services to a secure transactional platform without tightly coupling them to orders, payments, or inventory.

B. Contributions

The work contributes: (1) an independently deployable five-service e-commerce backend; (2) a centralized gateway for routing, authentication, and error handling; (3) database-per-service isolation; (4) Groq-powered conversational shopping assistance; (5) Pinecone-based semantic product retrieval; (6) JWT and role-based protection for customer and administrator operations; and (7) an evaluated end-to-end workflow from product discovery to payment and stock update.

II. RELATED WORK

A. Microservices and Domain Decomposition

Microservices organize software around business capabilities and lightweight communication mechanisms. Newman, Richardson, Evans, and Fowler emphasize bounded contexts, autonomous deployment, and explicit interfaces as mechanisms for reducing coordination costs in large systems [1]–[4]. Empirical studies report benefits in scalability, fault isolation, and deployment agility, while also identifying distributed-data consistency, observability, testing, and operational complexity as important trade-offs [5]–[10]. ShopMicro uses business boundaries—authentication, products, orders, payments, and inventory—rather than technical layers as service boundaries.

B. AI-Supported Product Discovery

Recommendation and semantic retrieval systems improve discovery when catalog size makes manual browsing difficult. Vector databases store dense representations of product descriptions, enabling similarity-based retrieval for natural-language queries. An LLM can then use retrieved product information as bounded context when answering user questions. This retrieval-oriented pattern limits unsupported responses because recommendations are grounded in available catalog data rather than generated without product context [11]–[14].

C. Security and API Design

Stateless JWT authentication is suitable for distributed services because identity claims can be verified at the gateway and again by protected services. Passwords are stored as

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

adaptive bcrypt hashes rather than reversible encryption. REST-oriented APIs, input validation, least-privilege roles, secure secret handling, and independent database credentials reduce the attack surface [15]–[19]. ShopMicro applies these controls to separate customer and administrator permissions.

III. MATERIALS AND METHODS

A. System Architecture

The architecture in Fig. 1 consists of a React/Vite customer interface, an administrative dashboard, an API Gateway, five backend microservices, five PostgreSQL databases, and external Groq and Pinecone services. The gateway performs request routing, token validation, error handling, and unified client communication. Each microservice owns its schema and exposes only REST endpoints, preventing direct cross-service database access.

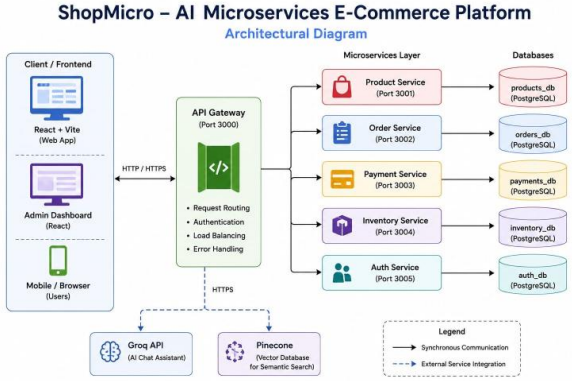

Fig. 1. ShopMicro System Architecture

This separation supports independent scaling. Product search may require more instances during browsing peaks, while order and payment services can scale according to transaction load. A failure in the AI assistant does not need to stop conventional product browsing or checkout, and a product-service update does not require redeploying authentication or payment logic.

B. Service Boundaries and Data Ownership

Table I. Core service boundaries in ShopMicro

Component	Primary responsibility
API Gateway	Request routing, JWT validation, error handling, and unified client access.
Authentication	Registration, login, bcrypt hashing, JWT issuance, and role-based access.
Product	Product/category CRUD, filtering, semantic search, and AI recommendations.
Order	Order creation, history, status updates, cancellation, and invoice workflow.
Payment	Payment processing, verification, records, refund handling, and history.
Inventory	Stock updates, low-stock monitoring, synchronization, and reporting.

Database-per-service design avoids a shared schema becoming an integration bottleneck. The order service stores order state, the payment service stores transaction state, and inventory stores stock quantities. Cross-service coordination occurs through API calls. In a production extension, asynchronous events and an outbox pattern can improve consistency for order-created, payment-confirmed, and inventory-updated events [7], [8].

C. Semantic Search and AI Assistant

Product descriptions are converted into vector embeddings and indexed in Pinecone. For a user query vector q and product vector p, semantic relevance can be expressed using cosine similarity:

$$\text{sim}(q,p) = (q \cdot p) / (||q||\,||p||) \quad (1)$$

The highest-scoring product records are supplied as context to the Groq assistant, which generates a natural-language response and recommendations. Restricting context to catalog records reduces hallucination risk and ensures that the assistant recommends products that exist in the platform. Traditional filtering remains available for category, price, and exact-name searches.

D. Authentication and Transaction Flow

During login, the authentication service verifies the bcrypt hash and issues a signed JWT containing user identity and role. The gateway validates the token before forwarding protected requests. During checkout, the order service confirms product information and quantity, the payment service records the simulated transaction, and the inventory service deducts stock after successful order completion. Administrator routes require the Admin role, while customer routes allow access only to the owner's cart and order history.

E. End-to-End Customer and Administrator Workflows

The customer workflow begins with registration or login and continues through browsing, category filtering, semantic search, product selection, cart review, checkout, payment selection, order confirmation, and order-history access. The frontend stores only presentation state and authentication information required for the current session; business decisions remain in the responsible backend service. This prevents the interface from becoming a second implementation of product, payment, or inventory rules.

The administrator workflow uses the same gateway but a different authorization policy. After validating an Admin role, the platform permits product and category creation, modification, and deletion; inventory monitoring; order-status management; payment review; and analytical dashboard access. Keeping administrative functions behind the gateway enables consistent token validation, audit logging, and error responses. It also permits future separation of administrative endpoints into a dedicated gateway or management network without rewriting each service.

F. Inter-Service Communication and Error Handling

The prototype uses synchronous HTTP/REST communication because it is easy to inspect and suitable for academic demonstration. The gateway forwards a request to one service, receives a response, and returns a normalized result to the frontend. Common error cases include expired tokens, invalid input, missing products, insufficient stock, payment failure, unavailable AI services, and database connectivity problems. The gateway should convert internal errors into stable client responses while preserving diagnostic details in server logs.

Synchronous calls simplify request tracing but can create cascading latency when one operation depends on several services. A production architecture can reduce this coupling through asynchronous domain events. For example, OrderCreated can reserve stock, PaymentConfirmed can finalize the order, and InventoryUpdated can notify the

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

customer. Dead-letter queues, retry limits, correlation identifiers, and idempotent consumers are necessary to ensure that events are processed once from a business perspective even when delivery is repeated.

G. Database Design and Consistency Management

Five PostgreSQL databases—auth_db, products_db, orders_db, payments_db, and inventory_db—enforce service ownership of data. The product database stores catalog and category data, while inventory stores quantities and stock status rather than duplicating the complete product record. Orders retain the product identifiers, quantities, totals, and status required for history. Payments retain order references, amount, method, status, and transaction details. Authentication stores identity, password hash, and role.

Because no distributed transaction spans all databases, consistency must be handled at the workflow level. The prototype performs operations in a controlled sequence and updates inventory after successful order processing. In a production environment, compensating actions are required: a failed payment should cancel or expire the order reservation, and a failed stock update should prevent final confirmation. The saga pattern, transactional outbox, and idempotency keys provide practical mechanisms for this coordination without introducing a global database transaction.

H. Retrieval Grounding and AI Response Control

The AI assistant is most useful when it is constrained by product data. The application first retrieves a bounded set of relevant catalog records and then supplies names, prices, categories, and descriptions as context to the language model. The prompt instructs the model to answer from this context and to avoid claiming availability for absent products. Responses should include product identifiers or links so that the user can verify recommendations directly in the catalog.

Operational safeguards include query-length limits, prompt-injection filtering, content moderation, rate limiting, timeout handling, and removal of secrets or internal fields from the model context. User conversations should not be retained longer than necessary, and personal data should not be sent to external AI services unless consent and data-processing requirements are satisfied. These controls separate conversational convenience from the authoritative transactional state maintained by the microservices.

IV. IMPLEMENTATION AND RESULTS

A. Technology Stack and Implementation

The frontend uses React 18, Vite, TypeScript, Tailwind CSS, shadcn/ui, React Router, and Axios. The backend uses Node.js, Express.js, and TypeScript. PostgreSQL is accessed independently by each service, while Postman supports API testing and Git/GitHub support version control. The Product, Order, Payment, Inventory, and Authentication services run on separate ports and are exposed to the client only through the API Gateway.

The implemented customer pages include Home, Shop, Registration, Login, Cart, Checkout, My Orders, and an AI chat interface. The administrator dashboard supports product and category maintenance, order and payment monitoring, inventory control, and analytical views. Fig. 2 shows the product-catalog output from the uploaded project document.

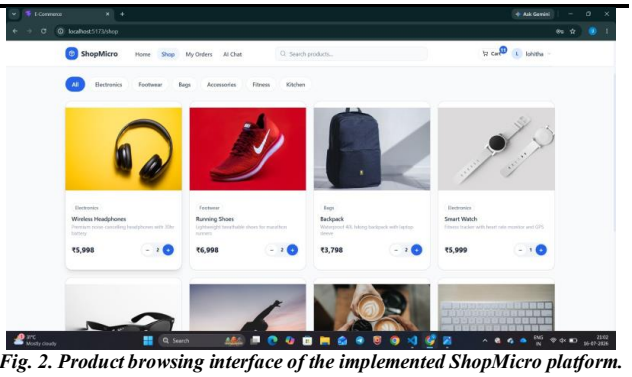

Fig. 2. Product browsing interface of the implemented ShopMicro platform.

B. Evaluation Method

The project evaluated service startup, database connectivity, gateway routing, authentication, product retrieval, semantic search, order placement, payment processing, and inventory synchronization. Test-case tables are intentionally omitted from this paper; only aggregate performance values reported in the project documentation are analyzed. For n requests, average response time is computed as

$$T\_avg = \Sigma(i=1 \text{ to } n)\ [t\_out(i) - t\_in(i)] / n \quad (2)$$

Service availability is represented as the proportion of observed uptime U to total monitored time:

$$\text{Availability} = U / (U + D) \times 100\% \quad (3)$$

where D is downtime. Request throughput can be measured as completed requests divided by the observation interval. These measures complement functional validation by describing responsiveness and operational continuity.

C. Measurement Environment and Operational Checks

The reported measurements were collected from the integrated prototype with the frontend, gateway, service processes, and PostgreSQL databases active. The evaluation verified that each service connected to its own database and that the gateway routed requests to the configured internal port. Authentication timing includes credential lookup, bcrypt comparison, JWT generation, and response serialization. Order and payment timing includes multiple validation and persistence steps, which explains their higher ranges compared with simple product retrieval.

Operational checks covered service startup, health of database connections, protected-route behavior, correct role enforcement, inventory reduction after completed orders, and continuity of independent services. Concurrent-request testing reported approximately 50 requests per service without errors. Although this load is limited compared with a commercial platform, it is sufficient to demonstrate that the architecture does not depend on a single monolithic process and that multiple service instances can be evaluated separately.

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

D. Performance Results

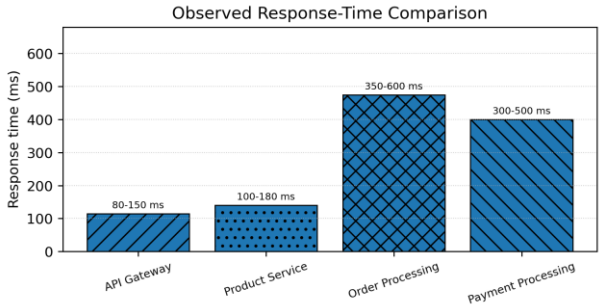

Fig. 3. Four-bar comparison using the midpoints of reported response-time ranges; exact ranges are shown above each bar.

Table II. Observed system performance metrics

Metric	Observed value
API gateway response	80-150 ms
Product retrieval/search	100-180 ms
Order processing	350-600 ms
Payment processing	300-500 ms
Inventory update	< 100 ms
JWT authentication	200-350 ms
Availability / concurrency	99.8%; about 50 requests per service

The gateway and product-service paths remain below 180 ms in the reported observations. Inventory updates complete in under 100 ms, while order and payment operations take longer because they coordinate multiple business steps and database writes. Authentication completes in approximately 200–350 ms. The reported 99.8% service availability and successful handling of about 50 concurrent requests per service indicate stable prototype operation under the evaluated load.

E. Discussion

The results satisfy the project's non-functional target that ordinary product, search, and order APIs respond within three seconds. The largest measured interval, order processing at 350–600 ms, remains substantially below that threshold. The distributed architecture also preserves failure isolation: each service has a dedicated database, and requests are routed through the gateway rather than through direct client access to internal ports.

Response-time interpretation should consider the type of operation rather than ranking every service by speed alone. Product retrieval is read-oriented and benefits from straightforward indexed queries. Order processing validates cart information and creates persistent business state, while payment processing records a financial event and may invoke additional checks. Inventory updates are small writes and therefore complete quickly in the prototype. The measurements are consistent with the relative complexity of these workflows.

The availability figure indicates that the services remained accessible for nearly all of the observed test period. It does not yet represent a long-running production service-level agreement because the evaluation window and failure-injection procedure are limited. Future evaluation should report percentile latency, error rate, throughput, CPU and memory consumption, database connection utilization, and recovery time after deliberately stopping a service or database.

The platform's AI functions improve discovery but introduce dependencies on external network services and configured API keys. Therefore, production deployment should include timeouts, circuit breakers, cached recommendations, graceful fallback to keyword search, and monitoring of LLM response quality. Transaction processing also requires stronger consistency mechanisms, idempotency keys, secure payment-gateway integration, audit logging, and retry policies before commercial use.

G. Comparative Analysis

Compared with a monolithic e-commerce application, ShopMicro introduces more deployment units and network communication, but it avoids rebuilding and restarting unrelated modules for a localized update. Product-service replicas can be increased during browsing campaigns, while payment and order services remain at transaction-oriented capacity. Separate databases reduce accidental cross-module dependencies and allow schema evolution to follow the needs of each bounded context.

Compared with exact keyword search, vector search improves retrieval for descriptive queries such as "lightweight headphones for travel" or "formal shoes within a budget," even when the wording differs from the catalog title. The AI assistant adds explanation and conversational refinement, while conventional filters remain important for deterministic constraints such as category and price. The combined approach therefore uses AI for discovery without replacing authoritative catalog and transaction logic.

F. Scalability, Maintainability, and Security

Independent deployment allows high-demand services to scale without replicating the entire application. TypeScript interfaces, service-specific repositories, and gateway contracts improve maintainability. Security is strengthened through bcrypt, JWT validation, role-based access, protected routes, and isolated credentials. Future cloud deployment should add TLS termination, secret management, container orchestration, centralized logs, distributed tracing, rate limiting, health checks, and automated database backup.

Security testing should also verify token expiration, signature validation, privilege escalation attempts, SQL injection resistance, malformed JSON handling, brute-force login controls, and unauthorized access to administrative routes. Payment endpoints require idempotency tokens so repeated client requests cannot create duplicate transactions. Audit records should capture actor, operation, target resource, timestamp, and outcome without storing raw passwords, access tokens, or sensitive payment data.

Maintainability depends on explicit API contracts and versioning. Contract tests can detect breaking changes before deployment, while service-level unit and integration tests isolate defects. Centralized observability should attach a correlation identifier at the gateway and propagate it across all calls. Metrics, structured logs, and traces allow operators to locate whether delay originates in the frontend, gateway, service logic, external AI call, or database query.

H. Limitations and Future Scope

The current implementation uses simulated payments and prototype-scale load testing. It depends on active PostgreSQL instances, a stable internet connection, and valid Groq and Pinecone credentials. Future work includes real payment gateways such as Razorpay or Stripe, event-driven communication, mobile clients, multilingual and voice

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

shopping assistance, personalized recommendations based on purchase history, cloud deployment, multi-currency support, and stronger privacy controls for user-behaviour data.

A cloud-ready release can package each service as a container and deploy it through Kubernetes or a managed container platform. Horizontal autoscaling can use request rate, CPU use, or queue length. A service mesh can provide mutual TLS and traffic policies, while managed PostgreSQL instances can provide backup, replication, and failover. Blue-green or canary deployment reduces risk when releasing a new service version.

Recommendation quality should be evaluated with catalog-specific relevance judgments, click-through rate, conversion rate, and user satisfaction rather than only response speed. Offline tests can compare keyword, vector, and hybrid retrieval, while online experiments can evaluate whether explanations and conversational refinement improve product discovery. Privacy-aware personalization should allow users to inspect or delete profile data and should minimize the retention of browsing history.

Additional business functions may include coupon management, shipping-provider integration, tax calculation, returns, seller marketplaces, fraud detection, and regional inventory. These features should be introduced as bounded services only when they have independent data and scaling needs; creating excessively small services would increase operational complexity without providing corresponding business value.

I. Production Deployment and Evaluation Roadmap

A production release should replace manual multi-terminal startup with automated build and deployment pipelines. Each service can be packaged as a container containing only its runtime, compiled TypeScript output, and declared dependencies. A continuous-integration pipeline should execute linting, unit tests, contract tests, dependency scanning, and container-image scanning before publishing a versioned image. Deployment manifests can then define replicas, resource limits, readiness probes, health checks, environment variables, and secrets for each service.

The API Gateway becomes the operational entry point and should be deployed behind a reverse proxy or managed load balancer. TLS termination, HTTP security headers, cross-origin policies, request-size limits, and rate limits should be configured centrally. Internal service endpoints should not be exposed to the public network. Database traffic should remain in a private network segment, and each service account should receive access only to its own database. Rotating JWT secrets and external API keys must not require rebuilding application images.

Resilience testing should deliberately introduce realistic failures. Useful experiments include terminating one product-service instance, disconnecting Pinecone, delaying the Groq response, exhausting a database connection pool, and restarting the payment service during a transaction. The expected result is graceful degradation rather than complete platform failure. Conventional keyword search should remain available during vector-service outages, and the shopping workflow should display a clear retry message when an external assistant is unavailable.

Performance evaluation should progress from simple response observations to repeatable workload scenarios. A workload model can include browsing-heavy traffic, promotional bursts, mixed customer and administrator operations, and checkout peaks. For each scenario, the study should report median and 95th-percentile latency, throughput, failure rate, CPU consumption, memory use, and database load. Autoscaling experiments can then determine whether additional replicas reduce latency without creating excessive database contention.

AI evaluation requires a separate methodology because fast responses do not guarantee relevant recommendations. A representative set of product-discovery queries should be judged for relevance, completeness, factual consistency, and catalog grounding. Vector-only, keyword-only, and hybrid retrieval can be compared using precision at k, recall at k, mean reciprocal rank, and user preference. Conversational answers should be checked for incorrect prices, unsupported availability claims, unsafe content, and failure to respect category or budget constraints.

The deployment roadmap should also include data-governance controls. User profiles, orders, and payment records require defined retention periods and access policies. Browsing behavior used for personalization should be collected only for a documented purpose and should be removable on request. Logs must avoid recording passwords, JWTs, complete payment data, or raw prompts containing personal information. Backups should be encrypted, tested through restoration drills, and retained according to institutional or commercial requirements.

J. Quality Attribute Assessment

ShopMicro can be assessed against core software-quality attributes. Functional suitability is demonstrated by the complete shopping and administration workflows. Performance efficiency is supported by sub-second prototype response ranges for the measured services. Compatibility is provided through REST interfaces and browser-based clients. Usability is addressed through responsive React pages and natural-language assistance. Reliability is improved through service isolation, although production-grade retry and failover mechanisms remain future work.

Security is addressed through password hashing, token-based authentication, role separation, protected routes, and data isolation. Maintainability benefits from typed code, bounded services, and independent schemas. Portability is supported by the cross-platform Node.js and PostgreSQL stack and can be strengthened through containers. Scalability is architectural rather than fully proven by the current load test; independent replication is possible, but broader benchmarking is required to determine database and external-service limits.

These quality attributes interact and sometimes conflict. Very fine service decomposition can improve deployment independence but increase latency and operational overhead. Longer AI prompts can improve answer context but increase response time and cost. Strict consistency can simplify transaction reasoning but reduce availability during partial failures. The architecture should therefore be evolved through measured trade-offs rather than by maximizing one attribute in isolation.

For academic use, the present platform provides a complete demonstration of frontend engineering, secure API design, database management, distributed services, and applied AI. For real-world use, the same design should be supported by

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

service-level objectives, incident procedures, monitoring dashboards, data-protection documentation, and a staged release process. This distinction prevents prototype success from being overstated while preserving the value of the implemented architecture.

V. CONCLUSION

ShopMicro demonstrates how microservices, artificial intelligence, semantic retrieval, and secure web engineering can be integrated into one e-commerce platform. The solution separates authentication, products, orders, payments, and inventory into independent services, each with isolated PostgreSQL storage, and coordinates them through a centralized API Gateway. Groq enables conversational shopping support, while Pinecone enables intent-oriented product discovery.

The implementation delivered complete customer and administrator workflows and reported responsive prototype performance: 80–150 ms gateway routing, 100–180 ms product retrieval, 350–600 ms order processing, 300–500 ms payment processing, sub-100 ms inventory updates, and 99.8% availability. These results show that the architecture is a practical academic prototype and a strong basis for cloud deployment. Commercial readiness will require real payment integration, event-driven consistency, production observability, resilience patterns, security hardening, and broader load evaluation.

The main lesson of the work is that AI features are most reliable when they are integrated as bounded supporting capabilities rather than embedded into every transaction. Semantic search and conversational assistance improve discovery, whereas product, order, payment, and inventory services remain the authoritative sources for business decisions. This separation permits the AI layer to evolve independently and provides a fallback path when an external AI service is unavailable.

ShopMicro therefore provides both an implementable application and a reusable architectural pattern for academic e-commerce projects. Its next stage should emphasize automated deployment, event-driven coordination, systematic security testing, realistic workload generation, and user-centered evaluation of AI recommendations. With these extensions, the prototype can progress from a functional demonstration toward a resilient production platform.

From a software-engineering perspective, the project shows that service boundaries should follow business responsibility. Authentication owns identity and access, Product owns catalog discovery, Order owns the purchase lifecycle, Payment owns transaction state, and Inventory owns stock. This ownership model makes interfaces and data responsibilities explicit, which is more valuable than dividing a system only by frontend and backend layers. It also creates a clear basis for assigning tests, monitoring, and future team ownership.

From an AI perspective, the platform demonstrates a controlled integration pattern. The language model assists users, but it does not directly modify orders, payments, or inventory. Pinecone retrieval supplies product context, and the authoritative action still occurs through validated service endpoints. This arrangement reduces the risk that an unconstrained generated response changes transactional state or presents unavailable products as confirmed purchases.

The reported measurements should be interpreted as prototype evidence rather than a commercial service guarantee. Nevertheless, they confirm that the selected stack can coordinate independent services with response times comfortably below the project targets. The architecture is therefore suitable for further experimentation with container orchestration, event streaming, observability, real payment gateways, and larger catalog workloads.

Overall, ShopMicro fulfills the objective of building a secure, modular, and intelligent shopping platform while preserving editable and extensible service boundaries.

REFERENCES

[1] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
[2] C. Richardson, Microservices Patterns. Shelter Island, NY, USA: Manning, 2018.
[3] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley, 2003.
[4] J. Lewis and M. Fowler, "Microservices: A Definition of This New Architectural Term," 2014.
[5] N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.
[6] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, no. 3, pp. 42–52, 2016.
[7] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," IEEE Cloud Computing, vol. 4, no. 5, pp. 22–32, 2017.
[8] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with Microservices: A Systematic Mapping Study," Journal of Systems and Software, vol. 150, pp. 77–97, 2019.
[9] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in Proc. CLOSER, 2016, pp. 137–146.
[10] H. Knoche and W. Hasselbring, "Drivers and Barriers for Microservice Adoption—A Survey among Professionals in Germany," Enterprise Modelling and Information Systems Architectures, vol. 14, 2019.
[11] F. Ricci, L. Rokach, and B. Shapira, Eds., Recommender Systems Handbook, 3rd ed. Springer, 2022.
[12] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535–547, 2021.
[13] Pinecone Systems, "Pinecone Vector Database Documentation," product documentation.
[14] Groq, "GroqCloud API Documentation," developer documentation.
[15] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, IETF, May 2015.
[16] N. Provos and D. Mazières, "A Future-Adaptable Password Scheme," in Proc. USENIX Annual Technical Conference, 1999.
[17] R. T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," Ph.D. dissertation, Univ. of California, Irvine, 2000.
[18] National Institute of Standards and Technology, Security Strategies for Microservices-Based Application Systems, NIST SP 800-204, 2019.
[19] National Institute of Standards and Technology, Building Secure Microservices-Based Applications Using Service-Mesh Architecture, NIST SP 800-204A, 2020.
[20] OWASP Foundation, Application Security Verification Standard 4.0.3, 2021.
[21] Meta Platforms, "React Documentation," developer documentation.
[22] OpenJS Foundation, "Node.js Documentation," developer documentation.
[23] OpenJS Foundation, "Express.js Documentation," developer documentation.
[24] Microsoft, "TypeScript Documentation," developer documentation.

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

[25] PostgreSQL Global Development Group, "PostgreSQL Documentation," database documentation.
[26] Vite Team, "Vite Documentation," frontend tooling documentation.
[27] Tailwind Labs, "Tailwind CSS Documentation," user-interface framework documentation.
[28] I. Sommerville, Software Engineering, 10th ed. Boston, MA, USA: Pearson, 2016.
[29] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill, 2019.
[30] International Organization for Standardization, ISO/IEC 25010: Systems and Software Quality Models, Geneva, Switzerland, 2011.

Received: 16-06-2026 | Accepted: 21-07-2026 | Published: 27-07-2026