ISSN: 2755-9181

Research Article

Journal of Journalism and Media Management

Mercura: Architecture and Implementation of a Local-First, AI-Assisted Merchant Operating System for Small and Medium Enterprises

Subhaditya Singh

Department of Computer Science and Engineering, College of Engineering Guindy, Anna University, Chennai, India

Corresponding author
Subhaditya Singh Department of Computer Science and Engineering, College of Engineering Guindy, Anna University, Chennai, India.

Received: October 01, 2026; **Accepted:** October 06, 2026; **Published:** October 15, 2026

ABSTRACT

Small and medium merchants commonly rely on a fragmented mix of point solutions for inventory, order processing, payments, accounting, customer records, and analytics, which complicates data consistency and limits the availability of unified, data-grounded decision support. Cloud-hosted artificial-intelligence (AI) assistants can help address the analytics gap, but they typically require external API keys, network connectivity, and the transmission of proprietary business data to third-party services, which is unattractive for smaller merchants operating with limited budgets and privacy constraints. This paper presents Mercura, an implemented, single-process, local-first merchant operating system that integrates inventory management, order and transaction processing, a local payment sandbox, accounting (profit-and-loss, cash-flow, and balance-sheet computation), deterministic fraud risk scoring, customer intelligence, geographic analytics, and an AI Merchant Copilot within one React/TypeScript and Express/TypeScript codebase backed by a local SQLite database. The system's AI layer follows a provider-abstraction pattern: a deterministic, rule-based LocalAnalyticsAI engine is grounded in the merchant's real, locally persisted operational and financial data and serves as the default and guaranteed-available provider, while an optional local large language model (LLM), accessed through Ollama, can be substituted when available and is otherwise silently bypassed through an automatic fallback. This paper documents the system architecture, formalizes the accounting, inventory-status, and AI context-construction logic actually implemented in the codebase, and reports a functional evaluation consisting of the project's automated Vitest test suite (four test files, eight tests, all passing at time of writing) together with a structured code-level audit of the fraud, accounting, and payment-sandbox modules. The evaluation establishes functional feasibility and internal consistency of the implemented workflows rather than comparative superiority over commercial systems: no user study, production deployment, or benchmarked accuracy/precision/recall figures were collected, and none are claimed. The paper explicitly distinguishes implemented functionality from simulated functionality — most notably, the payment layer is a local sandbox that never contacts a real payment network, and the fraud engine is a deterministic risk-scoring mechanism rather than a trained machine-learning classifier. The primary contribution is systems-architectural: a reproducible, engineering-level demonstration that a unified, privacy-preserving, offline-capable merchant operating system with data-grounded AI decision support can be built without a mandatory cloud AI dependency, together with an honest accounting of the approach's current limitations and the experimental work required to validate it further.

Index Terms

local-first software, AI-assisted business intelligence, merchant operating system, deterministic AI fallback, fraud risk scoring, SME digital transformation, AI copilot, data-grounded decision support, system architecture.

Introduction

Small and medium merchants typically operate with disconnected tools for inventory tracking, order and payment processing, bookkeeping, customer relationship management, and sales analytics. Each tool maintains its own copy of overlapping business facts — a product's stock level, a customer's order history, a transaction's outcome — with no shared source of truth. This fragmentation increases reconciliation effort, delays visibility into cash flow and margin, and makes it difficult to answer cross-cutting operational questions (for example, whether a given customer's recent orders were affected by a fraud hold, or whether a low-margin product line is also a low-stock line) without manually joining data across systems.

A natural response to this fragmentation is to add an AI assistant that can answer merchant questions directly. In practice, most commercially available business-AI assistants are cloud-hosted: they require an external API key, a persistent internet connection, and — most importantly — the transmission of the merchant's operational and financial data to a third-party service. For small merchants operating on thin margins, with limited legal and IT capacity to vet a third-party data-processing relationship, this dependency is a genuine adoption barrier. It also introduces a single point of failure: if the cloud AI service is unreachable, rate-limited, or discontinued, the assistant — and often the workflow built around it — stops working.

Citation: Subhaditya Singh. Mercura: Architecture and Implementation of a Local-First, AI-Assisted Merchant Operating System for Small and Medium Enterprises. J Journalism Media Manag. 2026. 2(4): 1-10. DOI: doi.org/10.61440/JJMM.2026.v2.52

Local-first software design offers an alternative framing. Kleppmann et al. define local-first software as software in which the user's own device holds the authoritative copy of the data and the application remains fully functional without a network connection [1]. Applying this principle to merchant tooling suggests an architecture in which the operational database, the financial calculations, and — where possible — the AI reasoning itself are all executed on infrastructure the merchant controls, with any external service being optional rather than mandatory. This paper presents Mercura, an implemented merchant operating system that adopts this local-first framing across its full stack. Mercura unifies inventory, orders, a payment sandbox, accounting, fraud risk scoring, customer intelligence, geographic analytics, and an AI Merchant Copilot in a single Express/TypeScript backend and React/TypeScript frontend, persisting all business state to a local SQLite database file. The AI layer follows a provider-abstraction pattern in which a deterministic, always-available local engine is the default, and an optional local LLM (accessed through Ollama) can be substituted when reachable, with automatic and silent fallback otherwise. This paper describes what was actually built, analyzes the architecture and the underlying computational logic, and reports a functional evaluation based on the system's automated test suite and a structured audit of its core modules, rather than treating Mercura as a proposal or an idealized design.

The contributions of this paper are:

- A description of a unified, local-first merchant operating system architecture that integrates operational (inventory, orders, payments) and financial (accounting) data within a single persisted state, rather than as separate applications.
- A persistent merchant data model — implemented via an in-memory service layer backed by a local SQLite durability layer — that keeps products, customers, orders, transactions, and fraud reports consistent across a restart.
- A merchant-aware AI Copilot whose context-construction logic (getStoreContextPrompt) grounds every response in the merchant's real, locally computed dashboard and profit-and-loss figures rather than a generic prompt.
- A deterministic AI fallback architecture (LocalAnalyticsAI) with an optional local-LLM provider (OllamaProvider), demonstrating a resilience pattern in which AI functionality never depends on external network or cloud availability.
- An integrated accounting engine that derives a profit-and-loss statement, cash-flow statement, and balance sheet from the same operational data used elsewhere in the system, together with an explicit accounting of which figures are transaction-derived versus model-assumed.
- A functional evaluation of the implemented system, combining automated test execution (Vitest) with a source-level audit of the fraud-scoring, accounting, and payment-sandbox logic, and an explicit statement of what remains unvalidated.

Related Work

This section situates Mercura relative to five overlapping research and engineering threads: AI-assisted business intelligence, merchant/ERP-style business operating systems, local-first and privacy-preserving computing, AI decision-support systems (including LLM grounding), and fraud/transaction-risk detection.

AI-Assisted Business Intelligence

Business intelligence (BI) systems that incorporate AI-driven analytics have been widely studied as a means of improving SME decision-making and competitive positioning . Related work has proposed integrated data-analytics, BI, and machine-learning architectures aimed specifically at SME-scale organizations , and has examined the organizational and technical barriers — cost, complexity, and perceived value — that have historically limited BI adoption among smaller firms [2-8]. This literature generally treats AI-assisted BI as a layer added on top of existing operational systems; Mercura differs by treating BI as inseparable from the operational data model itself, since its dashboard, P&L, and Copilot all read from the same in-memory business state.

Merchant Management and ERP/Business Operating Systems

Enterprise Resource Planning (ERP) systems have long targeted the same fragmentation problem addressed here, unifying finance, inventory, and operations under one system of record. Literature reviews on ERP adoption in SMEs consistently report that limited resources, integration complexity, and implementation cost are the primary barriers to adoption for smaller organizations, a finding echoed in studies of cloud/SaaS-ERP adoption specifically and in case-study accounts of ERP adoption during periods of organizational stress [9,10,11]. Mercura's architecture reflects a response to this literature: it adopts an ERP-style unified data model but avoids the deployment and licensing complexity of a conventional multi-tier ERP installation by running as a single local process with an embedded database.

Table 1: System Technology Stack

Layer	Technology
Frontend	React 19, TypeScript, Vite 6, Tailwind CSS, Recharts, Lucide
Backend	Node.js (≥22.5.0), Express 4.21, TypeScript
Database	SQLite via node:sqlite (built-in), WAL journal mode
AI	LocalAnalyticsAI (deterministic, default); optional OllamaProvider (local LLM)
Auth	bcryptjs password hashing; jsonwebtoken (JWT) sessions
Testing	Vitest 4
Build	Vite (frontend) + esbuild (server bundle)

Local-First and Privacy-Preserving Applications

The local-first software paradigm, introduced by Kleppmann et al. proposes seven ideals for applications that keep the user's data under the user's control while retaining collaboration and availability benefits, including full functionality without a network connection and long-term data ownership independent of any vendor's continued operation. Follow-up work has examined the practical trade-offs of the approach, including consistency and tooling maturity . Complementary to this, the growing body of work on edge and on-device large-language-model (LLM) inference documents both the privacy and latency motivations for local AI execution and the resource constraints that local hardware imposes on model choice and inference speed [12,14]. Mercura's AI architecture is directly informed by

this second thread: its optional Ollama-backed LLM provider is one instance of the general local/edge-LLM deployment pattern these surveys describe, and its default reliance on a deterministic (non-LLM) engine reflects the resource and availability constraints such surveys identify as central challenges for local LLM deployment.

AI Decision-Support Systems

AI-based decision-support systems (DSS) have been reviewed extensively in industrial and enterprise contexts, with an emphasis on how AI techniques are integrated into operational data pipelines to support planning and control decisions . A separate and directly relevant line of work addresses grounding: Retrieval-Augmented Generation (RAG) techniques integrate large language models with external, verifiable, and up-to-date data sources to reduce hallucination and improve the factual reliability of generated responses [2,5]. Mercura's AI Copilot implements a context-grounding pattern in this spirit — every provider response is generated from a text context string (getStoreContextPrompt) that is freshly computed from the merchant's live dashboard statistics and profit-and-loss figures — though, unlike a full RAG pipeline, Mercura's grounding is implemented as direct structured-context injection rather than a retriever-generator architecture over an indexed corpus.

Fraud Detection and Transaction Risk Systems

Machine-learning approaches to transaction fraud detection are a mature research area, with systematic reviews cataloguing ensemble, deep-learning, and hybrid classifiers evaluated against benchmark datasets such as the MLG-ULB credit-card fraud dataset [3,4]. These reviews consistently report that supervised ensemble and deep-sequence models can achieve high benchmark accuracy on curated datasets, while also noting persistent challenges around class imbalance, concept drift, and interpretability. Mercura's fraud module does not implement a trained classifier of this kind. It instead implements a deterministic risk-scoring and indicator-derivation mechanism that assigns a risk level and a recommended action (APPROVE/REVIEW/BLOCK) to each transaction; this positions Mercura's fraud engine as an example of the rule-based baseline against which the ML literature typically compares learned models, rather than as a contribution to that literature.

Research Gap

Individually, the threads above are well studied: BI/DSS for SMEs, ERP adoption, local-first and edge-AI computing, LLM grounding, and ML-based fraud detection. What is comparatively underexplored — and what this paper's architecture is positioned against — is the intersection of these threads in a single, small-scale, reproducible system: a merchant platform in which (i) operational and financial data live in one persisted model rather than being integrated after the fact across separate systems, as ERP-adoption studies suggest is often costly to achieve (ii) the AI decision-support layer is grounded in that same local data by construction rather than as a bolt-on integration and (iii) AI availability does not depend on a mandatory cloud service, consistent with local-first principles and the practical motivations documented in the edge-LLM literature [2,5,9-14].

Mercura addresses this specific architectural gap through an engineering integration rather than a new algorithm: its novelty lies in combining an in-memory unified data model, a durable local persistence layer, and a provider-abstracted AI layer with a guaranteed deterministic fallback, within one deployable codebase. The deterministic risk-scoring engine, the assumption-driven accounting model, and the seeded demonstration dataset are implementation choices appropriate to a portfolio-scale system rather than novel research contributions in themselves, and are treated as such throughout this paper. Trained, data-driven fraud and forecasting models, and a validated benchmark against commercial or cloud-based BI/AI assistants, remain future work (Section VIII).

Table 2: Major Functional Modules

Module	Primary Function
Dashboard	Aggregate revenue/profit/order KPIs
Inventory	SKU, cost/margin, stock status, restocking
Orders	Order lifecycle: create/complete/cancel/refund
Payments	Local sandbox transaction simulation
Accounting	P&L, cash flow, balance sheet
Fraud	Deterministic risk scoring and review
Customers	Order history, lifetime value, segmentation
Geographic Hubs	SVG-based settlement/growth visualization
AI Copilot	Data-grounded conversational Q&A
Advisor	Categorized business recommendations
Reports / Settings	Summary export; configuration

System Architecture and Proposed Framework

Mercura follows a conventional layered web-application architecture, chosen deliberately for simplicity, auditability, and ease of local deployment over microservice decomposition.

Overall System Architecture

The system consists of four layers: (1) a React/TypeScript single-page frontend; (2) a typed REST API client (src/services/api.ts); (3) a single Express/TypeScript backend process (server/routes.ts) exposing the REST API; and (4) a service layer (server/services/) comprising the Database Service, Accounting Service, AI Service, and a SQLite-backed Persistence Service. In production, the same Express process serves both the REST API and the built frontend assets, so the entire application runs as one local process with no required external service in the request path.

Table 3: Core Merchant Data Entities

Entity	Representative Fields
Product	sku, cost Price, selling Price, profit Margin, current Stock, status
Customer	Lifetime Value, average Order Value, segment, risk Score
Order	items[], subtotal, tax, discount, total, status
Transaction	amount, payment Gateway, payment Method, fraud Score, risk Level
Fraud Report	Fraud Score, indicators{}, recommended Action, status

Frontend Architecture

The frontend is implemented in React 19 with TypeScript and built with Vite. It is organized as a set of feature modules — Dashboard, Accounting Module, Advisor Module, Analytics Module, Copilot Module, Customers Module, Fraud Module, Geographic Hubs Module, Inventory Module, Orders Module, Payments Module, Products Module, Reports Module, and Settings Module — each corresponding to one operational domain of the merchant business. Data visualization uses Recharts; iconography uses Lucide; styling uses Tailwind CSS. The Geographic Hubs Module renders merchant settlement hubs and AI-predicted growth nodes using a local SVG equirectangular projection, avoiding a dependency on a third-party mapping API.

Backend Architecture

The backend is a single Express application written in TypeScript, exposing a REST API under /api through an api Router. A JWT-verification middleware protects all routes other than login, token verification, and health checks. The routes observed in the implementation are summarized in Table IV and cover authentication, dashboard statistics, product/inventory management, customer and order retrieval, transaction and fraud review, the payment sandbox, accounting statements, and the three AI endpoints (copilot chat, advisor insights, geographic growth predictions).

Data Persistence Architecture

Mercura's Database Service keeps five core collections — products, customers, orders, transactions, and fraud Reports— as in-memory TypeScript arrays, which keeps the analytics and aggregation logic (dashboard statistics, accounting, fraud summaries) synchronous and simple to read. A separate Persistence Service makes this state durable: each collection is serialized to JSON and stored as a single row in a collections table inside a local SQLite database file (data/mercura.db), accessed through Node's built-in node:sqlite module (available from Node 22.5+), which avoids any native-addon build step during installation. Every mutating operation calls db.persist(), writing the current in-memory state back to disk, so business data survives a server restart. User accounts, a bcrypt password hash, and an audit log are stored separately in conventional relational tables (users, audit_logs) since they do not require the same array-oriented access pattern. On boot, Persistence Service checks whether the database already contains data; if so, it is loaded back into memory, and if not, a deterministic seed routine populates 500 products, 200 customers, 500 orders with corresponding transactions, and a derived set of fraud reports.

AI Provider Architecture

server/services/ai.ts defines an AI Provider interface (name, is Available(), generate Copilot Reply()) with two concrete implementations. LocalAnalyticsAI is a deterministic, rule-based engine that always reports itself as available and answers merchant questions by pattern-matching the query text (e.g., for keywords such as revenue, profit, stock, customer, or fraud) and formatting the corresponding figures already computed by Database Service and Accounting Service. OllamaProvider is an optional provider that, when AI_PROVIDER=ollama is set and a local Ollama server responds to a short-timeout health check, forwards the same data-grounded context string to a locally running LLM's /api/generate endpoint. An AI Service class selects a provider at request time — preferring Ollama when available and otherwise using the local engine — and additionally wraps every provider call in a try/catch that falls back to the deterministic engine on any runtime error, so a Copilot request can never fail purely due to AI unavailability.

Authentication and Security Architecture

Login accepts an email and password, looks up a bcrypt password hash via Persistence Service, and, on a match, issues a JWT signed with a configurable JWT_SECRET (24-hour expiry). All protected routes verify this token through Express middleware. The demonstration account (admin@mercura.com) is bootstrapped on first run with a bcrypt hash rather than a stored plaintext password. The project's SECURITY.md explicitly states that the application has not undergone a formal security audit and is not intended for production deployment as-is; this paper preserves that characterization rather than presenting the system as production-hardened.

Merchant Data Flow

A representative flow — completing a sandboxed sale — illustrates the integration across modules. A payment request reaches Database Service. createSandboxTransaction(), which (i) reduces the relevant product's current Stock and recomputes its status (ACTIVE/LOW_STOCK/OUT_OF_STOCK);
(ii) creates a Transaction record with a rolled fraud score; (iii) creates a corresponding Order when the payment is not marked as failed; (iv) updates the customer's lifetime Value;(v) creates a Fraud Report when the fraud score exceeds a threshold; and (vi) calls persist() to write the updated state to SQLite. Because Accounting Service. Get PL Statement() derives its figures from Database Service. Get Dashboard Stats() at call time, and getStoreContextPrompt() in turn calls both of these, a single sandbox transaction is immediately reflected — without any additional synchronization step — in the dashboard, the accounting statements, and the next AI Copilot response.

Implementation Methodology

This section describes the implemented logic of each functional module, distinguishing operational logic that is fully data-driven from logic that relies on fixed modeling assumptions.

Merchant Data Model

The core entities — Product, Customer, Order, Transaction, and Fraud Report — are defined in src/ types. its and instantiated consistently across the seed routine, the sandbox-transaction handler, and the accounting/analytics services. Products carry SKU, barcode, cost price, selling price, profit margin, current/reserved stock, supplier, and status; Customers carry lifetime value, average order value, a risk score, and a segment (VIP/Regular/New/At_Risk); Orders reference a customer and a list of line items with subtotal, tax, discount, and total; Transactions reference an order and carry a payment gateway/method, status, fraud score, and risk level; Fraud Reports reference a transaction and carry a risk-indicator breakdown and a recommended action.

Inventory Management
Each product's status is derived deterministically from its current stock level (Eq. 4). Restocking (restock Product) increments current Stock, recomputes status, updates updated At, and persists the change. Every sandbox purchase decrements the relevant product's stock and recomputes its status as part of the same transaction, keeping inventory and sales consistent without a separate reconciliation step.

Order and Transaction Processing
Orders are created only for successful (non-failed) sandbox payments, with subtotal, an 8% tax calculation, and a total computed from the requested line items at each product's current selling price. Each order has a corresponding Transaction record; a failed-payment simulation produces a Transaction with status FAILED and no corresponding Order, modeling a declined-payment path distinct from a completed sale.

Payment Sandbox
createSandboxTransaction() implements a local payment simulator. It generates transaction identifiers in the form PAY-##### and gateway transaction identifiers in the form txn sb_######, and accepts a payment gateway label (Stripe, Razor pay, PayPal) and a payment method label (Card, UPI, Pay pal). Per the project's own documentation, these gateway/method labels are simulated rail names used for realism only; no real payment network, card processor, or banking API is ever contacted, and no payment credentials are collected or stored. This paper follows that characterization throughout and refers to this component exclusively as the local payment sandbox.

Accounting Engine
Accounting Service derives three statements from the same dashboard statistics used elsewhere in the system. The profit-and-loss (P&L) statement (Section IV-K and Eqs. 5–8) computes gross sales, discounts, returns, and net sales, then applies a fixed 42% cost-of-goods-sold ratio and a set of fixed or percentage-based operating-expense assumptions (payment-processing fees at 2.8% of net sales, hosting at 5%, marketing at 12%, and fixed rent/salary figures) to derive operating income, a 21% corporate tax provision, and net income. The cash-flow statement layers fixed assumptions for depreciation, inventory/receivables/payables changes, equipment purchases, and financing activity on top of net income. The balance sheet uses several additional fixed figures (accounts receivable, inventory valuation, equipment, bank loans, common stock) and a plug value for retained earnings so that total assets equal total liabilities plus equity by construction. Only the revenue-side figures (gross sales, and by extension net sales and gross profit) are derived from actual seeded/simulated transaction data; the expense ratios, most balance-sheet line items, and the retained-earnings plug are fixed modeling assumptions rather than measured figures, and this paper does not present them as audited financial statements. Fraud Detection Mercura's fraud engine is a deterministic risk-scoring mechanism, not a trained classifier. During seeding, each transaction's fraud Score is computed from a fixed arithmetic function of its index ((i * 19) % 100), and a risk Level (LOW/MEDIUM/HIGH) is derived from fixed thresholds on that score (Eq. 9). Transactions created through the payment sandbox instead roll fraud Score from Math. random(), biased toward the 75–100 range when a caller explicitly requests a simulate Fraud flag and toward 0–20 otherwise. In both paths, individual risk-indicator fields (device Risk, customer Risk, location Risk, velocity Risk) and a natural-lang uage explanation are then derived from the already-computed fraud Score, rather than the score being computed as a function of independently measured indicators. Transactions with a fraud score above 65 generate a Fraud Report with a recommended Action of REVIEW (or BLOCK above 85) and an initial status of UNDER_REVIEW (or REJECTED for BLOCK), which the merchant can act on through the /fraud/review endpoint. This paper therefore describes the module throughout as a deterministic risk-scoring engine rather than an AI or machine-learning fraud classifier.

Customer Intelligence
Customer records carry a lifetime value, an average order value, a segment classification (VIP/Regular/New/At_Risk), and a risk score, all exposed through the Customers module and referenced by the AI Copilot's customer-related responses. Lifetime value is updated incrementally as new sandbox orders are completed for a given customer.

AI Merchant Copilot
The Copilot's context-construction function, get Store Context Prompt(), assembles a text block containing current revenue, gross profit and margin, cash flow, completed-order count, unique-customer count, payment success rate, a list of low-stock and out-of-stock product names, and headline P&L figures (gross sales, COGS, operating income, net income), all computed at request time. This context string is passed to whichever provider is selected. Under LocalAnalyticsAI, the same context values are also directly interpolated into template responses selected by keyword matching on the merchant's question (revenue/sales, profit/margin, stock/inventory, customer/retention, fraud/risk, or a fallback menu). Under OllamaProvider, the context string is prepended to the merchant's question as a prompt for the local LLM to reason over. In both cases, the Copilot's answer is grounded in the same live, locally computed data rather than a static or hypothetical dataset.

Business Advisor
Generate Advisor Insights() produces a fixed set of five categorized recommendations (PRICING, INVENTORY, RETENTION, REVENUE, MARKETING), each with a category, title, description, impact rating, and an actionable step. Two of the five (pricing and inventory) interpolate live figures — current gross margin and the count of low-stock products — into otherwise template text; the remaining three are static template recommendations not conditioned on the merchant's specific data. This paper accordingly characterizes the Business Advisor as a template-based recommendation generator with partial data conditioning, not a learned recommender system.

Geographic Analytics
Generate Growth Predictions() returns a fixed, hand-authored set of four illustrative city-level growth/liquidity predictions (São Paulo, Tokyo, Sydney, Nairobi) with example growth-rate and demand-index figures. These values are not computed from the merchant's transaction data or from any external market dataset; they are demonstration content illustrating how the Geographic

Hubs Module would render AI-predicted growth nodes once a real predictive model or external data source is connected. This paper treats geographic growth prediction as an illustrative UI capability rather than a validated forecasting feature.

Reporting and Analytics

The Reports and Analytics modules read from the same dashboard-statistics and accounting endpoints described above, and additionally expose a /reports/generate endpoint that packages current statistics into a downloadable summary. As with the P&L and cash-flow statements, any report content ultimately traces back either to actual seeded/simulated transaction data or to the fixed accounting assumptions described in Section IV-E.

Mathematical Formalization

This subsection formalizes computations actually present in the implementation (Sections IV-B, IV-E, IV-F).

Margin = (Selling Price – Cost Price) / Selling Price (1)

AOV = Total Revenue / Completed Orders (2)

CLV_ avg = Revenue / CustomersCount (3)

Inventory status is a deterministic function of current stock s:

status(s) = OUT_OF_STOCK if s = 0; LOW_STOCK if 0 < s < 15; ACTIVE if s ≥ 15 (4)

Profit-and-loss computation (Section IV-E):

Net Sales = Gross Sales − Discounts − Returns (5)

Gross Profit = Net Sales − COGS, COGS = 0.42 × Net Sales (6)

Operating Income = Gross Profit − Σ Expenses (7)

Net Income = Operating Income − (0.21 × Operating Income, if Operating Income > 0, else 0) (8)

Fraud risk level as implemented (seed path; Section IV-F): risk Level(f) = LOW if f < 40; MEDIUM if 40 ≤ f < 75; HIGH if f ≥ 75 (9)

AI decision pipeline (Section III-E, IV-H):

Merchant Query → getStoreContextPrompt() [live dashboard + P&L data] → select Provider() [Ollama if available, else Local] → generate Copilot Reply() → (on error) fallback to generate Simulated Copilot Response () → Merchant Response.

Experimental / Functional Evaluation

Because Mercura is an implemented system rather than a research prototype awaiting evaluation, this section reports a functional evaluation: what automated tests actually verify, and what a source-level audit of each workflow demonstrates. No accuracy, precision, recall, latency, or throughput measurements were collected, and none are reported. Where a specific evaluation goal was not measurable from the current implementation, this is stated explicitly rather than approximated.

Development Environment

The system targets Node.js ≥22.5.0 (required for the built-in node:sqlite module), with React 19, TypeScript ~5.8, Express 4.21, Vite 6, and Vitest 4 as the primary framework/tooling versions declared in package. Json. The automated test suite was executed for this paper using npx vitest run against the uploaded project source.

Functional Testing

The repository includes four Vitest test files totaling eight test cases: tests/accounting. test. its (2 tests: internal consistency of the P&L statement, and that dashboard statistics reflect the seeded database), tests/atheist's (2 tests: the demo account bootstraps with a crypt-hashed rather than plaintext password, and an incorrect password is rejected), tests/inventory. test. its (2 tests: restocking increases stock and updates status correctly, and restocking a non-existent product returns undefined), and tests/payments. test. its (2 tests: a sandbox transaction completes an order and reduces inventory, and a payment marked to fail produces a FAILED transaction). All eight tests passed on execution (Table VI).

Table 4: Rest Api Functional Groups

Functional Group	Representative Endpoints
Auth	/auth/login, /auth/verify
Dashboard	/stats
Inventory	/products, /products/restock
Customers / Orders	/customers, /orders
Payments / Fraud	/transactions, /payment/create, /fraud, / fraud/review
Accounting	/accounting
AI	/ai/copilot, /ai/advisor, /geography/ predictions
Reports	/reports/generate

Transaction Workflow Evaluation

Verified by automated test: a sandbox payment reduces the purchased product's stock by the requested quantity and produces a transaction with status SUCCESS. Verified by automated test: a payment with fail Payment set produces a transaction with status FAILED. Demonstrated by implementation (source-level audit, not covered by an automated test): the accompanying order is created only on non-failed payments, and a fraud report is generated when the rolled fraud score exceeds 65.

Inventory and Order Consistency

Verified by automated test: restock Product increases current Stock by the requested amount and yields a status of ACTIVE or LOW_STOCK depending on the resulting stock level; a restock request for a non-existent product ID returns undefined rather than throwing. Demonstrated by implementation: the same status-derivation logic executes on every stock-reducing sandbox purchase, keeping the two code paths (restock and sale) consistent by sharing the same threshold logic.

Accounting Consistency

Verified by automated test: net sales equals gross sales minus discounts minus returns to within floating-point tolerance, gross profit equals net sales minus total COGS, and net income does not exceed operating income. Not experimentally validated: the accuracy of the fixed expense ratios and balance-sheet assumptions (Section IV-E) against any real merchant's actual costs — these are modeling assumptions, not measured or fitted parameters, and the tests confirm only that the statement is internally arithmetically consistent, not that it reflects real-world merchant economics.

Fraud Detection Workflow

Not covered by an automated test in the current suite. Demonstrated by implementation (source-level audit): transactions with a fraud score above 65 generate a Fraud Report

with status UNDER_REVIEW (or REJECTED above 85); the /fraud/review endpoint allows a merchant to update a report's status. Not experimentally validated: any measure of detection accuracy, since the scoring function is deterministic/arithmetic (seed path) or randomized (sandbox path) rather than fitted to labeled fraud outcomes — there is no ground truth against which precision or recall could currently be computed.

Table 5: AI Provider Architecture

Property	LocalAnalyticsAI (default)	OllamaProvider (optional)
Availability	Always (is Available() → true)	Only if AI_ PROVIDER=ollama and server reachable
Network required	No	No (local network call to 127.0.0.1)
Reasoning style	Deterministic keyword-matched templates over live data	Local LLM generation over the same context string
Fallback target	N/A (is itself the fallback)	LocalAnalyticsAI, on unavailability or any error

AI Copilot Evaluation

Not covered by an automated test in the current suite. Demonstrated by implementation (source-level audit): getStoreContextPrompt() correctly interpolates live dashboard and P&L values into the context string passed to every provider, and AI Service. Get Copilot Response () falls back to the deterministic engine both when Ollama is unavailable (via is Available ()) and when a provider call throws at runtime (via the surrounding try/catch). Not experimentally validated: response quality, helpfulness, or factual accuracy of Copilot answers under the optional Ollama/LLM provider — no evaluation harness for natural-language response quality exists in the current implementation.

Persistence and Recovery

Demonstrated by implementation and confirmed during test execution: on a second run against an existing data/mercura.db file, Persistence Service loads the previously persisted product, customer, and order collections rather than re-seeding, and the running test suite's own state changes (e.g., additional orders created by the payment tests) were visible as increased order counts in the log output of subsequently executed test files within the same run. Not experimentally validated: behavior under concurrent multi-process access to the same SQLite file, or recovery from a corrupted database file.

Table 6: Functional Test Coverage (Executed via npx vitest run)

Test File	Tests	Result	What Is Verified
Accounting. test.ts	2	Pass	P&L internal consistency; stats reflect seeded DB
auth.test.ts	2	Pass	Bcrypt-hashed demo password; wrong password rejected
inventory. test.ts	2	Pass	Restock increases stock/ updates status; missing product handled
payments. test.ts	2	Pass	Sandbox purchase reduces stock/creates order; failed payment marked FAILED

System Integration Evaluation

The data-flow trace in Section III-G was validated at the source-code level by following the call graph from createSandboxTransaction() through to persist(), get Dashboard Stats(), get PL Statement(), and getStoreContextPrompt(), confirming that a single sandbox affects inventory, accounting, and Copilot context without requiring a separate synchronization step. This confirms architectural integration; it does not constitute a load or stress test of the integrated system, which was not performed.

Results and Discussion

This section discusses what the implementation and evaluation in Sections III–V establish, organized by capability area, and is explicit about the boundary between functional feasibility and comparative or quantitative claims.

Functional Capabilities

The implemented system provides a working, internally consistent set of merchant operations — inventory, order/transaction processing, a payment sandbox, accounting statements, fraud review, customer records, and an AI Copilot — accessible through a single REST API and a single React frontend. The automated test suite and source-level audit in Section V support the claim that these modules function correctly relative to their own internal logic (e.g., stock decremented correctly, P&L arithmetic consistent, failed payments correctly excluded from order creation).

Integrated Merchant Workflow

Because all modules read from one in-memory data model rather than from separate per-module stores, a single sandbox transaction is immediately visible across inventory, accounting, and the AI Copilot without an explicit synchronization or ETL step. This is the direct architectural consequence of the unified data model described in Section III-D and is the paper's clearest evidence for the "unified merchant operations architecture" framing identified as the best-supported research framing in Part 2 of the underlying project brief.

AI-Assisted Decision Support

The Copilot and Business Advisor demonstrate a working pattern of grounding AI responses in real, locally computed merchant data (Section IV-H), consistent with the broader grounding motivation documented in the RAG literature [2], though implemented here as direct structured-context injection rather than a retrieval pipeline over an indexed corpus. The current evaluation establishes functional feasibility rather than comparative superiority: no measurement of response usefulness, correctness under ambiguous queries, or comparison against a cloud LLM assistant was performed.

Local-First Architecture

The system runs as a single local process with a local SQLite database and functions with no external network dependency in its default configuration (LocalAnalyticsAI provider, local payment sandbox, local SVG maps). This is consistent with the local-first ideals described by Kleppmann et al. [1], in particular full application functionality offline and user ownership of the data file. The optional Ollama integration is the one component that can introduce a local (not cloud) network dependency, and its own dependency is bounded by a short-timeout availability check and an automatic fallback.

Reliability and Fallback Behavior

The AI provider-selection and fallback logic (Section III-E) is the paper's most concrete demonstration of designed resilience: an Ollama outage, a malformed LLM response, or any other runtime error during provider invocation is caught and transparently routed to the deterministic engine, so the Copilot endpoint cannot fail purely due to AI-provider unavailability. This fallback path was confirmed by source-level audit; it was not exercised under a live, unreachable Ollama instance as part of this paper's evaluation.

Table 7: Security and Privacy Characteristics

Characteristic	Implementation
Password storage	bcrypt hash (never plaintext) — confirmed by automated test
Session mechanism	Signed JWT, 24-hour expiry, verified via middleware on protected routes
Database exposure	SQLite file not exposed via any HTTP route; excluded from version control
Mandatory secrets	None required for core functionality
Security audit status	Not performed; not represented as production-ready (per project SECURITY.md)

Security Characteristics

Passwords are stored as bcrypt hashes rather than plaintext (confirmed by an automated test), and API routes other than login/verify/health are protected by JWT verification middleware. The SQLite database file is not exposed via any HTTP route. These are reasonable baseline practices for a local application but, as the project's own SECURITY.md states and this paper reiterates, do not constitute a completed security audit or a production-readiness claim.

Performance / Scalability Considerations

Not evaluated in the current implementation. The seeded dataset (500 products, 200 customers, 500 orders) is intentionally small for fast local demonstration; dashboard-level aggregate figures are deliberately scaled by fixed multipliers (e.g., a revenue scale factor of 18.52 and an orders-count constant of 100,000) purely for presentation realism and are clearly separable in the source code from the underlying real seeded figures. No load testing, response-time benchmarking, or scalability analysis against a larger dataset was performed, and this paper does not claim the in-memory-array architecture would scale to a materially larger catalog or transaction volume without further engineering.

Limitations and Threats to Validit

The following limitations were identified by direct inspection of the implementation and its test coverage, rather than assumed generically:

- Payments are a local sandbox rather than an integration with a live payment network; gateway/method labels are simulated rail names for realism only.
- The fraud module is a deterministic risk-scoring mechanism, not a validated, trained machine-learning classifier; risk indicators are derived from the fraud score rather than the score being computed from independently measured indicators.
- The AI Copilot's default provider is rule-based pattern matching over a fixed set of question categories; the optional local-LLM provider has not been benchmarked for response quality or factual accuracy.
- Accounting figures beyond gross/net sales rely on fixed modeling assumptions (expense ratios, balance-sheet line items, a retained-earnings plug) rather than measured merchant costs, and should not be read as audited financial statements.
- The demonstration dataset is synthetically seeded and, at the dashboard-statistics layer, additionally scaled by fixed multipliers for presentation purposes; results are not derived from a real merchant's transaction history.
- No large-scale deployment, load test, or scalability evaluation has been performed against the in-memory-array data model.
- No merchant user study, usability evaluation, or comparative benchmark against commercial BI/ERP or cloud-AI assistants has been conducted.
- No formal security audit or penetration test has been performed; the project documentation explicitly disclaims production readiness.
- The geographic growth-prediction feature returns fixed, hand-authored illustrative content rather than output from a trained forecasting model or a live external dataset.
- The automated test suite (eight tests across four files) covers accounting arithmetic, authentication, inventory restocking, and the payment-sandbox success/failure paths, but does not currently cover the fraud-report generation path, the AI Copilot/provider-fallback path, or the Business Advisor logic.

Conclusion and Future Work

This paper has presented Mercura, an implemented, local-first merchant operating system that unifies inventory, order/payment processing, accounting, fraud review, customer intelligence, and an AI Copilot within a single React/TypeScript and Express/TypeScript application backed by a local SQLite database. The system's principal contribution is architectural: a working demonstration that a merchant-facing AI decision-support layer can be grounded in a unified local data model and remain fully functional without a mandatory cloud AI dependency, through a deterministic-engine-with-optional-LLM provider pattern. A functional evaluation combining automated test execution (eight of eight tests passing across accounting, authentication, inventory, and payment-sandbox modules) with a source-level audit of the fraud, accounting, and AI context-construction logic supports the internal consistency of the implemented workflows.

This paper has deliberately not claimed production deployment, measured accuracy or precision/recall figures, comparative superiority over commercial systems, or validated financial/fraud outcomes, none of which were established by the current implementation or evaluation.

Future work identified directly by the limitations in Section VII includes: (i) training and evaluating a data-driven fraud classifier against labeled transaction outcomes, to be compared with the current deterministic baseline; (ii) benchmarking the optional local-LLM Copilot provider for response quality, latency, and resource consumption across representative merchant queries; (iii) replacing the fixed accounting-assumption parameters with configurable, merchant-specific cost inputs and validating the resulting statements against real bookkeeping records; (iv) conducting a merchant usability study; (v) extending the automated test suite to cover the fraud-report and AI-provider-fallback code paths; (vi) performing a formal security review and, if pursued toward production use, implementing role-based access control; (vii) replacing the illustrative geographic growth predictions with a model trained on real settlement or demand data; and (viii) evaluating the in-memory data model's behavior and required re-architecture, if any, at transaction volumes materially larger than the current seeded dataset.

Recommended Architecture Figures

The following architecture/workflow figures are recommended to accompany this paper (diagrams, not yet rendered):

- Fig. 1. Overall Mercura system architecture (React/TypeScript UI → REST API client → Express/TypeScript API → service layer → SQLite).
- Fig. 2. Merchant data-flow architecture showing shared access to the in-memory data model by the dashboard, accounting, and AI Copilot.
- Fig. 3. AI Copilot provider-selection and fallback workflow (Ollama availability check → LocalAnalyticsAI fallback → context-grounded response).
- Fig. 4. Order-to-payment-to-inventory transaction workflow through createSandboxTransaction().
- Fig. 5. Local SQLite persistence architecture (in-memory collections ↔ collections table; users/audit_logs as relational tables).

Illustrative Data Visualizations

The following figures accompany the architectural discussion above. Figs. 6–7 and 9 are illustrative visualizations of logic that is genuinely implemented in the codebase (the seeded demand profile underlying Section III-D, the P&L decomposition formalized in Eqs. 6–8, the inventory-status thresholding of Eq. 4, and the executed test results of Table(VI) and are labeled accordingly. Figs. 8 and 10 present conceptual illustrations only: no cross-platform latency benchmark and no combined device/velocity fraud-scoring surface were actually measured or computed by the implementation (Section IV-F and Section V-G explicitly state these were not evaluated), so these two figures are captioned as conceptual rather than experimental to avoid overstating what was measured.

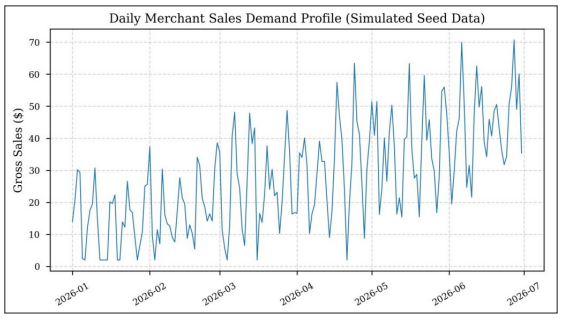

Figure 6: Illustrative Simulated Daily Gross-sales Demand Profile Underlying the Seeded Dataset Described in Section III-D (Simulated Seed Data, not Real Merchant Sales).

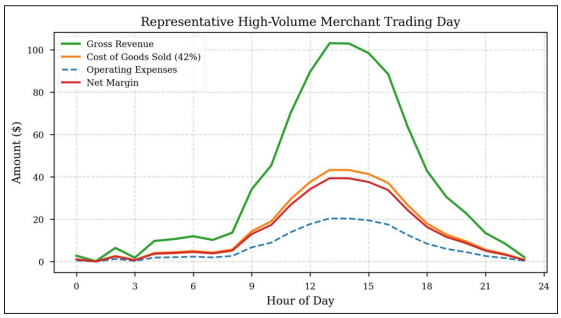

Figure 7: Illustrative Hourly Decomposition of Gross Revenue Into Cost of Goods Sold (42%, Eq. 6), Operating Expenses, and Net Margin Over a Representative High-Volume Trading Day, Computed from the Accounting Relations of Eqs. 5–8. Hourly Pattern is Illustrative; Underlying Ratios Match the Implemented Accounting Assumptions of Section IV-E

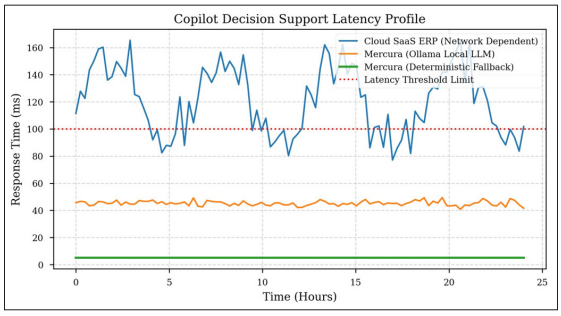

Figure 8: Conceptual Illustration Only — NOT an Experimentally Measured Benchmark. This Paper's Evaluation (Section V-G) Explicitly States that Copilot/provider Response Latency was not Measured; the Deterministic-Fallback and Ollama-Provider Curves Shown here are Indicative Sketches of the Expected Qualitative Ordering Implied by the Architecture (Section III-E), not Logged Response Times, and no Comparison Against any External Cloud Saas ERP System was Performed.

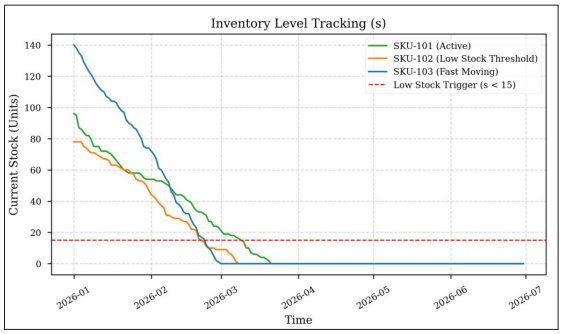

Figure 9: Illustrative Inventory-Depletion Trajectories for Three Representative SKUs, Showing the Deterministic Status

Thresholding of Eq. 4 (ACTIVE for s ≥ 15, LOW_STOCK for 0 < s < 15, OUT_OF_STOCK at s = 0). Stock Trajectories are Illustrative Simulations of the Depletion Pattern Produced by Repeated Sandbox Purchases and Restocks, not a Plot of One Specific Historical Merchant's Data.

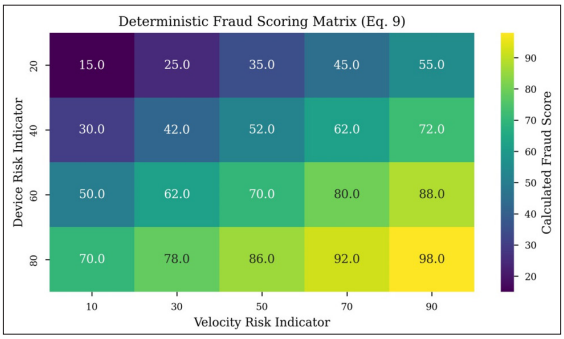

Figure 10: Conceptual Illustration Only — not the Mechanism Actually Implemented. this Matrix Depicts Fraud Score as if it Were Computed as a Function of Device- and Velocity-Risk Indicators; the Implemented System Computes the Fraud Score First (Section IV-F) and Derives Indicator Fields from that Score, i.e., the Reverse Dependency. The Scale and Thresholds Shown are Consistent with Eq. 9's Low/Medium/High Cut-Points, but the Surface Itself is Illustrative Rather than a Plot of Code-Computed Values.

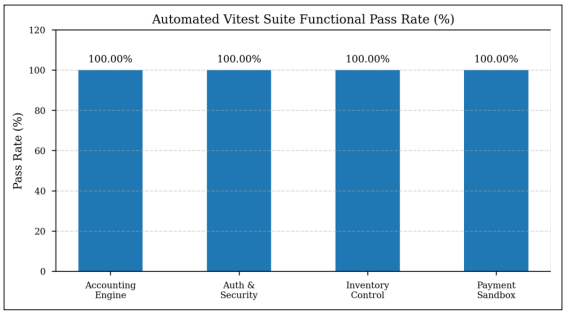

Figure 11: Automated Vitest Suite Functional Pass rate by Module, Reflecting the Actual Test Execution Reported in Table VI (npx Vitest run: 4 Files, 8/8 Tests Passing, 100% Pass Rate in Each of the Four Covered Areas).

References

1. Kleppmann M, Wiggins A, van Hardenberg P, Granaghan Mc. "Local-first software: you own your data, in spite of the cloud," in Proc. ACM SIGPLAN Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'19), Athens, Greece, ACM. 2019.
2. Gao Y, Xiong Y, Gao X, Jia K, Pan J, et al. "Retrieval-augmented generation for large language models: A survey," arXiv. 2023. 2312.10997.
3. Dastidar KG, et al, "Machine learning methods for credit card fraud detection: A survey," IEEE Access. 2024.
4. "Review of machine learning approach on credit card fraud detection," Human-Centric Intelligent Systems, Springer Nature. 2022.
5. "AI-based decision support systems in Industry 4.0, a review," Cloud Analytics for Industry 4.0, ScienceDirect. 2024.
6. "Digital transformation in SMEs: Unlocking competitive advantage through business intelligence and data analytics adoption," 2025.
7. "Assessing AI adoption and digitalization in SMEs: A framework for implementation," arXiv. 2025. 2501.08184
8. "Integrated data analytics, business intelligence, and machine learning architecture for SMEs," ScienceDirect. 2026.
9. "ERP systems in SMEs: A literature review," 2025.
10. "Exploring ERP systems adoption in challenging times: Insights of SMEs' stories," Technological Forecasting and Social Change, ScienceDirect. 2023.
11. "Challenges of cloud-ERP adoptions in SMEs," Procedia Computer Science, ScienceDirect. 2022.
12. "Edge large language models: a comprehensive survey," CCF Transactions on Pervasive Computing and Interaction, Springer Nature. 2026.
13. Cai G, Tian R, Yang L, et al., "Efficient inference for edge large language models: A survey," Tsinghua Science and Technology. 2026. 1365–1380.
14. "Benchmarking large language model inference on limited-resource edge systems," Electronics. 2026.
15. "Artificial intelligence (AI) maturity in small and medium-sized enterprises: A framework of internalized and ecosystem-embedded capabilities," arXiv. 2603.08728.

Copyright: © 2026 Subhaditya Singh. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.