IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

Smart Pharmacy AI: A Full-Stack Web Application for Medicine Inventory Management with an Extensible Computer-Vision and AI-Chatbot Architecture

T Aakash[1], G S Praveen Balaji[2], M Jaiaakash[3], Dr. Beenarani Manoj[4]

Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Chennai, India[1-3]

Professor, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Chennai, India[4]

Abstract: Community pharmacies and small dispensaries in India largely continue to track stock, batch numbers, and expiry dates on paper registers or in disconnected spreadsheets, a practice that leaves them exposed to stock-outs, undetected expired inventory, and slow reconciliation. This paper presents Smart Pharmacy AI, a full-stack web application that digitizes medicine inventory management through a Flask and SQLAlchemy REST API backed by a five-table relational data model, and a React (Vite, Tailwind CSS) single-page front end. The system exposes CRUD endpoints for medicine records and a dashboard endpoint that aggregates total stock, low-stock counts, and expired-item counts directly from the database using SQL aggregate functions. Beyond the operational CRUD and analytics layer, the system is architected with dedicated data models and route modules (Scan History, OCR Record, Inventory History, Chat History) that are designed to host a computer-vision medicine-package detector, an OCR-based batch/expiry-date extractor, and a conversational assistant as they are progressively integrated. We describe the system architecture, the relational schema, the implemented REST endpoints, the front-end design, the testing procedure followed, security and data-integrity considerations, a comparison against manual and generic point-of-sale alternatives, and the current implementation status of every module, before outlining the machine-learning components planned for the computer-vision and OCR subsystems and the corresponding development roadmap.

Keywords: Pharmacy Inventory Management, Flask REST API, SQLAlchemy ORM, React.js, Single-Page Application, Computer Vision, Optical Character Recognition, Expiry Tracking, AI Chatbot, Full-Stack Web Development.

I.INTRODUCTION

Retail and hospital pharmacies handle perishable inventory whose value depends on accurate, up-to-date tracking of quantity, batch number, and expiry date. In small and mid-sized pharmacies, this tracking is frequently performed manually, which is slow, error-prone, and does not scale as the number of stock-keeping units grows. Delayed detection of low stock leads to shortages, while undetected expired stock creates both financial loss and patient-safety risk. Web-based inventory systems built on modern full-stack frameworks have been shown to reduce these errors by centralizing data and automating routine checks such as low-stock and expiry alerts [1].Separately, computer vision and optical character recognition (OCR) have matured to the point where they can automate physical stock-taking tasks that were previously manual. Object-detection architectures in the YOLO (You Only Look Once) family are widely used for real-time recognition of physical items, including in medical and pharmaceutical contexts, because they combine competitive accuracy with inference speeds suitable for interactive use [2], [3]. Likewise, OCR pipelines tuned to extract printed dates and lot codes from curved or low-contrast packaging have been proposed for food and pharmaceutical labels, and report that generic OCR engines perform poorly on this task without domain-specific pre-processing or a dedicated text-detection stage [4]. On the conversational side, large-language-model (LLM) chatbots fine-tuned on medical or pharmacy data have recently been shown to answer medicine-related natural-language queries with usable accuracy, suggesting that a similar assistant could sit on top of a pharmacy's own inventory data [5].

Most publicly documented student and open-source pharmacy projects address only one of these concerns at a time: either a CRUD-only inventory tracker with no perception layer, or a standalone computer-vision or chatbot prototype with no integration into an operational inventory system. This gap motivates a design in which the transactional core and

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

the intelligent layer share one schema and one API surface from the outset, so that the two can be developed on independent timelines without a later, disruptive integration effort.

This paper presents Smart Pharmacy AI, a project that combines these two directions: (i) a fully functional web-based medicine inventory system with CRUD operations and a live analytics dashboard, and (ii) an extensible backend schema and module layout — a scanning route, an OCR record model, and a chatbot route — that is designed in advance to host a YOLO-based package detector, an OCR-based label reader, and an LLM-backed pharmacy assistant as those components are trained and integrated. The remainder of the paper is organized as follows. Section II reviews related work in web-based inventory systems, vision-based package recognition, packaging OCR, and pharmacy chatbots. Section III describes the proposed three-tier system architecture. Section IV details the relational schema and the REST API. Section V discusses the front-end design and user experience. Section VI covers implementation details such as session handling, configuration and the dependency stack. Section VII describes the testing procedure followed. Section VIII discusses security and data-integrity considerations. Section IX compares the proposed system against manual and generic alternatives. Section X presents the current implementation status and discussion, and Section XI concludes with a phased roadmap for future work.

II. RELATED WORK

Web-based inventory management. Progressive Web Application (PWA) based inventory systems have been proposed as a scalable, cross-device alternative to native inventory software, combining CRUD operations, barcode/QR scanning and offline-capable synchronization within a single responsive codebase [1]. This line of work establishes that a browser-based front end backed by a REST API is a practical foundation for inventory systems ranging from small enterprises to larger warehouses, which motivates the choice of a React single-page application communicating with a Flask REST API in the present work, rather than a native desktop or mobile client.

Computer-vision package and object recognition. The YOLO family of one-stage object detectors has been applied extensively to medical object-detection tasks — including lesion detection, personal-protective-equipment detection, and instrument recognition — and a systematic review of 124 studies between 2018 and 2023 found that YOLO variants consistently outperform two-stage detectors on inference speed while remaining competitive on accuracy, provided a sufficiently large and well-annotated task-specific dataset is available [2]. A broader survey of the Ultralytics YOLO lineage (YOLOv5 through the current YOLO26 release) further documents architectural changes — including NMS-free inference and small-target-aware label assignment — that are directly relevant to detecting small, visually similar medicine packages under warehouse or counter lighting [3]. These findings inform the choice of a YOLO-based detector, rather than a classical color/edge-threshold pipeline, for the planned Scanner module described in Section III-D.

Packaging OCR and expiry-date extraction. Generic OCR engines such as Tesseract have been reported to fail on printed expiry dates because of dot-matrix fonts, low contrast, and cluttered backgrounds; a two-stage deep-learning pipeline that first detects the date region with a scene-text detector and then recognizes the characters with a dedicated recognition network was shown to substantially outperform direct OCR on this task [4]. This detect-then-recognize pattern — isolate the region of interest before applying character recognition — is the same pattern adopted for the OCR stage of the planned scanning pipeline in this work, where a cropped label region detected by the vision model would be passed to an OCR stage rather than running OCR over the full package image.

AI chatbots for pharmacy and healthcare. LLM-based chatbots fine-tun
ed on medical or pharmacological data have been developed to answer medicine-related questions in natural language; one such system, based on a fine-tuned open-source LLM with a multi-modal text/speech interface, reported 87% accuracy and a 92.16% F1 score on medicine-prescription queries in a usability study with 33 participants [5]. This body of work indicates that a conversational layer over structured pharmacy data is both technically feasible and well-received by users, which supports the architectural decision in this paper to reserve a Chat History table and a dedicated chatbot route ahead of wiring the assistant to a live completion endpoint.

Positioning of this work. Where the systems above each address one concern in isolation — general inventory PWAs, medical object detection, packaging OCR, or pharmacy chatbots — Smart Pharmacy AI is positioned as an integration exercise: a single relational schema and REST surface designed so that a YOLO detector, a detect-then-recognize OCR stage, and an LLM chatbot can each be dropped into an already-operational CRUD and analytics system without a later schema migration or architectural rewrite.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

III. PROPOSED SYSTEM ARCHITECTURE

The system follows a three-tier architecture consisting of a presentation layer, an application/API layer, and a data layer, connected over a REST interface secured with Cross-Origin Resource Sharing (CORS). Fig. 1 (described in Table III) summarizes the responsibility of each tier; the subsections below expand on each in turn, including the two planned extension points that are architected into the system ahead of their full implementation.

TABLE III. Architectural Tiers and Responsibilities

Tier	Technology	Responsibility
Presentation	React, Vite, Tailwind CSS, react-router	Rendering, client-side routing, state, API calls
Application / API	Flask, Flask-CORS, Blueprints	Request handling, validation, business logic
Data	SQLAlchemy ORM, SQLite	Persistence, aggregate queries, schema

A. Presentation Layer

The front end is a React single-page application scaffolded with Vite and styled with Tailwind CSS. Client-side routing (react-router) maps seven views to a persistent navigation bar: a Dashboard page that renders summary statistics and stock charts, a Medicines page listing all inventory records, a Medicine Form page for adding new stock, a Scanner page reserved for camera-based capture, an Inventory History page, an Expiry page that surfaces items nearing or past their expiry date, and a Chatbot page that provides a conversational query interface over the inventory. All pages communicate with the backend exclusively through a thin API service module, which keeps the HTTP layer isolated from the UI components and means a future change of backend (for example, adding authentication headers) touches one file rather than every page.

B. Application / API Layer

The backend is implemented in Flask and organized as two blueprints — a medicines blueprint exposing CRUD operations and a dashboard blueprint exposing aggregate analytics — both mounted under an /api prefix. Flask-CORS is enabled so the React development server can call the API during local development. Environment configuration is loaded from a .env file via python-dotenv when present, and the SQLite database is initialized automatically on application start-up by creating any missing tables from the ORM models. Organizing routes as blueprints rather than as flat routes on a single Flask app keeps the medicines and dashboard concerns independently testable and makes it straightforward to register a third blueprint — for example a scan or chatbot blueprint — without touching existing route definitions.

C. Data Layer

Persistence uses SQLAlchemy's declarative ORM over a file-based SQLite database, chosen for its zero-configuration setup during development while remaining swappable for a server-based engine (e.g., PostgreSQL) in production, since SQLAlchemy abstracts the underlying SQL dialect behind the same session and query API. Five tables model the domain: Medicine (the core inventory record), Scan History, OCR Record, Inventory History and Chat History. The latter three are deliberately included ahead of full feature implementation so that the computer-vision, OCR and chatbot subsystems described below and evaluated in Section X can be integrated without a schema migration disrupting the already-deployed CRUD and dashboard functionality.

D. Planned Computer-Vision and Conversational Subsystems

Two extension points are architected into the system. First, a scanning pipeline: images captured on the Scanner page are intended to be logged to ScanHistory (image path, detected name, confidence, detected quantity) and passed to a detector module, for which the requirements file already reserves OpenCV and the Ultralytics YOLO package. Detected label regions would then be cropped and passed to an OCR stage — EasyOCR is reserved for this purpose in the project dependencies — whose output is written to OCRRecord (medicine name, batch number, manufacturing date, expiry date, MRP) for operator confirmation before it updates the Medicine table. This mirrors the detect-then-recognize pattern discussed in Section II, where a lightweight detector first isolates the region of interest before a heavier recognition model is applied, which keeps end-to-end latency low enough for interactive counter use. Second, a conversational layer: the Chatbot page already calls a postChat service and persists question/answer pairs to ChatHistory, and is designed to be backed by an OpenAI-compatible completion API that answers natural-language questions (e.g., "which items expire this month?") by querying the Medicine table and formatting the result as a grounded, structured answer rather than a free-

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026
DOI: 10.17148/IJIREEICE.2026.14807

form generation, reducing the risk of hallucinated stock figures that would otherwise be unacceptable in an inventory context.

E. Deployment and Runtime Environment

The backend runs as a standard WSGI Flask application (development server on port 5000) inside a Python virtual environment, while the front end is served by the Vite development server during development and would be built to static assets for production. Because the API and the client are decoupled behind HTTP and CORS rather than server-side templating, either side can be deployed independently — for example, the React build can be hosted on a static CDN while the Flask API and SQLite/PostgreSQL database run on a separate application server — without changes to the other.

IV. DATABASE SCHEMA AND REST API

This section details the full relational schema underlying the system and the REST endpoints implemented and exercised in the current build, followed by a brief discussion of the API design conventions used.

A. Relational Schema

Table IV summarizes the core Medicine entity, which anchors the inventory and is the only table currently populated through the user interface. Table V summarizes the four supporting tables that record scan results, OCR extractions, stock-movement history, and chatbot transcripts respectively; these are present in the schema and ready to receive data as their corresponding modules are completed.

TABLE IV. Medicine Entity — Core Fields

Field	Type	Description
name	String	Medicine / brand name
batch_number	String	Manufacturer batch code
manufacturing_date	Date	Date of manufacture
expiry_date	Date	Date of expiry
mrp	Float	Maximum retail price
quantity	Integer	Current stock on hand
minimum_stock	Integer	Reorder threshold
category	String	Therapeutic / storage category

TABLE V. Supporting Entities for the Vision, OCR, History and Chat Modules

Table	Key Fields	Purpose
ScanHistory	image_path, detected_name, confidence, detected_quantity	Logs each camera capture and the detector's output
OCRRecord	medicine_id, raw_text, batch_number, mfg./expiry date, mrp	Stores OCR-extracted label fields pending confirmation
InventoryHistory	medicine_id, change_type, quantity, description	Audit trail of stock additions, removals and adjustments
ChatHistory	question, answer	Transcript of chatbot interactions for later review

B. REST API

Table VI lists the endpoints implemented in the medicines and dashboard blueprints. All endpoints return a JSON envelope of the form {"success": true|false, "data": ...} (or an "error" field on failure), which gives the front end a single, predictable shape to parse regardless of endpoint.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

TABLE VI. Implemented REST Endpoints

Method	Endpoint	Function
GET	/api/medicines	List all medicine records
POST	/api/medicines	Create a new medicine record
GET	/api/medicines/<id>	Retrieve a single record
PUT	/api/medicines/<id>	Update an existing record
DELETE	/api/medicines/<id>	Remove a record
GET	/api/dashboard	Aggregate stock, low-stock and expiry statistics

The dashboard endpoint deserves particular attention because it is computed rather than stored: total medicine count and total stock quantity are produced with SQL COUNT and SUM aggregates over the Medicine table, the low-stock count filters records where quantity is below minimum_stock, and the expired count filters records where expiry_date is earlier than the database's current date, all evaluated at query time so that the dashboard is always consistent with the latest CRUD operation rather than a periodically refreshed cache.

Each write endpoint (POST, PUT, DELETE) wraps its SQLAlchemy session operations in a try/except block that catches SQLAlchemyError, calls session.rollback() on failure, and returns a 400 response with the error message, while every endpoint closes its session in a finally block regardless of outcome. This request-scoped session lifecycle — one SessionLocal() per request, always closed — avoids the connection leaks that can occur if a long-lived session is shared across requests in a multi-threaded WSGI server.

V. FRONTEND DESIGN AND USER EXPERIENCE

The client is organized around a persistent NavBar component and a routed main content area; a location-keyed wrapper around the route outlet applies a page-enter transition class on every navigation so that moving between pages feels continuous rather than a hard reload, while React Router itself avoids a full page refresh. Table VII summarizes the responsibility of each page.

TABLE VII. Front-End Pages and Responsibilities

Page	Route	Responsibility
Dashboard	/dashboard	Summary statistics and stock chart from /api/dashboard
Medicines	/medicines	Tabular listing of all inventory records
Medicine Form	/medicines/add	Form to create a new medicine record
Scanner	/scan	Reserved camera-capture entry point for the vision pipeline
Inventory History	/inventory	Reserved view of the InventoryHistory audit trail
Expiry	/expiry	Items nearing or past their expiry date
Chatbot	/chatbot	Conversational query interface over the inventory

All network access is centralized in a single API service module rather than being called directly from components, so that pages depend on a small function surface (for example postChat for the chatbot) instead of on request/response details such as headers or the base URL. Within the Chatbot page specifically, component state (messages, the current input value, and a loading flag) is held with React's useState hook; a submitted question is optimistically appended to the message list before the network call resolves, and the corresponding answer — or a fallback error message if the request fails — is appended once the promise settles, keeping the interface responsive even while the backend call is in flight. Styling is handled entirely with Tailwind CSS utility classes rather than a separate stylesheet per component, which keeps visual variations (such as the distinct message-bubble colors for user versus bot turns in the Chatbot page) co-located with the markup that uses them and avoids CSS specificity conflicts as the page count grows.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

VI. IMPLEMENTATION DETAILS

Table VIII lists the primary dependencies used on each side of the stack, drawn directly from the project's requirements.txt and package.json manifests, to make the implementation reproducible.

TABLE VIII. Primary Dependencies

Layer	Dependency	Role
Backend	Flask 3.0	HTTP application framework
Backend	Flask-CORS 5.0	Cross-origin request handling
Backend	SQLAlchemy 2.0	ORM and database session management
Backend	python-dotenv 1.0	Environment variable loading
Backend (planned)	OpenCV, Ultralytics YOLO, EasyOCR	Detection and OCR pipeline
Frontend	React + Vite	Component rendering and dev/build tooling
Frontend	react-router-dom	Client-side routing
Frontend	Tailwind CSS	Utility-first styling

Configuration is environment-driven: the Flask application conditionally loads a .env file with python-dotenv only if one is present, so the same codebase runs with sensible defaults in a fresh checkout and with overridden values (for example, a future database URL or LLM API key) in a deployed environment, without a code change. The SQLite database file itself is created inside an instance/ directory that is derived at runtime relative to the database module's own file location rather than the process's working directory, which keeps the database path stable regardless of where the Flask process is launched from.

Error handling follows a consistent pattern across every write endpoint: business logic executes inside a try block, a SQLAlchemyError triggers an explicit session.rollback() before a 400 JSON error response is returned, and the session is closed in a finally block whether or not an exception occurred. Read endpoints follow the same open/query/close-in-finally shape without the rollback branch. This uniformity means a new endpoint (for example, one added for the scanning or chatbot blueprint) can be implemented by following the same three-part template already established by the medicines blueprint.

VII. TESTING AND VALIDATION

The implemented CRUD and dashboard functionality was validated manually against the running Flask development server using direct HTTP requests to each endpoint and by exercising the corresponding React pages in the browser. Table IX summarizes the test cases exercised and their observed outcome.

TABLE IX. Manual Test Cases

#	Test Case	Expected / Observed Result
1	POST /api/medicines with valid payload	201 response; new record persisted and visible in GET
2	GET /api/medicines	200 response; JSON array of all records with ISO-formatted dates
3	GET /api/medicines/<id> for a non-existent id	404 response with a not-found error message
4	PUT /api/medicines/<id> updating quantity	200 response; updated value reflected on next GET
5	DELETE /api/medicines/<id>	200 response; record no longer present in subsequent GET
6	GET /api/dashboard after adding/removing stock	Totals and low-stock/expired counts update accordingly
7	Front end: add a medicine via the Medicine Form page	New record appears on the Medicines and Dashboard pages

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 8, August 2026

DOI: 10.17148/IJIREEICE.2026.14807

All seven cases produced the expected outcome. Because a formal automated test suite (for example, pytest against a temporary SQLite database) has not yet been added to the repository, this validation should be read as functional confirmation of the current build rather than as regression protection; adding such a suite is included in the roadmap in Section XI.

VIII. SECURITY AND DATA-INTEGRITY CONSIDERATIONS

Because all database access goes through SQLAlchemy's parameterized query interface rather than string-formatted SQL, the system is not exposed to classical SQL-injection through the medicines or dashboard endpoints. CORS is currently enabled broadly to simplify local development, which is appropriate for the current single-developer, localhost deployment but would need to be restricted to a specific origin before any public deployment. Two further gaps are acknowledged as current limitations rather than as solved problems: first, the API does not yet implement authentication or role-based access control, so any client that can reach the Flask process can read or write inventory data — acceptable for a prototype behind a private network, but a prerequisite for production use by multiple pharmacy staff with different permissions; second, request payloads are consumed with request.get_json() and read via .get() without a schema-validation layer (such as Marshmallow or Pydantic), so malformed input is currently caught only indirectly through the database's own type and not-null constraints rather than through an explicit, user-facing validation error.

IX. COMPARATIVE ANALYSIS

Table X positions the proposed system against two common alternatives available to a small pharmacy: a manual paper/spreadsheet register, and a generic, non-extensible point-of-sale (POS) inventory package.

TABLE X. Comparison Against Manual and Generic POS Alternatives

Capability	Manual Register	Generic POS System	Smart Pharmacy AI
Real-time low-stock alerting	No	Sometimes	Yes (dashboard query)
Real-time expiry alerting	No	Sometimes	Yes (dashboard query)
Web-based, multi-device access	No	Varies	Yes (React SPA + REST API)
Vision-based package scanning	No	Rarely	Planned (schema-ready)
OCR label/expiry extraction	No	No	Planned (schema-ready)
Natural-language inventory queries	No	No	Planned (schema-ready)
Extensible, open codebase	N/A	No (closed source)	Yes

The comparison highlights that the immediate, measurable advantage of the proposed system over a manual register is the real-time dashboard, which requires no computer-vision component at all; the vision, OCR and chatbot rows are marked as planned rather than delivered, consistent with the honest implementation status reported in Section X, and are the dimension along which the system would differentiate itself further from a generic, closed-source POS package once completed.

X. IMPLEMENTATION STATUS AND DISCUSSION

At the current stage, the CRUD layer and the analytics dashboard are fully implemented and functional end to end: the medicines blueprint performs create, read, update and delete operations against the SQLite database with SQLAlchemy session handling and rollback on integrity errors, and the dashboard endpoint computes total medicine count, total stock quantity, low-stock count (quantity below minimum_stock) and expired-item count (expiry_date earlier than the current date) using SQL aggregate functions evaluated at query time, together with a chart-ready series of medicine names and quantities for the front end. The React client consumes these endpoints through a dedicated API service layer and renders them across the Dashboard, Medicines, Add-Medicine and Expiry pages with client-side routing and Tailwind-based styling.

The computer-vision, OCR and chatbot subsystems are at an earlier stage: their data models (ScanHistory, OCRRecord, ChatHistory) and route/page scaffolding already exist, and the project dependencies reserve the corresponding libraries (OpenCV, Ultralytics YOLO, EasyOCR, an OpenAI-compatible client), but the detector and OCR modules are placeholders pending a labelled dataset of medicine packaging and model training, and the chatbot's language-model

DOI: 10.17148/IJIREEICE.2026.14807

integration is not yet wired to a live completion endpoint. This staged approach — shipping the transactional core first and designing the schema so that the perception layer can be dropped in without a migration — mirrors the two-stage detect-then-recognize pipelines used in prior expiry-date and package OCR work, where a lightweight detector isolates the region of interest before a recognition model is applied [2], [4], and is consistent with review evidence that YOLO-based detectors are best deployed once a task-specific, annotated dataset is available rather than adapted directly from general-purpose weights [3].

A practical implication of the current design is that the low-stock and expiry alerts already give a pharmacy operator real value without any computer-vision component: because minimum_stock and expiry_date are first-class, indexed fields on every record, the dashboard query surfaces at-risk stock on every page load at negligible computational cost, which is the highest-value automation for a small pharmacy relative to its implementation effort. The reported usability and accuracy figures for fine-tuned pharmacy chatbots in prior work [5] further suggest that the planned conversational layer is a realistic near-term extension once the underlying Medicine table is queried through a grounded (rather than free-generation) prompting strategy, which is the design already assumed for the Chat History integration described in Section III-D.

XI. CONCLUSION AND FUTURE WORK

This paper presented Smart Pharmacy AI, a Flask/React/SQLite web application for pharmacy medicine inventory management, and described a system architecture in which the transactional CRUD and analytics core is fully implemented while the data schema and route layout are deliberately prepared to host a computer-vision package detector, an OCR-based label reader and a conversational assistant. This design allows the perception and language-model components to be integrated incrementally without disrupting the operational core that pharmacy staff already depend on. Table XI summarizes the planned development roadmap in three phases.

TABLE XI. Planned Development Roadmap

Phase	Milestone	Depends On
1	Automated test suite (pytest) and input-schema validation	Current CRUD/dashboard API
2	Dataset collection/annotation and YOLO detector training for the Scanner page	Phase 1
3	OCR pipeline integration into OCRRecord with operator confirmation step	Phase 2
4	Chatbot wired to a live, grounded completion API over ChatHistory	Phase 1
5	Authentication, role-based access, and expiry push/email notifications	Phase 1

Executing this roadmap would close the gap between the currently operational transactional system and the fully intelligent pharmacy assistant outlined in Sections III and X, while preserving the schema and API contract already validated in Sections IV, VI and VII.

REFERENCES

[1]. A. Desai, "Enhancing Inventory Management with Progressive Web Applications (PWAs): A Scalable Solution for Small and Large Enterprises," arXiv:2506.11011, 2025.

[2]. M. G. Ragab et al., "A Comprehensive Systematic Review of YOLO for Medical Object Detection (2018 to 2023)," IEEE Access, 2024.

[3]. R. Sapkota and M. Karkee, "Ultralytics YOLO Evolution: An Overview of YOLO26, YOLO11, YOLOv8 and YOLOv5 Object Detectors for Computer Vision and Pattern Recognition," arXiv:2510.09653, 2026.

[4]. V. Florea and T. Rebedea, "Expiry Date Recognition Using Deep Neural Networks," International Journal of User-System Interaction, vol. 13, no. 1, pp. 1–17, 2020.

[5]. A. Azam, Z. Naz, and M. U. G. Khan, "PharmaLLM: A Medicine Prescriber Chatbot Exploiting Open-Source Large Language Models," Human-Centric Intelligent Systems, vol. 4, no. 4, pp. 527–544, 2024.