
Meditrak

Health Information Management System

IT Capstone Project Documentation

Course	Senior Design Capstone II — Spring 2026
Institution	Florida International University — KFSCIS
Instructor	Dr. Masoud Sadjadi
Team	Johnathan Doralus · Isaiah Lora · Barbaro Ibanez · Noah Cook
Tech Stack	Python 3.14 · Flask · SQLite 3 · Jinja2 · HTML5/CSS3/JS
Version	1.0 — April 2026

ACM Student Outcomes (O1–O6)

Outcome	Description	Addressed In
O1. Problem Analysis & Requirements	Elicit and analyze user/stakeholder needs; define constraints and success criteria.	§2, Appendix A, Backlog
O2. System Design & Implementation	Design solutions using appropriate abstractions, patterns, and tools; implement maintainable code.	§3, §5, README
O3. Experimentation, Testing & Evaluation	Devise evaluation methods; collect evidence; interpret results to improve the system.	§6, Test Suite (62 tests)
O4. Communication & Collaboration	Communicate effectively with technical and non-technical stakeholders; collaborate in teams.	§1, §9, Scrum Evidence
O5. Professional, Ethical, Legal & Societal	Recognize responsibilities; address ethics, security, privacy, accessibility, and sustainability.	§7, Security Model
O6. Systems Integration, DevOps & Operations	Discipline-specific depth outcome; addressed in dedicated sections below.	§4, DEVOPS.md

Outcome & Rubric Crosswalk

Outcome	Where It Appears	External Evidence	Status
O1	§2 Problem & Requirements; §3 System Overview	Backlog, User Stories, Sprint Docs	☑
O2	§3 Architecture; §5 Implementation Notes	app.py, init_db.py, README	☑
O3	§6 Testing & Evaluation	62 passing pytest tests, run_tests.bat	☑
O4	§1 Executive Summary; §9 Future Work	Poster, Scrum meeting minutes	☑
O5	§7 Security/Privacy/Compliance	SHA-256 hashing, parameterized queries	☑
O6	§4 Discipline-Specific Depth	DEVOPS.md, logging, start.bat, run_tests.bat	☑

1. Executive Summary

Small healthcare clinics often rely on paper forms and spreadsheets to manage patient appointments, staff records, and health information. These methods are slow, error-prone, and offer no access control, audit trail, or data integrity guarantees. The result is scheduling conflicts, lost records, and inefficient workflows that directly impact patient care.

Meditrak is a full-stack web application built to solve this problem for small and mid-size general practice clinics. The platform provides a unified, role-based management system that serves four distinct user types: Administrators, Doctors, Receptionists, and Patients. Each role has a tailored dashboard and precisely scoped permissions enforced at the server level on every request.

Key capabilities include patient self-registration, self-service appointment booking with live calendar availability, double-booking prevention, visit summary recording (diagnosis, treatment, prescriptions, follow-up), live patient search for receptionists, appointment analytics dashboards, and a data retention archive system. The application is deployed locally as a Python Flask server with SQLite as the backend database — requiring no external services, cloud accounts, or infrastructure beyond Python itself.

From a DevOps perspective, the project includes a 62-test automated QA suite (pytest), structured application logging with rotating log files, a one-command environment provisioning script, and a documented release process. All tests pass as of the final submission.

Meditrak was built over two capstone semesters using disciplined Scrum methodology, with documented sprint planning, daily stand-ups, backlog grooming, sprint reviews, and retrospectives across six sprints.

2. Problem & Requirements (O1)

2.1 Problem Context

Small healthcare clinics typically operate without a dedicated IT system. Patient intake is handled via paper forms. Appointment scheduling is managed in paper calendars or basic spreadsheets. There is no centralized record of visit history, diagnoses, or prescriptions. Staff have no way to search patients by name or ID, and double-bookings occur frequently due to lack of real-time conflict detection.

2.2 Stakeholders & Personas

Stakeholder	Role & Needs
Clinic Administrator	Needs full visibility into clinic operations, staff, and patient roster. Manages all accounts and monitors analytics.
Doctor	Needs to see their upcoming schedule, review patient histories, and document visit outcomes.
Receptionist	Needs to schedule appointments quickly, search patients by name or phone, and manage the daily schedule.
Patient	Wants to self-register, book their own appointments, and view their health records and visit history.

2.3 Functional Requirements

- Role-based authentication with four roles: Admin, Doctor, Receptionist, Patient
- Patient self-registration from the public login page
- Patient self-scheduling with real-time calendar availability view
- Receptionist appointment scheduling with live patient search (no dropdown required)
- Double-booking prevention — system detects and blocks time slot conflicts
- Appointment management: view, cancel, reschedule, mark complete
- Visit summaries: doctors record diagnosis, treatment, prescriptions, follow-up date
- Admin CRUD management for patients, doctors, and staff
- Appointment analytics: status trends, monthly volume, service popularity, doctor utilization
- Data retention archive: configurable archival and purge of old records

2.4 Non-Functional Requirements

- Security: all passwords hashed with SHA-256; all SQL via parameterized queries; role enforcement on every route
- Performance: all page loads complete in under 2 seconds on local hardware
- Reliability: application starts cleanly via start.bat on any Windows 10/11 machine with Python 3.10+
- Maintainability: single app.py file with clear section comments; Jinja2 templates for clean separation of concerns
- Testability: 62 automated tests with isolated in-memory database fixtures

2.5 Prioritized Product Backlog (Top 15 Items)

Story #	User Story	Acceptance Criteria	Points
1	Scheduling System Rework	Appointments link patients, doctors, services with datetime	10
2	Frontend Design Rework	Modern UI with sidebar navigation and role dashboards	10
3	Appointment Backend Refinement	Foreign keys, duration, status, reason fields	5
4	Rework Home Page UI	Stat cards, today's schedule table	5
5	Rework Schedule functions	Add reasoning for appointment scheduling and easier navigation	5
6	Product Documentation	Organize data and update main documentation	5
7	Staff/Doctor Credential Handling	Hashed passwords, unique usernames, role isolation	20
8	Patient Data Access Control	Patients see only their own records	20
9	Appointment Analytics	Charts for status, monthly trend, service, utilization	20
10/12	Data Retention Policies	Archive and purge old records with configurable thresholds	10
11	Patient Portal	Portal shows upcoming appointments, history, diagnoses	20
13	Test Appointment Analytics	Analytics data verified against known seed appointments	15
14	Test Patient Portal	Portal shows correct upcoming and past appointments	15
15	Create poster	Clear poster with issue, requirements, solution, etc	20
16	Create Presentation	PowerPoint of project with explanations of the process and final product	20
17	UI Adjustments	Fix color inconsistencies, font sizing, and button placements	12
18	Finalize documentation	Finalize references, user stories, and section 7	20
19	Final Deliverable prep	Organize file paths and edit videos for final view	20

3. System Overview & Architecture (O2)

3.1 Architecture Overview

Meditrak follows a server-side MVC pattern. Flask acts as the controller layer, routing HTTP requests to Python functions that query SQLite and pass data to Jinja2 templates. There is no JavaScript framework — all rendering is server-side. A small vanilla JS layer handles UI interactions (modals, live search, calendar).

3.2 Request Lifecycle

Layer	Technology	Responsibility
Client	Browser (HTML/CSS/JS)	Renders UI, handles form submission, calendar interaction
Routing	Flask (app.py)	Maps URLs to functions, enforces auth via decorators
Auth	@role_required decorator	Checks session role on every protected route
Business Logic	Python functions in app.py	Validates input, runs queries, prepares template context
Data	SQLite 3 (hims.db)	Stores all application data with foreign key constraints
View	Jinja2 templates	Role-specific HTML rendered server-side
Logging	Python logging module	Writes structured events to logs/meditrak.log

3.3 Technology Stack

Component	Technology
Language	Python 3.14
Web Framework	Flask 3.x
Database	SQLite 3 (built into Python — no install required)
Templating	Jinja2
Frontend	HTML5, CSS3 (custom design system), Vanilla JavaScript
Authentication	Flask sessions + SHA-256 password hashing (hashlib)
Testing	pytest with isolated in-memory SQLite fixtures
Logging	Python logging module with RotatingFileHandler
Deployment	Local — start.bat launches Flask dev server on port 5000
Fonts	DM Serif Display + DM Sans (Google Fonts)

3.4 Database Schema

The database consists of 9 tables connected by foreign key relationships:

Table	Primary Key	Key Relationships
admins	admin_id	Standalone — system admin accounts
departments	department_id	Referenced by doctors
doctors	doctor_id	FK: department_id → departments
receptionists	receptionist_id	Referenced by appointments
patients	patient_id	Referenced by appointments
services	service_id	UNIQUE on name; referenced by appointments
appointments	appointment_id	FK: patient_id, doctor_id, receptionist_id, service_id
visit_summaries	summary_id	UNIQUE FK: appointment_id → appointments
archived_appointments	archive_id	Snapshot copy; no FK constraints

4. Discipline-Specific Depth (O6)

4.1 Architecture & Integration

Meditrak is a self-contained monolithic application designed for local clinic deployment. The architecture prioritizes simplicity and zero-dependency deployment: SQLite eliminates the need for a database server, Flask's built-in development server handles HTTP, and all authentication is session-based. No cloud services, message queues, or microservices are used. The single `app.py` file contains all route definitions, making the codebase easy to audit and modify.

4.2 DevOps & SRE

CI — Automated Testing

Meditrak includes a 62-test pytest suite covering three suites:

- `test_auth.py` — 15 tests covering login, logout, invalid credentials, and session handling
- `test_roles.py` — 25 tests verifying every role's permitted and blocked routes
- `test_features.py` — 22 tests covering registration, scheduling, double-booking, cancellation, and search API

Tests run against an isolated in-memory SQLite database. No real data is modified. Run with: `python -m pytest tests/ -v`

CD — Deployment

`start.bat` serves as the deployment script. It installs Flask, runs `init_db.py` to provision the database, and starts the server. `init_db.py` is idempotent — it can be run repeatedly without duplicating data thanks to `INSERT OR IGNORE` and a fresh-database check for sample appointments.

Observability — Logging

Application events are written to `logs/meditrak.log` using Python's `RotatingFileHandler` (rotates at 1MB, keeps 3 files). Logged events include:

- `LOGIN SUCCESS` — username, role, IP address
- `LOGIN FAILED` — attempted username, IP address
- `LOGOUT` — username and role
- `REGISTRATION` — new patient username and email
- `APPOINTMENT SCHEDULED` — patient ID, doctor ID, datetime, scheduled by
- `DB ERROR` — exception message for database failures

4.3 Operations & Security

Access Control

Every route in `app.py` is protected by one of two decorators: `@login_required` (any authenticated user) or `@role_required` (specific roles only). Attempting to access a restricted route redirects to the dashboard. Session data is stored server-side by Flask — clients only hold a signed cookie.

Password Security

All passwords are hashed with SHA-256 via Python's hashlib before storage. Plain-text passwords are never written to the database. During login, the input is hashed and compared against the stored hash.

SQL Injection Prevention

All database queries use SQLite's parameterized query interface (the ? placeholder). No user input is ever concatenated directly into a SQL string.

Data Retention

The archive system allows admins to move completed/cancelled appointments older than a configurable threshold into an archive table. A separate purge operation permanently deletes archive records older than a minimum of 180 days. Both operations are documented in DEVOPS.md with the release checklist.

5. Implementation Notes (O2)

5.1 Repository Structure

The project is organized as a single Flask application package:

File / Folder	Purpose
app.py	All routes, decorators, DB queries, logging setup. ~400 lines.
init_db.py	Database provisioning. Creates all tables and seeds demo data.
start.bat	Windows deployment script. Installs Flask, provisions DB, starts server.
run_tests.bat	Runs pytest suite. Prints PASSED / FAILED summary.
DEVOPS.md	CI/CD documentation, logging reference, release checklist.
database/hims.db	SQLite database file. Auto-created on first run.
logs/meditrak.log	Application log. Auto-created. Rotates at 1MB.
tests/	3 test files + conftest.py. 62 total tests.
static/css/style.css	Full design system. ~500 lines. CSS custom properties for theming.
static/js/main.js	Modal open/close, stat counter animation.
templates/	Jinja2 HTML templates organized by role and feature.

5.2 Coding Conventions

- All routes follow the pattern: validate auth → query DB → pass to template
- Database connection opened and closed within each request function (no persistent connections)
- SQLite rows are accessed as dicts via `conn.row_factory = sqlite3.Row`
- All user input sanitized through SQLite parameterized queries
- Jinja2 autoescaping active for all HTML templates (XSS prevention)
- CSS uses custom properties (variables) for all colors and spacing

5.3 Notable Patterns & Tradeoffs

The biggest tradeoff is the monolithic app.py design. All routes are in a single file for simplicity and auditability, but this would need to be split into blueprints for a production system with more than a few developers. SQLite was chosen over MySQL for zero-dependency local deployment but would need to be replaced with PostgreSQL or MySQL for concurrent multi-user production use. SHA-256 was used for password hashing for compatibility with Python's standard library; a production system should use bcrypt or Argon2.

6. Testing & Evaluation (O3)

6.1 Testing Strategy

Meditrak's test suite uses pytest with a conftest.py fixture that creates a temporary isolated SQLite database for each test run. This means tests are fully independent, never touch real data, and can run in any order. The fixture provisions all required tables and demo accounts before each test.

6.2 Test Coverage Summary

Suite	File	Tests	Coverage Area
Authentication	test_auth.py	15	Login, logout, wrong credentials, session clearing
Role Access	test_roles.py	25	All 4 roles — permitted and blocked routes
Features	test_features.py	22	Registration, scheduling, double-booking, search API
TOTAL		62	All passing as of final submission

6.3 Key Test Cases

- test_login_wrong_password — verifies invalid credentials are rejected
- test_patient_cannot_access_admin_dashboard — verifies role isolation
- test_double_booking_prevented — books a slot then attempts to book the same slot for a different patient; verifies conflict error
- test_successful_registration — creates account via form and verifies success message
- test_registered_user_can_login — registers then logs in with new credentials
- test_appointment_cancellation — schedules then cancels, verifies cancelled status
- test_search_blocked_for_patients — verifies search API returns 302 for patient role

6.4 Running Tests

Double-click run_tests.bat or run in terminal:

```
python -m pip install pytest      (first time only)
python -m pytest tests/ -v
```

7. Security, Privacy, Accessibility & Compliance (O5)

7.1 Security

Threat	Mitigation
Password compromise	SHA-256 hashing via Python hashlib. Plain-text never stored.
SQL injection	All queries use SQLite parameterized placeholders (?). No string concatenation.
Unauthorized access	@role_required decorator on every protected route. Server-side session.
Session hijacking	Flask signed cookie with secret key. Session regenerated on login.
Cross-site scripting (XSS)	Jinja2 autoescaping active on all templates.
Failed login tracking	Every failed login attempt logged with username and IP to meditrak.log.

7.2 Privacy

Meditrak stores patient personal data (name, DOB, gender, email, phone, insurance provider). In production deployment, this data would be subject to HIPAA regulations. For this capstone submission:

- All demo data is fictional — no real patient information is included
- Passwords are hashed and never displayed or logged
- Patient records are only accessible to the patient themselves and authorized staff
- Each role's data access is scoped at the query level in app.py

7.3 Accessibility

- Semantic HTML structure with proper heading hierarchy (h1→h2→h3)
- Form labels associated with all input fields
- Color contrast ratios meet WCAG AA standards (teal #00b8a6 on dark navy)
- Error messages displayed inline next to relevant form fields

7.4 Ethical Considerations

Meditrak handles sensitive health information. Key ethical considerations include the right of patients to view their own records (implemented via the patient portal), the principle of minimum necessary access (each role only sees what they need), and data retention limits (archive and purge system). The system does not share data with any third parties and operates entirely on local infrastructure.

8. Deployment & Operations

8.1 Local Deployment

Meditrak is deployed locally via a Windows batch script. The deployment process is:

1. Install Python 3.10+ from python.org (add to PATH)
2. Extract hims_flask.zip to any folder
3. Double-click start.bat
4. Open browser at <http://localhost:5000>

start.bat handles: `pip install flask`, `python init_db.py`, `python app.py`. The server runs on port 5000.

8.2 Environment Requirements

Requirement	Specification
Python	3.10 or higher
Flask	3.x (auto-installed by start.bat)
pytest	Latest (for tests only — <code>python -m pip install pytest</code>)
OS	Windows 10/11 (start.bat); macOS/Linux via <code>python app.py</code>
Browser	Any modern browser (Chrome recommended)
Disk	< 10MB for application + database
Port	5000 (configurable in app.py)

8.3 Operations

Stopping the server: CTRL+C in the Command Prompt window.

Restarting: double-click start.bat.

Viewing logs: open logs/meditrak.log in any text editor.

Running tests: double-click run_tests.bat or `python -m pytest tests/ -v`.

Resetting to clean state: delete database/hims.db, then run `python init_db.py`.

9. Limitations & Future Work

9.1 Known Limitations

- SQLite is not suitable for concurrent multi-user production deployments — should be replaced with PostgreSQL
- SHA-256 password hashing should be upgraded to bcrypt or Argon2 for production
- Flask's built-in development server is not production-ready — should be replaced with Gunicorn + Nginx
- No email or SMS notifications for appointment reminders
- Mobile layout is functional but not fully optimized for small screens
- No PDF export for visit summaries or appointment receipts
- No two-factor authentication

9.2 Technical Debt

- app.py is a single large file — should be refactored into Flask blueprints by feature area
- No database migration system (Alembic) schema changes require manual SQL
- CSS is a single large file — should be modularized for larger teams

9.3 Prioritized Roadmap

Priority	Feature	Rationale	Effort
1	Email/SMS appointment reminders	Reduces no-shows; top patient request	Medium
2	bcrypt password hashing	Security upgrade from SHA-256	Low
3	PDF visit summary export	Patients need printable records	Medium
4	Mobile-responsive UI	Staff use phones at reception	Medium
5	Gunicorn + Nginx deployment	Required for production use	High
6	PostgreSQL migration	Required for concurrent users	High
7	Two-factor authentication	HIPAA alignment for production	High

10. References

- [1] Pallets Projects. (2024). Flask Documentation (3.x). <https://flask.palletsprojects.com/>
- [2] Python Software Foundation. (2024). Python 3 Documentation — sqlite3. <https://docs.python.org/3/library/sqlite3.html>
- [3] Python Software Foundation. (2024). Python 3 Documentation — hashlib. <https://docs.python.org/3/library/hashlib.html>
- [4] Python Software Foundation. (2024). Python 3 Documentation — logging. <https://docs.python.org/3/library/logging.html>
- [5] pytest Development Team. (2024). pytest Documentation. <https://docs.pytest.org/>
- [6] Pallets Projects. (2024). Jinja2 Documentation. <https://jinja.palletsprojects.com/>
- [7] OWASP Foundation. (2024). OWASP Top 10. <https://owasp.org/www-project-top-ten/>
- [8] U.S. Department of Health & Human Services. (2024). HIPAA Security Rule Guidance. <https://www.hhs.gov/hipaa/>
- [9] Sadjadi, M. (2025). IT Capstone Project Documentation Template. Florida International University — KFSCIS.

Appendices

A. Installation Guide

See README.md in the project root for the full step-by-step installation guide with environment configuration details.

Quick summary:

5. Install Python 3.10+ from python.org — check "Add python.exe to PATH" during install
6. Extract hims_flask.zip to Documents or Desktop
7. Double-click start.bat — installs Flask, creates database, starts server
8. Open <http://localhost:5000> in any browser
9. Log in with admin / password123 to explore all features

B. User Manual

See README.md Section 5 for the complete role-by-role user manual with task-oriented walkthroughs for all features including:

- Logging in and navigating by role
- Patient self-registration and account creation
- Booking appointments via the calendar interface
- Receptionist scheduling with live patient search
- Doctor visit completion and summary recording
- Admin patient, doctor, and staff management
- Analytics dashboard interpretation
- Data archive and purge operations

C. Admin/Operations Guide

See DEVOPS.md in the project root for the full operations guide including:

- Starting and stopping the server
- Viewing and interpreting log files
- Running the test suite
- Resetting the database to a clean state
- Release checklist for submission packaging

D. API Reference

Meditrak exposes one internal JSON API endpoint used by the receptionist live patient search:

Property	Value
Endpoint	GET /api/patients/search
Auth	Required — admin or receptionist role only
Query Parameter	q (string) — search term (name, email, or patient ID)

Response	JSON array of patient objects: [{patient_id, name, email, phone}]
Empty query	Returns empty array []
No match	Returns empty array []
Unauthorized	302 redirect to dashboard

All other page interactions use standard HTML form POST/GET requests handled server-side by Flask.

E. Poster & Presentation

The project poster (meditrak_poster) is included in the submission. It covers the problem statement, system design, database schema, role-based access model, key features, tech stack, and future plans — formatted for 36" x 48" horizontal printing.

F. Scrum Evidence Index

Complete Scrum documentation is included in the Capstone_II.zip archive submitted with this project. Evidence includes:

Document Type	Coverage
Sprint Planning Minutes	7 sprints — Jan 12, Jan 26, Feb 9, Mar 2, Mar 16, Mar 30, Apr 13,
Daily Scrum Minutes	60+ daily stand-up records from Jan–Apr 2026
Sprint Backlog Grooming	7 grooming sessions with user story refinement
Sprint Review Minutes	7 sprint reviews with accepted/rejected story records
Sprint Retrospective Minutes	7 retrospectives with what went right/wrong and improvement actions