High Level Design (HLD) Presentation

Healthcare Management System

Version: 1.0
Date: February 2026
Status: Final

Table of Contents

---

1. Introduction

1.1 Scope of the Document

This High Level Design (HLD) document describes the architectural design of the **Healthcare Management System (HMS)** — a full-stack web application designed to digitize and streamline core healthcare operations.

In Scope:

- Patient registration and management
- Appointment scheduling and tracking

- Medical records creation and retrieval
- RESTful API backend (Flask)
- React-based frontend UI
- Containerized deployment (Docker / Kubernetes)
- CI/CD pipeline design

Out of Scope:

- Billing and insurance management
- Real-time video consultations
- Mobile application (iOS/Android)
- Third-party EHR integrations (Phase 2)

1.2 Intended Audience

Audience	Purpose
Software Architects	Review system design decisions
Backend Developers	Understand API and database design
Frontend Developers	Understand UI components and API contracts
DevOps Engineers	Understand deployment and infrastructure
QA Engineers	Understand system behavior for test planning
Project Managers	Understand scope and system boundaries
Stakeholders	Review overall system capabilities

1.3 System Overview

The Healthcare Management System is a **3-tier web application** that enables healthcare providers to manage patients, appointments, and medical records through a modern, responsive web interface.

Core Capabilities:

- ☑ Patient lifecycle management (Create, Read, Update, Delete)
- ☑ Appointment scheduling with doctor assignment
- ☑ Medical record creation and retrieval
- ☑ Real-time system health monitoring
- ☑ Secure containerized deployment
- ☑ Automated CI/CD pipeline

Technology Summary:

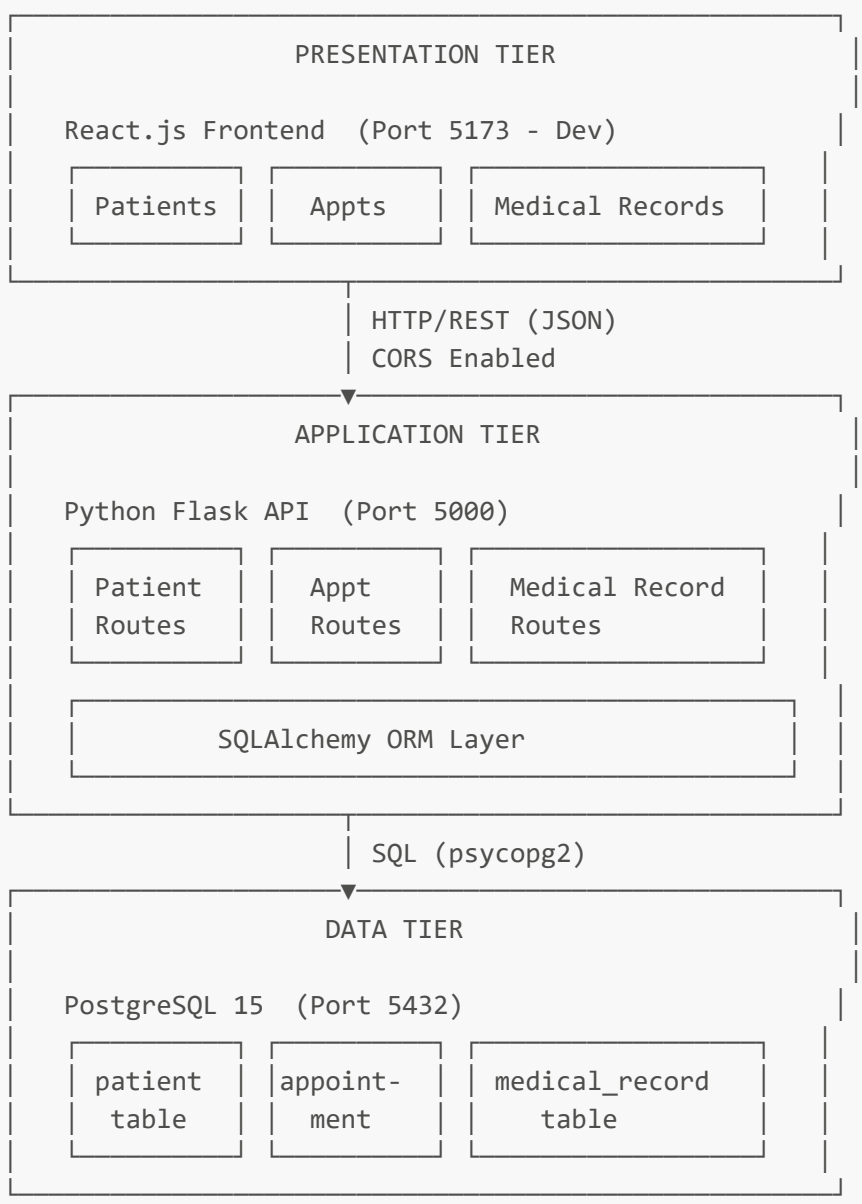
Layer	Technology	Version
Frontend	React + Tailwind CSS	18.x / 3.4
Backend	Python Flask	3.0.0

Layer	Technology	Version
Database	PostgreSQL	15
Container	Docker + Docker Compose	Latest
Orchestration	Kubernetes	1.28+
CI/CD	GitHub Actions + Jenkins	-

2. System Design

2.1 Application Design

The system follows a **3-Tier Client-Server Architecture**:



Design Principles Applied:

- **Separation of Concerns** — Each tier has a distinct responsibility
- **Loose Coupling** — Frontend and backend communicate only via REST API
- **Single Responsibility** — Each module handles one domain
- **DRY (Don't Repeat Yourself)** — Shared utilities and base classes

---

2.2 Process Flow

Patient Registration Flow

```
User fills form
      │
      ▼
Frontend validates input
      │
      ▼
POST /api/patients (JSON payload)
      │
      ▼
Flask validates required fields
      │
      ├── Validation fails ──► 400 Bad Request ──► Show error to user
      │
      ▼
Check email uniqueness in DB
      │
      ├── Duplicate email ──► 400 Error ──► "Email already exists"
      │
      ▼
Create Patient record (SQLAlchemy)
      │
      ▼
Commit to PostgreSQL
      │
      ▼
201 Created + patient data (JSON)
      │
      ▼
Frontend refreshes patient list
      │
      ▼
Show success message to user
```

Appointment Booking Flow

```
User selects patient + doctor + datetime
      │
      ▼
POST /api/appointments
```

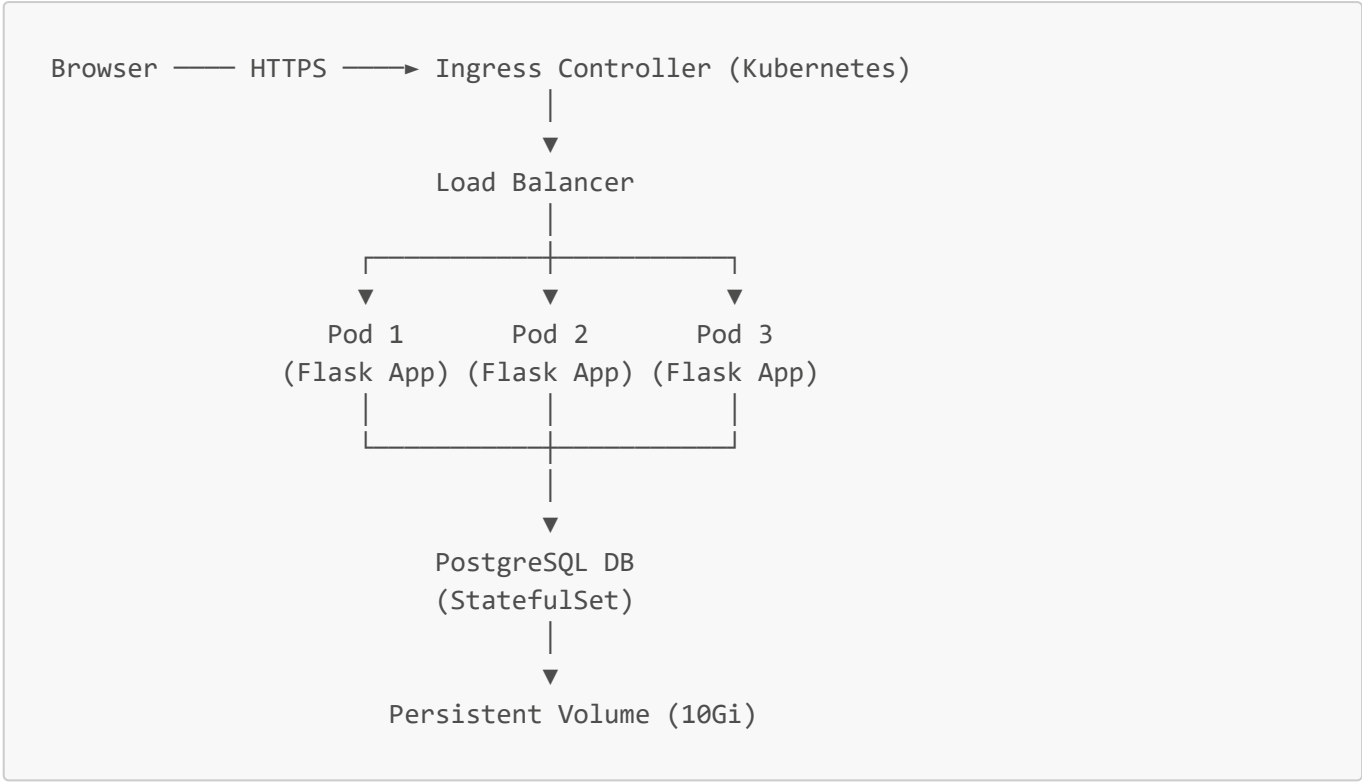
```
      │
      ▼
Validate patient_id exists in DB
      │
      ├── Patient not found ──► 404 Not Found
      │
      ▼
Parse datetime string
      │
      ▼
Create Appointment record
      │
      ▼
201 Created + appointment data
      │
      ▼
Frontend refreshes appointments list
```

Delete Flow (Patient / Appointment / Record)

```
User clicks Delete button
      │
      ▼
Confirmation dialog shown
      │
      ├── User cancels ──► No action
      │
      ▼
DELETE /api/{resource}/{id}
      │
      ▼
Flask fetches record by ID
      │
      ├── Not found ──► 404 Not Found
      │
      ▼
db.session.delete(record)
      │
      ▼
db.session.commit()
      │
      ▼
200 OK + success message
      │
      ▼
Frontend refreshes list
```

2.3 Information Flow

Request/Response Information Flow



Data Flow Diagram

Source	Data	Destination	Protocol
Browser	Form data (JSON)	Flask API	HTTP POST
Flask API	SQL INSERT	PostgreSQL	TCP/SQL
PostgreSQL	Result rows	Flask API	TCP/SQL
Flask API	JSON response	Browser	HTTP 200/201
Docker Health Check	HTTP GET /health	Flask API	HTTP
GitHub Actions	Docker image	Docker Registry	HTTPS
Kubernetes	Pull image	Docker Registry	HTTPS

2.4 Components Design

Backend Components

Component	File	Responsibility
App Factory	`app.py`	Flask app creation, route registration, DB init
Data Models	`models.py`	SQLAlchemy ORM models, relationships
Configuration	`config.py`	Environment-based config (dev/prod/test)
Patient Routes	`app.py`	CRUD endpoints for patients

Component	File	Responsibility
Appointment Routes	`app.py`	CRUD endpoints for appointments
Record Routes	`app.py`	CRUD endpoints for medical records
Health Endpoints	`app.py`	`/health` and `/ready` probes

Frontend Components

Component	File	Responsibility
App Root	`App.jsx`	State management, tab routing, API calls
API Service	`services/api.js`	Axios HTTP client, all API methods
Patient Form	`App.jsx`	Add new patient form
Appointment Form	`App.jsx`	Book appointment form
Record Form	`App.jsx`	Add medical record form
Status Bar	`App.jsx`	System health indicator
Data Cards	`App.jsx`	Display lists with delete buttons

Infrastructure Components

Component	Technology	Purpose
Web Server	Gunicorn (4 workers)	WSGI server for Flask
Reverse Proxy	NGINX Ingress	TLS termination, routing
Container Runtime	Docker	Application packaging
Orchestration	Kubernetes	Scaling, self-healing
Secret Store	K8s Secrets	Credentials management
Config Store	K8s ConfigMap	Environment configuration

2.5 Key Design Considerations

1. Stateless API Design

The Flask API is **completely stateless** — no server-side sessions. Each request contains all necessary information. This enables:

- Horizontal scaling (multiple pods)
- Load balancing without sticky sessions
- Easy failover

2. Database Connection Pooling

SQLAlchemy manages a **connection pool** to PostgreSQL:

- Pool size: 10 connections (default)
- Max overflow: 20
- Pool timeout: 30 seconds
- Prevents connection exhaustion under load

3. Graceful Error Handling

All API endpoints implement:

- Try/except blocks with rollback on failure
- Structured JSON error responses
- HTTP status codes (400, 404, 500)
- Detailed server-side logging

4. CORS Configuration

Flask-CORS enabled for:

- Development: `http://localhost:5173`
- Production: Configured via environment variable
- Prevents unauthorized cross-origin requests

5. Container Security

- Non-root user (`appuser`) in Docker container
- Read-only filesystem where possible
- Minimal base image (`python:3.11-slim`)
- No secrets in Docker image layers

6. Multi-Stage Docker Build

```
Stage 1 (Builder):          Stage 2 (Runtime):
python:3.11-slim            python:3.11-slim
+ gcc, build tools          + libpq5 only
+ pip install deps          + copy venv from Stage 1
+ compile Python files      + copy app code
                            + non-root user
                            → Final image: ~180MB
```

2.6 API Catalogue

Health Endpoints

Method	Endpoint	Description	Response

Method	Endpoint	Description	Response
GET	`/health`	Basic health check	`{"status":"healthy"}`
GET	`/ready`	DB connectivity check	`{"status":"ready","database":"connected"}`

Patient Endpoints

Method	Endpoint	Description	Request Body	Response
GET	`/api/patients`	List all patients	-	`{success, data[], count}`
GET	`/api/patients/<id>`	Get patient by ID	-	`{success, data}`
POST	`/api/patients`	Create patient	`{name, email, date_of_birth, phone?, address?, blood_group?}`	`201 {success, data}`
PUT	`/api/patients/<id>`	Update patient	`{name?, email?, phone?, ...}`	`{success, data}`
DELETE	`/api/patients/<id>`	Delete patient	-	`{success, message}`

Appointment Endpoints

Method	Endpoint	Description	Request Body	Response
GET	`/api/appointments`	List all appointments	-	`{success, data[], count}`
POST	`/api/appointments`	Create appointment	`{patient_id, doctor_name, appointment_datetime, reason?}`	`201 {success, data}`
PUT	`/api/appointments/<id>`	Update appointment	`{status?, notes?, appointment_datetime?}`	`{success, data}`
DELETE	`/api/appointments/<id>`	Delete appointment	-	`{success, message}`

Medical Record Endpoints

Method	Endpoint	Description	Request Body	Response

Method	Endpoint	Description	Request Body	Response
GET	`/api/patients/<id>/records`	Get patient records	-	`{success, data[], patient}`
POST	`/api/records`	Create medical record	`{patient_id, diagnosis, doctor_name, prescription?, record_date?}`	`201 {success, data}`
DELETE	`/api/records/<id>`	Delete medical record	-	`{success, message}`

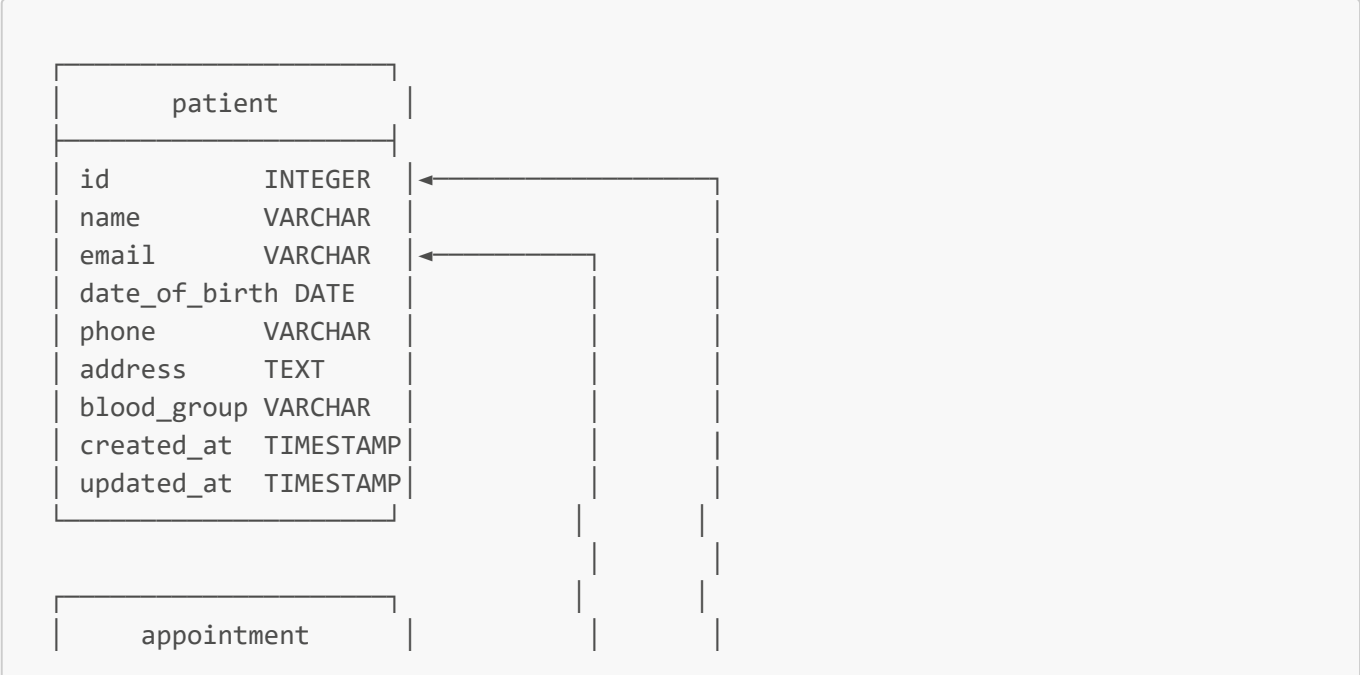
HTTP Status Codes Used

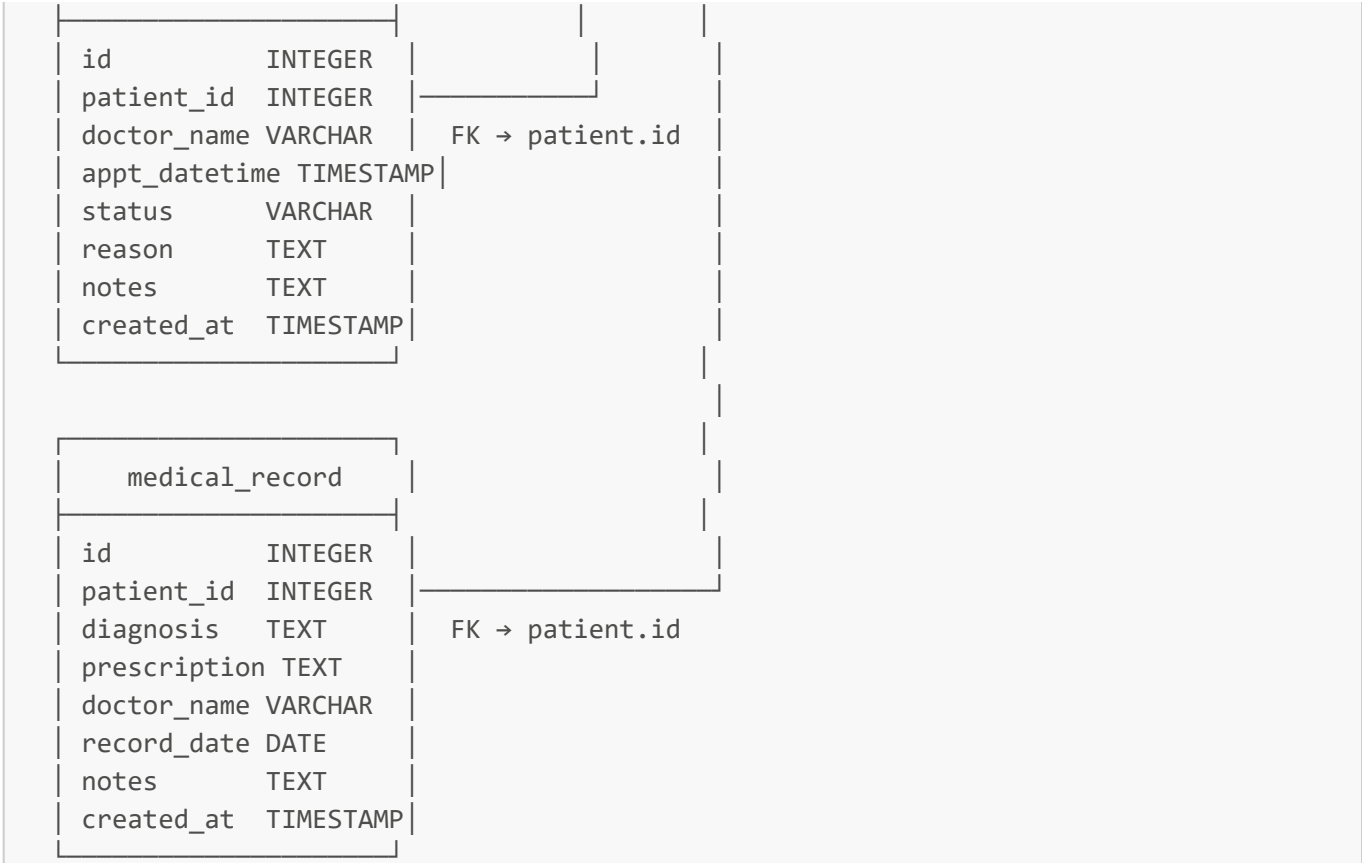
Code	Meaning	When Used
200	OK	Successful GET, PUT, DELETE
201	Created	Successful POST
400	Bad Request	Validation failure, missing fields
404	Not Found	Resource doesn't exist
500	Internal Server Error	Unexpected server error

3. Data Design

3.1 Data Model

Entity Relationship Diagram





Relationships

Relationship	Type	Cascade
Patient → Appointments	One-to-Many	DELETE CASCADE
Patient → Medical Records	One-to-Many	DELETE CASCADE

Table Definitions

patient

```
CREATE TABLE patient (
    id          SERIAL PRIMARY KEY,
    name        VARCHAR(100) NOT NULL,
    email       VARCHAR(120) UNIQUE NOT NULL,
    date_of_birth DATE NOT NULL,
    phone       VARCHAR(20),
    address     TEXT,
    blood_group VARCHAR(5),
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW()
);
```

appointment

```
CREATE TABLE appointment (
    id                  SERIAL PRIMARY KEY,
    patient_id          INTEGER NOT NULL REFERENCES patient(id) ON DELETE
CASCADE,
    doctor_name         VARCHAR(100) NOT NULL,
    appointment_datetime TIMESTAMP NOT NULL,
    status              VARCHAR(20) DEFAULT 'scheduled',
    reason              TEXT,
    notes               TEXT,
    created_at          TIMESTAMP DEFAULT NOW()
);
```

medical_record

```
CREATE TABLE medical_record (
    id          SERIAL PRIMARY KEY,
    patient_id  INTEGER NOT NULL REFERENCES patient(id) ON DELETE CASCADE,
    diagnosis   TEXT NOT NULL,
    prescription TEXT,
    doctor_name VARCHAR(100) NOT NULL,
    record_date DATE NOT NULL,
    notes       TEXT,
    created_at  TIMESTAMP DEFAULT NOW()
);
```

3.2 Data Access Mechanism

ORM Layer: SQLAlchemy 3.1.1

The application uses the **Repository Pattern** via SQLAlchemy ORM:

```
Application Code
      │
      ▼
SQLAlchemy ORM (Object Mapping)
      │
      ▼
psycopg2 Driver (Python-PostgreSQL adapter)
      │
      ▼
PostgreSQL 15 Database
```

Access Patterns:

Operation	SQLAlchemy Method	SQL Generated

Operation	SQLAlchemy Method	SQL Generated
Get all	`Model.query.all()`	`SELECT * FROM table`
Get by ID	`Model.query.get_or_404(id)`	`SELECT * FROM table WHERE id=?`
Filter	`Model.query.filter_by(...)`	`SELECT * FROM table WHERE ...`
Create	`db.session.add(obj); db.session.commit()`	`INSERT INTO table ...`
Update	`obj.field = value; db.session.commit()`	`UPDATE table SET ...`
Delete	`db.session.delete(obj); db.session.commit()`	`DELETE FROM table WHERE id=?`

Transaction Management:

- All write operations wrapped in try/except
- `db.session.rollback()` on any exception
- Atomic commits ensure data consistency

3.3 Data Retention Policies

Data Type	Retention Period	Policy
Patient Records	Indefinite	Retained until explicitly deleted
Appointments	Indefinite	Retained for audit/history
Medical Records	Indefinite	Legal requirement — permanent
Application Logs	30 days	Rotated automatically
Database Backups	90 days	Automated daily backups (planned)
Audit Logs	1 year	Compliance requirement (planned)

Future Enhancements:

- Soft delete (mark as deleted, don't remove)
- Automated archival of old appointments
- GDPR-compliant data purge workflows

3.4 Data Migration

Current Approach: SQLAlchemy `db.create_all()`

- Tables created automatically on first startup
- No manual SQL scripts required
- Idempotent — safe to run multiple times

Production Migration Strategy (Flask-Migrate):

```
# Initialize migrations
flask db init

# Create migration script
flask db migrate -m "Add new column"

# Apply migration
flask db upgrade

# Rollback if needed
flask db downgrade
```

Migration Principles:

- All migrations are version-controlled in `migrations/` folder
- Backward-compatible changes preferred
- Blue-green deployment for zero-downtime migrations
- Database backups taken before each migration

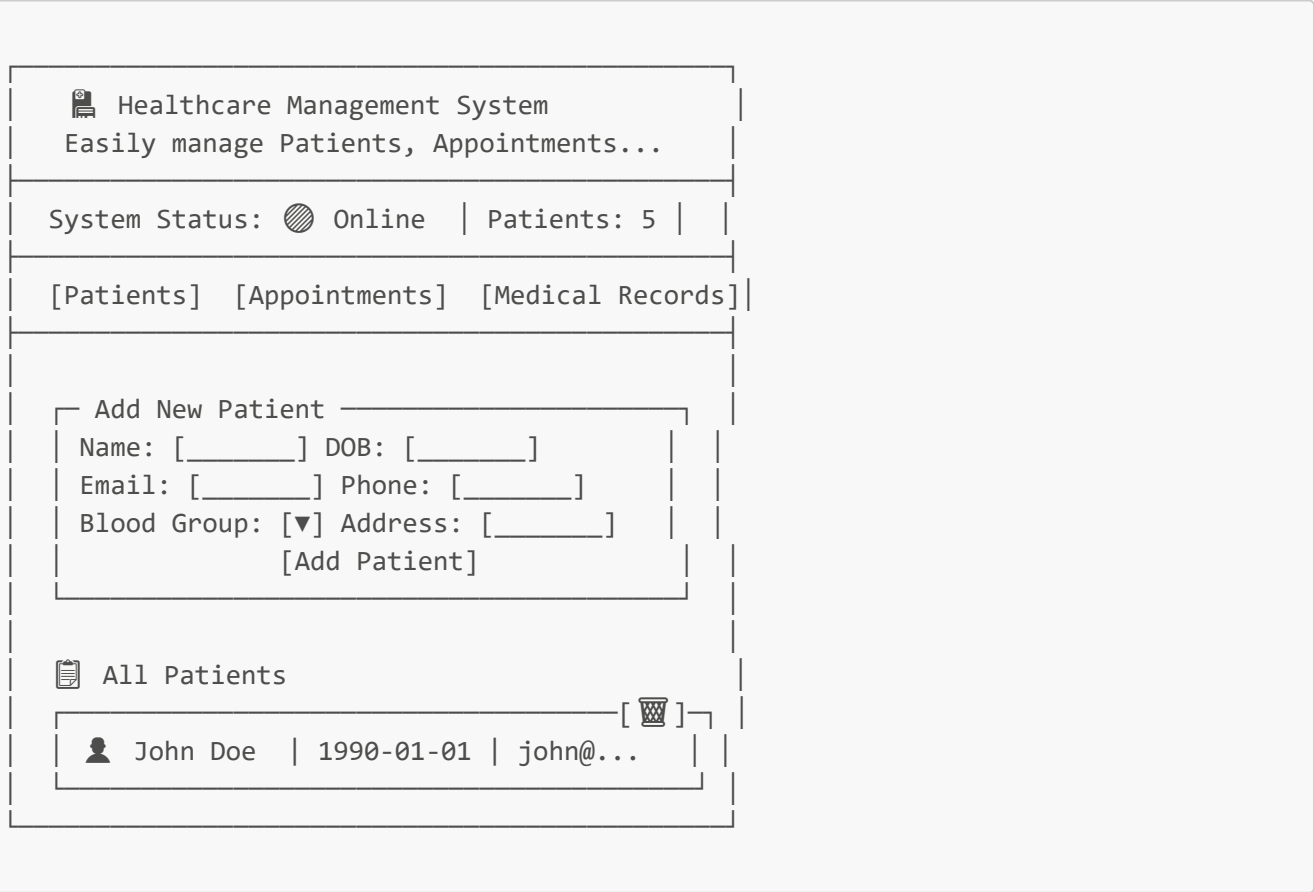
---

4. Interfaces

4.1 User Interface

Technology: React 18 + Tailwind CSS 3.4 + Vite 7

UI Layout:



UI Components:

Component	Description
Header	App title and description
Status Bar	System health, patient count, appointment count
Tab Navigation	Switch between Patients / Appointments / Records
Form Panels	Add new entity forms with validation
Data Cards	Display list items with delete button
Message Banner	Success/error notifications (auto-dismiss 5s)
Loading Spinner	Shown during API calls

4.2 API Interface

Protocol: REST over HTTP/HTTPS
Data Format: JSON
Content-Type: `application/json`
CORS: Enabled for frontend origin

Standard Response Format:

```
{
  "success": true,
  "data": { ... },
  "message": "Operation successful",
  "count": 10
}
```

Error Response Format:

```
{
  "success": false,
  "error": "Descriptive error message"
}
```

4.3 Database Interface

Driver: psycopg2 (Python PostgreSQL adapter)
ORM: SQLAlchemy 3.1.1
Connection String: `postgresql://user:pass@host:5432/dbname`
Connection Pooling: SQLAlchemy built-in pool

4.4 External Interfaces

Interface	Type	Status
Docker Registry	Container image pull/push	Active
GitHub Actions	CI/CD trigger	Active
Kubernetes API	Deployment management	Active
SMTP (Email)	Appointment notifications	Planned
SMS Gateway	Patient reminders	Planned

---

5. State and Session Management

5.1 Frontend State Management

Technology: React `useState` and `useEffect` hooks (no Redux)

State Variables:

State	Type	Description
`activeTab`	string	Current active tab (patients/appointments/records)
`systemStatus`	string	Backend health (online/offline/checking)
`patients`	array	List of all patients
`appointments`	array	List of all appointments
`records`	array	List of all medical records
`loading`	boolean	API call in progress
`message`	object	Success/error notification
`patientForm`	object	Patient form field values
`appointmentForm`	object	Appointment form field values
`recordForm`	object	Medical record form field values

State Lifecycle:

```
Component Mount
      │
      ▼
useEffect → loadData() based on activeTab
      │
      ▼
API call → setLoading(true)
      │
      ▼
Response → setState(data) → setLoading(false)
      │
```

```
      ▼
Re-render with new data
```

5.2 Backend Session Management

The Flask API is **completely stateless**:

- No server-side sessions
- No cookies
- No JWT tokens (planned for Phase 2)
- Each request is independent and self-contained

Health Check Polling:

- Frontend polls `/health` every **30 seconds**
- Updates `systemStatus` state (online/offline)
- Provides real-time system status to users

---

6. Caching

6.1 Current Caching Strategy

Browser-Level Caching:

- Vite build generates hashed filenames for cache busting
- Static assets cached by browser (CSS, JS, images)
- API responses: **No caching** (always fresh data)

Database Query Caching:

- SQLAlchemy identity map (within a session)
- No explicit query result caching currently

6.2 Planned Caching Strategy (Phase 2)

Redis Cache Layer:

Data	Cache Duration	Invalidation
Patient list	60 seconds	On create/update/delete
Individual patient	5 minutes	On update/delete
Appointment list	30 seconds	On create/update/delete
System health	10 seconds	Automatic TTL

Cache Architecture (Planned):

```
React Frontend
      │
      ▼
Flask API
      │
      ├──► Redis Cache (HIT) ──► Return cached data
      │
      └──► PostgreSQL (MISS) ──► Cache result ──► Return data
```

Cache Invalidation Strategy:

- Write-through: Update cache on every write
- TTL-based: Automatic expiry for time-sensitive data
- Event-driven: Invalidate on data mutation

---

7. Non-Functional Requirements

7.1 Security Aspects

Authentication & Authorization

Aspect	Current	Planned (Phase 2)
Authentication	None	JWT Bearer tokens
Authorization	None	Role-based (Admin/Doctor/Staff)
Password Storage	N/A	bcrypt hashing
Token Expiry	N/A	1 hour (access) / 7 days (refresh)

Data Security

Input Validation:

- Required field validation on all POST endpoints
- Data type validation (dates, emails)
- SQL injection prevention via SQLAlchemy ORM (parameterized queries)
- XSS prevention via React's JSX escaping

Transport Security:

- HTTPS enforced at Kubernetes Ingress
- TLS 1.2+ only
- HTTP → HTTPS redirect
- HSTS headers

Container Security:

```
☑ Non-root user (appuser)
☑ Minimal base image (python:3.11-slim)
☑ No secrets in image layers
☑ Read-only root filesystem (planned)
☑ Security scanning via Trivy in CI/CD
☑ No privileged containers
```

Secrets Management:

- Database credentials in Kubernetes Secrets
- Environment variables (never hardcoded)
- `.env.example` provided (never commit `.env`)
- Secret rotation supported via K8s Secret update

Network Security:

```
Internet ──► Ingress (NGINX) ──► Service (ClusterIP)
                                        │
                                Only internal traffic
                                between pods and DB
```

OWASP Top 10 Mitigations

Risk	Mitigation
Injection	SQLAlchemy ORM (parameterized queries)
Broken Auth	JWT planned; HTTPS enforced
Sensitive Data Exposure	HTTPS, no PII in logs
Security Misconfiguration	Docker non-root, minimal image
XSS	React JSX auto-escaping
CSRF	Stateless API (no cookies)
Vulnerable Components	Trivy scanning in CI/CD

7.2 Performance Aspects

Response Time Targets

Operation	Target	Current
GET /health	< 50ms	~10ms
GET /api/patients	< 200ms	~50ms
POST /api/patients	< 300ms	~80ms

Operation	Target	Current
DELETE /api/patients	< 200ms	~60ms
Page load (frontend)	< 2s	~1s

Throughput Targets

Metric	Target
Concurrent users	100+
Requests per minute	1,000+
Database connections	30 (pooled)
API availability	99.9% uptime

Scalability Design

Horizontal Scaling (Kubernetes):

```
Load Balancer
      │
      ├──► Flask Pod 1 (CPU: 100m-500m, RAM: 128Mi-512Mi)
      ├──► Flask Pod 2 (CPU: 100m-500m, RAM: 128Mi-512Mi)
      └──► Flask Pod 3 (CPU: 100m-500m, RAM: 128Mi-512Mi)
```

Auto-scaling (HPA):

- Scale up: CPU > 70% for 2 minutes
- Scale down: CPU < 30% for 5 minutes
- Min replicas: 2
- Max replicas: 10

Database Performance:

- Indexes on: `patient.email` (unique), `appointment.patient_id`, `medical_record.patient_id`
- Connection pooling: 10 base + 20 overflow
- Query optimization via SQLAlchemy lazy loading

Gunicorn Configuration

```
Workers: 4 (2 × CPU cores + 1)
Worker class: sync
Timeout: 30 seconds
Max requests: 1000 (auto-restart to prevent memory leaks)
Bind: 0.0.0.0:5000
```

8. References

Internal Documents

Document	Description
LLD_PRESENTATION.md	Low Level Design document
UML_DIAGRAMS.puml	PlantUML architecture diagrams
PROJECT_DOCUMENTATION.md	Complete project documentation
API.md	API reference documentation
README.md	Project setup and running instructions

Technology References

Technology	Documentation
Flask	https://flask.palletsprojects.com/
SQLAlchemy	https://docs.sqlalchemy.org/
React	https://react.dev/
Tailwind CSS	https://tailwindcss.com/docs
Docker	https://docs.docker.com/
Kubernetes	https://kubernetes.io/docs/
PostgreSQL	https://www.postgresql.org/docs/
Gunicorn	https://gunicorn.org/
GitHub Actions	https://docs.github.com/en/actions

Standards & Guidelines

Standard	Application
REST API Design	Richardson Maturity Model Level 2
HTTP Status Codes	RFC 7231
JSON Format	RFC 8259
OWASP Top 10	Security guidelines
12-Factor App	Application methodology
Semantic Versioning	Version numbering

Document prepared by: Healthcare Management System Team
Version: 1.0 | Date: February 2026
Classification: Internal