



Swasthya: India's Decentralized Health Intelligence Network

Version License Tech

A comprehensive, full-stack **AI agent system** for hospital operations management at a national scale. Swasthya combines decentralized data infrastructure with advanced AI to transform healthcare delivery in India, featuring intelligent orchestration, ML-powered predictions, and hybrid cloud deployment capabilities.

Problem Statement

India's healthcare system faces several critical challenges that Swasthya aims to address:

- **Fragmented Patient Records:** Medical histories are siloed across providers, leading to gaps in information and continuity of care.
- **Slow Emergency Response:** Emergency rooms often suffer long wait times and suboptimal triage, delaying critical interventions.
- **Manual & Error-Prone Processes:** Key operations like staff scheduling and discharge planning rely on manual workflows, resulting in inefficiencies and mistakes.
- **Lack of Real-Time Insights:** Hospitals have limited real-time monitoring of patient vitals and facility data, hindering proactive decision-making.
- **Data Privacy & Access Barriers:** Sharing health data between facilities is difficult, balancing patient privacy (especially under regulations) with the need for collaborative intelligence.

Vision

Swasthya envisions a **decentralized health intelligence network** that connects hospitals across India to overcome these challenges. The project's vision is to leverage cutting-edge technology so that **each hospital operates as part of a smarter, interconnected system** while respecting data sovereignty. Key elements of this vision include:

- **Digital Health Wallets (Blockchain + Aadhaar):** Every patient holds a secure, blockchain-based health record wallet linked to their Aadhaar (national ID), enabling portable and tamper-proof medical records. Authorized providers can access and update records on-chain, ensuring continuity of care across the country.
- **Real-Time Monitoring (IoT Devices):** Hospitals integrate IoT-enabled devices (wearables, bedside monitors) to continuously stream patient vital signs and sensor data. This provides live insights for early warning alerts and proactive interventions, transforming reactive care into **continuous, data-driven care**.
- **AI-Driven Decision Support (Multi-Agent AI Command Center):** A suite of AI agents (for forecasting, triage, scheduling, etc.) act as an "AI Command Center," orchestrated by a central

Supervisor. These agents employ advanced techniques – from time-series DL models to reinforcement learning and natural language understanding – to optimize operations. *For example, the system is designed to incorporate medical imaging AI (NVIDIA Clara) for diagnostics and large language models (Meta's LLaMA/Code Llama) for reasoning and explanations, bringing **state-of-the-art AI** to bedside decisions.*

- **Privacy-Preserving Collaboration (Federated Learning):** Instead of centralizing sensitive patient data, the network uses federated learning to train AI models across hospitals. Insights (model updates) are shared without raw data leaving local servers, aligning with privacy laws and **empowering hospitals in remote areas** to benefit from collective intelligence. This decentralized AI training complements the blockchain data layer to realize a secure, nationwide learning healthcare system.

Through these pillars, Swasthya's mission is to **dramatically improve healthcare delivery** – ensuring that no matter where a patient is in India, their data is available when needed, emergencies are handled with optimal speed, operations run efficiently, and care benefits from the latest AI innovations.

Overview

Swasthya is an advanced **AI-driven multi-agent system** designed to optimize hospital operations through a decentralized architecture. The system combines a TypeScript-based orchestrator (supervisory control) with Python AI agents to handle critical workflows – including patient triage, staff scheduling, demand forecasting, and discharge planning – in an intelligent, automated manner. It is built with India's healthcare ecosystem in mind, integrating with national digital infrastructure (e.g. Aadhaar) and supporting multi-hospital collaboration.

System Statistics

- **Total Services:** 16 containerized microservices (Docker-based)
- **AI Agents:** 6 specialized ML/optimization agents (see below)
- **Languages:** TypeScript (Node.js orchestrator) + Python (ML/optimization agents)
- **API Endpoints:** 50+ RESTful endpoints across services
- **Database Tables:** 8 main tables + 2 views (PostgreSQL for operational data)
- **Federated Learning:** Privacy-preserving multi-hospital training support (Flower framework)
- **Dataset Support:** Sample synthetic datasets included for FL training (demand & triage)

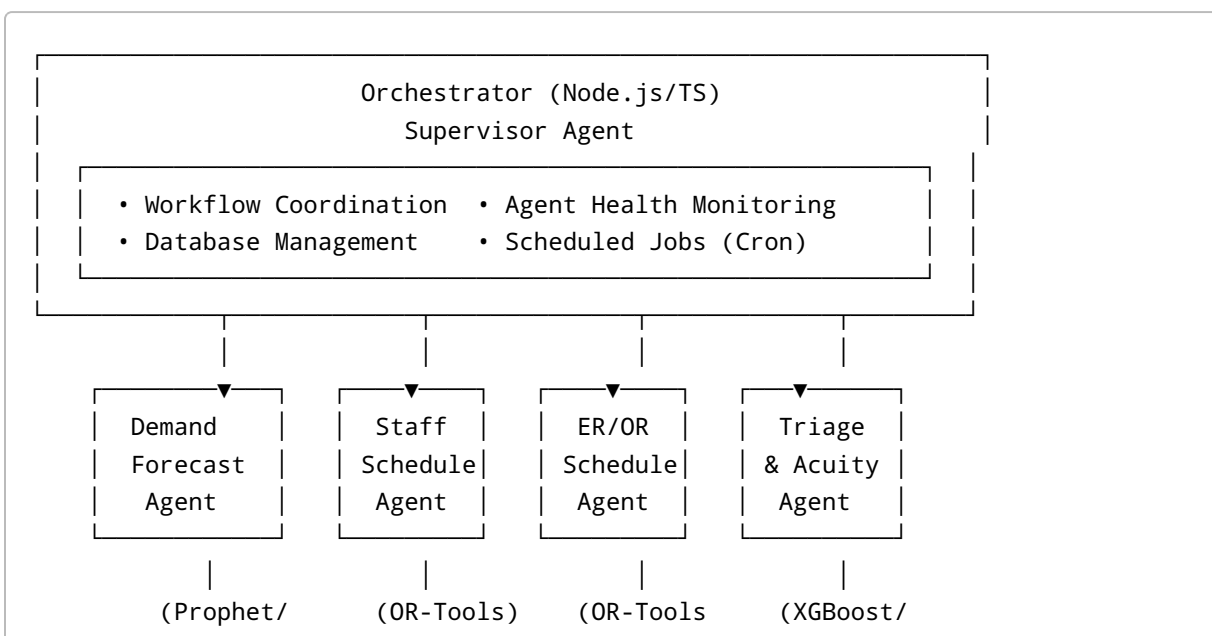
Key Features

- **Multi-Agent Architecture** – Seven modular AI components (six agents + one orchestrator) cooperate to manage different hospital functions, each specialized but centrally coordinated. This separation of concerns enables scalable development and deployment of each service.
- **MLflow Integration** – Complete ML lifecycle management with experiment tracking and a model registry. All model training runs (forecasts, predictive models, etc.) are logged to an MLflow server, ensuring reproducibility and traceability of AI performance.
- **Hybrid Deployment** – Supports flexible deployment topologies: model training and heavy computations can run on cloud GPUs, while inference and data storage remain on-premises at the hospital. This hybrid cloud design balances performance with data locality (crucial for compliance).

- **Federated Learning** – Privacy-preserving collaborative model training across hospitals using Flower. Multiple hospitals can train a shared model (for demand forecasting, triage risk stratification, etc.) without sharing sensitive patient data, enhancing insights on a national scale **while meeting privacy regulations**.
- **Real-Time Operations** – Event-driven and scheduled workflows enable real-time responsiveness. For example, the system auto-triages new ER arrivals and reprioritizes OR schedules dynamically if emergencies arise, addressing the problem of slow emergency response with AI-driven agility.
- **Healthcare-Specific Design** – Built for hospital workflows with compliance in mind (e.g. HIPAA/ GDPR). Domain-specific logic (clinical rules, acuity scales, HL7/FHIR integration) is embedded alongside AI to ensure recommendations are safe, interpretable, and align with medical protocols.
- **Containerized Microservices** – Fully Dockerized deployment with each agent and component isolated in its own container. This makes the system easy to deploy and scale on different environments (local, cloud, or Kubernetes). Versioned container images for all 16 services are defined for reproducibility.
- **Comprehensive Documentation** – Each service provides OpenAPI/Swagger docs for its REST API, and the repository includes detailed guides (setup, dev, usage). Inline code comments and a clear project structure (see below) facilitate onboarding new developers.
- **Decentralized Health Records (Blockchain)** – *Planned*: Integration of an Ethereum-based **health wallet** for patients. Health records will be stored on a blockchain ledger, linked via Aadhaar for identity, enabling secure sharing of records across facilities with patient consent. This adds trust and interoperability to medical data exchange.
- **IoT-Enabled Vitals Monitoring** – *Planned*: Continuous monitoring of patient vitals through connected IoT devices. Vital signs (heart rate, blood pressure, etc.) feed into the platform in real time. The AI agents (e.g., triage) can then trigger alerts or adjust risk scores immediately, bringing **real-time insight** to clinical care.

System Architecture

High-Level Architecture




```

// In practice, include access control checks
records[aadhaarId].push(Record(dataHash, msg.sender));
}

function getRecords(string memory aadhaarId) public view returns (Record[]
memory) {
    return records[aadhaarId];
}
}

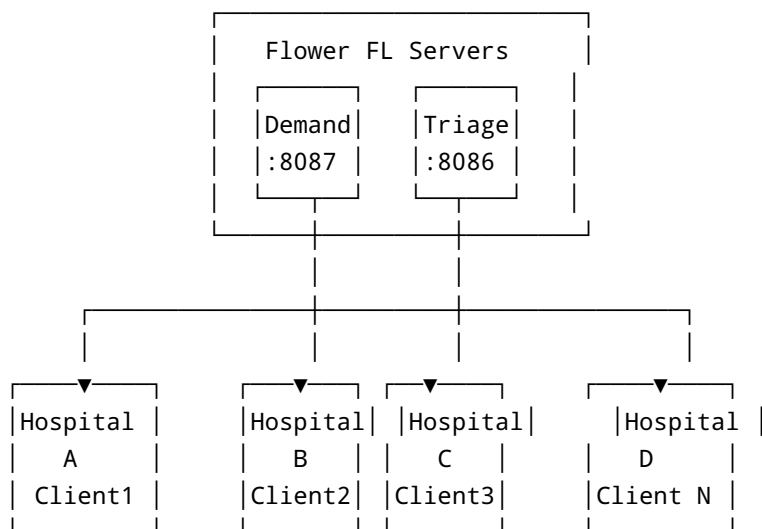
```

Example: In a real deployment, `dataHash` could reference an off-chain encrypted health record (e.g., IPFS hash of a medical PDF) to keep on-chain data minimal. This blockchain layer (if enabled) would interface with the Orchestrator or a dedicated service to read/write patient data.

- **IoT Real-Time Monitoring:** The architecture is designed to ingest streaming data from IoT devices. For example, wearable vital monitors or smart ICU devices can push patient telemetry (heart rate, SpO₂, ECG, etc.) into the system (e.g., via an IoT gateway service). The **Triage & Acuity agent** can then analyze these real-time signals to update patient risk scores or trigger alerts if thresholds are crossed. Likewise, an **alerts module** (future addition) in the Orchestrator could notify staff or re-prioritize the ER queue when a monitored metric indicates deterioration. This pillar brings live situational awareness to hospital operations, complementing the periodic updates from human staff.

(Both the blockchain and IoT integration are in the project roadmap – see Planned Enhancements – highlighting our commitment to a decentralized, real-time health network. If these features are not active in the current release, the documentation retains them as part of the system vision.)

Federated Learning Architecture



Local Data (Never Shared)	Local Data (Private)	Local Data (Private)	Local Data (Private)
------------------------------	-------------------------	-------------------------	-------------------------

In addition to the core agents, Swasthya includes a **Federated Learning (FL)** subsystem to enable multi-hospital AI collaboration. Two FL server services coordinate distributed training: one for the demand forecasting models and one for the triage acuity models. Hospitals run lightweight FL client instances that train on local data and send model updates to the server, which aggregates them and updates a global model.

- **Federated Learning Servers:**

- `fl-demand-server` (port 8087) – Coordinates federated training rounds for demand forecasting (e.g., aggregating ARIMA or other model parameters from each hospital).
- `fl-triage-server` (port 8086) – Coordinates federated training for triage acuity classification (aggregating XGBoost model updates from hospitals).

- **Federated Learning Clients:**

- `fl-demand-client-{1,2,3}` – Simulated hospital clients that train a demand forecasting model (initially ARIMA) on their local patient volume data.
- `fl-triage-client-{1,2,3}` – Simulated clients that train a triage classifier (XGBoost) on their local ER datasets.

Each FL server orchestrates a series of training **rounds**: clients train on local data, send model gradients/weights, and the server uses a custom strategy (e.g., best model selection) to aggregate updates into a global model. By default, 5 rounds are run per session (configurable in code).

Key Features of FL:

- **Privacy-Preserving:** Raw patient data never leaves the hospital premises – only model parameters or gradients are shared, and these can be encrypted or anonymized. This addresses data privacy regulations while still allowing collaborative learning.
- **Collaborative Modeling:** Hospitals with relatively small datasets can collectively train a more robust model than each could alone. For instance, demand forecasting improves with data from multiple regions/seasons.
- **Adaptive Rounds & Strategies:** The system uses a *BestModelStrategy* – after each round, it evaluates models from all clients and selects the best-performing one as the global model (rather than a simple average). Metrics used include negative RMSE for demand (lower is better) and accuracy for triage (higher is better).
- **Seamless Integration:** The trained global models from FL can be exported and loaded into the corresponding main agents. Both the `demand_forecast` and `triage_acuity` agents have an `fl_run_client.py` module that allows them to directly participate in federated training or pull the latest federated model for inference.

Dataset Requirements: (for running FL with provided data)

- **Demand Forecasting:** Each client expects a CSV file like `demand_client_X.csv` with historical daily patient volumes (`date, volume` format). Example provided for clients 1-3 in the `datasets/` folder.

- **Triage Classification:** Each client uses `triage_client_X.csv` with patient features and an `acuity_label` (1-5) column indicating triage level. This simulates labeled triage data at each hospital.

Usage: To experiment with FL, start the servers and clients (either via Docker Compose or manually):

```
# Start the full FL stack with Docker Compose (already included in main compose file)
docker-compose up -d

# Or run servers manually for development:
python -m federated_learning.server.demand_server # Runs on 8087
python -m federated_learning.server.triage_server # Runs on 8086

# (In separate terminals) start one or more clients:
python -m federated_learning.client.demand_client --cid 1 --server-address localhost:8087
python -m federated_learning.client.triage_client --cid 1 --server-address localhost:8086

# Monitor training progress via logs:
docker-compose logs -f fl-demand-server # or fl-triage-server
# Each round: clients train locally → send updates → server aggregates → repeat
```

Federated Learning Module Structure:

```
federated_learning/
├── server/
│   ├── demand_server.py    # Demand FL orchestrator
│   ├── triage_server.py    # Triage FL orchestrator
│   └── strategy.py         # "BestModelStrategy" aggregation logic
├── client/
│   ├── demand_client.py    # Demand client training logic
│   ├── triage_client.py    # Triage client training logic
│   └── serde.py            # Serialization utilities for parameters
├── Dockerfile.server       # FL server container image
├── Dockerfile.client       # FL client container image
├── requirements-server.txt  # Dependencies for server (Flower, NumPy, sklearn, etc.)
└── requirements-client.txt  # Dependencies for clients (Flower, pandas, xgboost, etc.)
```

The FL components are optional but provide a powerful way to utilize data from multiple facilities. They integrate with the main agents to form the **decentralized intelligence network** aspect of Swasthya.

Project Structure

```
swasthya-MumbaiHacks-Project/
├── agents/                                # ML Agent Services (Python)
│   ├── demand_forecast/                 # Time-series forecasting agent
│   │   ├── api.py                       # FastAPI service endpoints
│   │   ├── model.py                     # Forecasting models (Prophet/ARIMA)
│   │   ├── train.py                     # Training routine
│   │   ├── mlflow_tracking.py           # MLflow experiment logging
│   │   ├── fl_run_client.py             # Federated learning client integration
│   │   ├── config.py                    # Configurations
│   │   └── Dockerfile + requirements.txt
│   ├── staff_scheduling/                 # Staff scheduling optimizer agent
│   │   ├── api.py                       # FastAPI service
│   │   ├── scheduler.py                 # Constraint solver (CP-SAT via OR-Tools)
│   │   ├── config.py
│   │   └── Dockerfile + requirements.txt
│   ├── eror_scheduling/                 # ER/OR scheduling agent (Emergency & OR)
│   │   ├── api.py                       # FastAPI service
│   │   ├── scheduler.py                 # ER queue and OR block scheduling logic
│   │   ├── config.py
│   │   └── Dockerfile + requirements.txt
│   ├── discharge_planning/              # Discharge readiness predictor agent
│   │   ├── api.py                       # FastAPI service
│   │   ├── model.py                     # XGBoost model and rules
│   │   ├── config.py
│   │   └── Dockerfile + requirements.txt
│   └── triage_acuity/                   # Patient triage classifier agent
│       ├── api.py                       # FastAPI service
│       ├── model.py                     # XGBoost classification model
│       ├── text_parser.py               # NLP for symptom parsing
│       ├── fl_run_client.py             # Federated learning client integration
│       ├── config.py
│       └── Dockerfile + requirements.txt
├── orchestrator/                         # Central Orchestrator service (Node.js/TS)
│   ├── src/
│   │   ├── agents/                     # Agent API clients (for orchestrator to
│   │   │   └── call agents)
│   │   ├── config/                     # Configuration modules (env, constants)
│   │   ├── database/                   # Database access layer (TypeORM or Knex)
│   │   ├── middleware/                 # Express middleware (auth, etc.)
│   │   └── routes/                     # Express API routes (e.g., /api/forecast, /
│   │   └── api/triage)
│   └── scheduler/                       # Cron job definitions (daily tasks, etc.)
```

```

|   |   |— supervisor/           # Core orchestration logic (triggering
agents, etc.)
|   |   |   |— utils/           # Utility functions
|   |   |   |— package.json
|   |   |   |— tsconfig.json
|
|— federated_learning/          # Federated Learning module (Python/Flower)
|   |— server/                  # FL server implementations
|   |   |— demand_server.py     # Coordinates global demand model training
|   |   |— triage_server.py     # Coordinates global triage model training
|   |   |— strategy.py          # Custom aggregation strategy
(BestModelStrategy)
|   |— client/                  # FL client implementations
|   |   |— demand_client.py     # Local training for demand forecasting
|   |   |— triage_client.py     # Local training for triage classification
|   |   |— serde.py             # Helper for (de)serializing model updates
|   |— Dockerfile.server
|   |— Dockerfile.client
|   |— requirements-server.txt
|   |— requirements-client.txt
|
|— data/                        # Database SQL initialization scripts
|   |— 01_create_databases.sql  # Creates MLflow tracking DB
|   |— init_database.sql        # Main schema (tables, views) + seed data
|
|— datasets/                    # Sample datasets for FL training
|   |— demand_client_1.csv
|   |— demand_client_2.csv
|   |— demand_client_3.csv
|   |— triage_client_1.csv
|   |— triage_client_2.csv
|   |— triage_client_3.csv
|   |— (etc. for additional clients or scenarios)
|
|— mlflow/                     # MLflow server configuration
|   |— Dockerfile              # Container for MLflow tracking UI
|
|— deployment/                 # (Reserved for deployment configs or
manifests)
|
|— docker-compose.yml           # Defines all 16 services and their
interconnections
|— env.example                  # Example environment variables file
|— .gitignore
|— README.md                    # Project documentation (this file)

```

Note: The directory `error_scheduling` stands for **ER/OR Scheduling** (Emergency Room / Operating Room). It is named as such in the codebase; throughout the documentation we refer to it clearly as the ER/OR Scheduling agent to avoid confusion.

The project structure above provides a roadmap of the codebase. Each agent has a dedicated directory encapsulating its service logic, which makes the system modular. This structure is especially useful for developers to navigate and understand where various functionalities reside. (For instance, all learning-related code for federated clients is co-located with the agents in `fl_run_client.py`, linking the FL module with the main agents.)

Quick Start

Getting Started

Recommended: Run the full system with Docker Compose (all components in containers). See installation steps below.

Prerequisites

For Docker Deployment (easiest way to run everything):

- Docker 20.10+
- Docker Compose 2.0+

For Local Development (alternative, for modifying code):

- Node.js 18+ (for the Orchestrator service)
- Python 3.10+ (for AI agents and FL services)
- PostgreSQL 15+ (optional if not using Docker for the database)

Installation

1. Clone the repository

```
git clone <repository-url>
cd swasthya
```

2. Configure environment variables

```
cp env.example .env
# Edit .env with your configuration as needed.
# (Default values are provided for a local Docker-based setup)
```

3. Prepare datasets for Federated Learning (optional)

If you plan to run the federated learning components, ensure the sample datasets are in place:

```
ls datasets/
# You should see:
# demand_client_1.csv, demand_client_2.csv, demand_client_3.csv
# triage_client_1.csv, triage_client_2.csv, triage_client_3.csv
#
# (If these files are missing, you can add your own or proceed;
# the FL services will simply idle if no data is available, without affecting
# core functionality.)
```

4. Start all services with Docker Compose

```
docker-compose up -d
```

5. Initialize the database (if not already auto-initialized)

```
# The PostgreSQL DB is set to initialize on first run via Docker.
# If needed, you can manually initialize the schema:
docker-compose exec postgres psql -U postgres -d swasthya_db -f /docker-
entrypoint-initdb.d/init.sql
```

6. Verify services are running

```
# Check that all containers are up (you should see 16 containers running):
docker-compose ps

# Core Infrastructure:
# - swasthya-postgres          (PostgreSQL database)
# - swasthya-mlflow            (MLflow tracking server)
# - swasthya-orchestrator      (Main orchestrator API)
#
# AI Agent Services (5):
# - swasthya-demand-forecast
# - swasthya-staff-scheduling
# - swasthya-eror-scheduling
# - swasthya-discharge-planning
# - swasthya-triage-acuity
#
# Federated Learning Services (8):
# - swasthya-fl-demand-server  + 3 clients
# - swasthya-fl-triage-server  + 3 clients
#
# Quick smoke tests:
curl http://localhost:3000      # Orchestrator (should return a basic
response)
```

```
curl http://localhost:5000          # MLflow UI (will return HTML or a redirect)
# FL servers have no direct HTTP endpoint; check their logs for activity:
docker-compose logs fl-demand-server | tail -n 20
docker-compose logs fl-triage-server | tail -n 20
```

Access Points

Once running, the following services are accessible:

Core Services:

- **Orchestrator API:** <http://localhost:3000>
- **MLflow UI:** <http://localhost:5000>

AI Agent Services (REST APIs):

- **Demand Forecast Agent:** <http://localhost:8001>
- **Staff Scheduling Agent:** <http://localhost:8002>
- **ER/OR Scheduling Agent:** <http://localhost:8003>
- **Discharge Planning Agent:** <http://localhost:8004>
- **Triage & Acuity Agent:** <http://localhost:8005>

Federated Learning Servers:

- **FL Triage Server:** <http://localhost:8086> (Note: no web UI; uses Flower protocols)
- **FL Demand Server:** <http://localhost:8087> (no direct UI; check logs or use Flower client)

Interactive API Documentation:

- Each agent provides Swagger/OpenAPI docs at its `/docs` endpoint for easy testing.
- Examples:
 - Demand Forecast API docs at <http://localhost:8001/docs>
 - Triage & Acuity API docs at <http://localhost:8005/docs>

Agent Details

Below is a breakdown of each intelligent agent in the Swasthya system, including their purpose, technology stack, features, and API interface. (All agents are implemented as **FastAPI** web services in Python, except the orchestrator which is a Node.js service.)

1. Demand Forecast Agent

Port: 8001 | **Technology:** Prophet (Facebook) / ARIMA / Moving Average

Predicts patient volumes (admissions, ER visits, etc.) to help with resource planning. This agent consumes historical patient arrival data and produces forecasts for future days or weeks.

- **Key Features:**

- Time-series forecasting with seasonality and trend decomposition.
- Multiple model support (Facebook Prophet for yearly/weekly seasonality, ARIMA for short-term trends, simple moving averages for baseline comparisons).
- Confidence intervals for all predictions, giving a range (upper/lower bound).
- MLflow integration for tracking forecast model performance over time.

Future: This agent is slated to incorporate advanced deep learning models for improved accuracy. A planned upgrade will utilize Meta's **Kats** library with PyTorch **Temporal Fusion Transformers (TFT)**, which can capture complex patterns and offer better interpretability than classic statistical models. (The current simpler Prophet/ARIMA approach was chosen for initial deployment speed; the roadmap includes migrating to TFT for state-of-the-art performance.)

- **API Endpoints:**

- `POST /predict` – Generate a new patient volume forecast. Expects historical data or uses default internal dataset; returns predicted volumes for upcoming days.
- `POST /train` – Trigger (re)training of the forecasting model on updated data. Useful if new data has been added to the system.
- `GET /health` – Health check endpoint for monitoring service availability.

2. Staff Scheduling Agent

Port: 8002 | **Technology:** Google OR-Tools (CP-SAT Solver)

Optimizes staff work schedules based on predicted demand and various constraints. The agent takes forecasted patient load and scheduling rules as input, and produces optimal shift schedules for doctors, nurses, and other staff.

- **Key Features:**

- **Constraint Programming Optimization:** Uses OR-Tools CP-SAT to solve scheduling as a constraint satisfaction problem, honoring rules like maximum shift length, rest requirements, skill-mix per shift, etc.
- **Fair Workload Distribution:** Ensures equitable allocation of shifts (e.g., night/weekend rotations are balanced among staff).
- **Coverage Guarantees:** Adheres to required staffing levels per department and shift (no understaffing).
- **Role-Based Assignments:** Accounts for staff roles/qualifications (e.g., at least one senior surgeon on call, or a certain nurse:patient ratio).

Future: Originally, this agent was envisioned as a **Reinforcement Learning** system that dynamically adjusts schedules in response to real-time changes, coupled with a language model to explain scheduling decisions in plain language. Those features are in development – the roadmap includes integrating an RL algorithm for adaptive re-scheduling (to handle last-minute call-outs or surges) and using an LLM (e.g., **Code LLaMA**)

to generate human-readable explanations for the suggested schedule (improving transparency for staff). For now, the agent uses a deterministic solver for stability.

- **API Endpoints:**
- `POST /schedule` – Generate an optimized schedule given the current staffing requirements and constraints. Returns a set of shifts/assignments.
- `GET /constraints/default` – Retrieve the default scheduling constraints and parameters (as a template or for UI purposes).

3. ER/OR Scheduling Agent

Port: 8003 | **Technology:** OR-Tools + ML (XGBoost for prediction)

Manages **Emergency Room triage queues** and **Operating Room scheduling** in tandem. It prioritizes ER patients by acuity and optimally allocates operating room slots for surgeries, coordinating these two high-pressure areas.

- **Key Features:**
- **Dynamic ER Queue Management:** Continuously reprioritizes ER patients based on acuity (critical patients seen first) and waiting time, using a triage score (from the Triage agent) to sort the queue.
- **OR Block Scheduling Optimization:** Assigns surgeries to available OR time blocks, aiming to maximize utilization while handling priority cases (e.g., emergency surgeries might bump electives).
- **Surgery Duration Prediction:** Uses an ML model (e.g., XGBoost regression) to predict surgery lengths based on historical data and procedure type, to better schedule OR blocks and reduce idle or overrun times.
- **Real-Time Reprioritization:** Integrates with the Orchestrator to reshuffle OR and ER priorities if emergencies occur or if surgeries run long/short, ensuring critical cases are always accommodated.

Future: The initial design leveraged GPU-accelerated algorithms for real-time rescheduling – combining **NVIDIA RAPIDS (cuDF/cuML)** for high-speed data processing, an XGBoost model for duration predictions, and a **reinforcement learning** agent to dynamically allocate OR resources under uncertainty. While the current implementation uses OR-Tools and a simpler ML component, a future update will reintroduce the RL-based approach for continuous, intelligent OR management (particularly valuable in large hospitals with multiple concurrent surgeries). This will enable learning from disruptions (like emergency insertions) to improve scheduling policies over time.

- **API Endpoints:**
- `POST /er/add-patient` – Add a new patient to the ER queue (with details like acuity level). Triggers an immediate re-sorting of the queue.
- `GET /er/next-patient` – Retrieve the next patient to be seen from the ER queue (according to current prioritization).
- `POST /or/schedule` – Schedule a set of surgeries/procedures into available OR slots. Input includes surgery metadata; output is a scheduled timeline or an error if constraints cannot be met.

4. Discharge Planning Agent

Port: 8004 | **Technology:** XGBoost + Rules Engine

Identifies inpatients who are ready for discharge and provides decision support for discharge planning. It uses a combination of machine learning and rule-based criteria to flag patients that could be discharged safely, helping to reduce unnecessary prolonged hospital stays.

- **Key Features:**

- **ML-Based Readiness Scoring:** An XGBoost model evaluates patient data (vitals stability, lab results, length of stay, etc.) to produce a discharge readiness score. A high score suggests the patient may be fit for discharge.
- **Clinical Rule Validation:** Built-in clinical rules (configured with medical expert input) serve as guardrails. For instance, if a patient hasn't had a normal temperature for 24 hours or if certain critical lab results are pending, they should not be discharged regardless of ML score.
- **Social/Environmental Factors:** Incorporates non-clinical factors (if available) like whether the patient has home support or needs post-discharge care, which can influence discharge decisions.
- **Estimated Discharge Date:** For patients not yet ready, the agent can estimate when they likely will be (given their trajectory), aiding in bed management and communicating with families.

Future: A planned enhancement for this agent is a **Language Model-powered assistant** to help clinicians with discharge planning. This LLM would take the ML outputs and clinical notes to generate a concise rationale for why a patient is or isn't ready, and even draft personalized discharge instructions or checklists. This will improve transparency and patient communication. As the project evolves, integration with a large language model (such as GPT-4 or fine-tuned medical LLaMA) will allow the agent to explain its recommendations in natural language and suggest next steps (e.g., follow-up appointments, home care guidelines).

- **API Endpoints:**

- `POST /analyze` – Analyze all current inpatients and return a list of those potentially ready for discharge (with scores and reasoning for each). This can be run as part of a daily routine.
- `POST /analyze-single` – Analyze a single patient (by ID) for discharge readiness, returning a detailed result for that patient.
- `GET /criteria` – Retrieve the list of rules/criteria currently used in discharge decisions (for transparency or UI display).

5. Triage & Acuity Agent

Port: 8005 | **Technology:** XGBoost + NLP

Assists with initial patient triage by analyzing symptoms and vitals to assign an acuity level (e.g., 1 = critical, 5 = non-urgent). It processes incoming patient information (symptom descriptions, vital sign readings, basic lab results) to support nurses and doctors in making quick triage decisions.

- **Key Features:**

- **NLP Symptom Extraction:** Uses a lightweight NLP component to parse chief complaints and medical history text. It can pick out key symptoms and relevant negations (e.g., “no chest pain,” “history of diabetes”) to inform triage scoring.
- **Vital Sign Analysis:** Incorporates vital readings (heart rate, blood pressure, oxygen saturation, etc.) into the acuity assessment. For example, very abnormal vitals will escalate priority.

- **Red Flag Detection:** Contains a library of “red flag” rules – critical findings that should immediately mark a case as high priority (for instance, signs of stroke, heart attack, sepsis indicators). These can override the ML model if present.
- **Rule-Based Overrides:** For safety, certain conditions always trigger an override. A nurse or physician can also manually override the AI suggestion via the Orchestrator interface, with the override logged for audit.

Future: The triage agent will be extended to incorporate **medical imaging data** for a more comprehensive emergency assessment. In particular, integration with **NVIDIA Clara** is planned, enabling analysis of imaging studies (like X-rays or CT scans) as part of triage in the ER. For example, if a portable chest X-ray shows a pneumothorax, the system would raise the acuity score. This imaging AI component, along with continuous IoT vital monitoring, will significantly enhance the triage accuracy for critical cases. The agent’s NLP capabilities may also be upgraded with advanced transformers for understanding free-text nurse notes or referral letters.

- **API Endpoints:**
 - `POST /triage` – Submit a new patient’s data (symptoms, vitals, etc.) for triage. Returns an acuity level and explanation (which factors contributed to that decision).
 - `POST /batch-triage` – Batch version of the above, allowing processing of multiple patient entries at once (useful when initializing the system with a list of patients or in mass-casualty incidents).
 - `POST /override` – Log a manual override for a patient’s triage level. This informs the system that for this patient, a human decided a different acuity, which could be used to improve the model or for auditing.

6. Supervisor/Orchestrator Agent

Port: 3000 | **Technology:** Node.js + TypeScript (Express)

The central **coordination hub** that orchestrates all other agents and manages system-wide workflows. The Supervisor (Orchestrator) doesn’t perform domain-specific predictions itself, but it ensures the right agent is invoked at the right time and aggregates results for a holistic outcome.

- **Key Features:**
 - **Workflow Orchestration:** Implements scheduled workflows (using cron-like schedules) such as a **daily hospital readiness routine** (morning forecast -> staffing update -> schedule optimization -> discharge roundup) and reactive flows (e.g., on new ER admission event, call Triage agent then possibly update ER/OR schedule).
 - **Scheduled Job Management:** Uses a scheduler (like `node-cron`) to trigger tasks at defined times. For example, run demand forecasting every midnight, staff scheduling every Sunday evening, etc. These jobs are defined in the `orchestrator/src/scheduler` module.
 - **Agent Health Monitoring:** Periodically pings each agent’s health endpoint to ensure all services are up. If an agent is unresponsive, the Orchestrator can raise an alert or attempt a restart (depending on deployment setup). A consolidated `/api/agents/health` endpoint provides a snapshot of all agent statuses.
 - **Database Persistence:** Serves as an API gateway to the PostgreSQL database for certain operations. While each agent directly writes its outcomes to the DB, the Orchestrator can also query the DB (e.g., to combine data from multiple agents or to serve the frontend) via its own endpoints.

Future: The orchestrator is planned to evolve into an **AI-driven coordinator** itself. The vision is to embed a **reasoning engine (LLM)** within the supervisor, giving it the ability to interpret complex scenarios and make high-level decisions. For instance, the LLM could analyze a summary of the hospital's state (outputs from all agents, current wait times, resource utilization) and propose strategic adjustments or explanations. This could function as an AI "chief operating officer" that not only follows predefined workflows but also provides recommendations (in natural language) to hospital administrators. Currently, the Orchestrator logic is rule-based (deterministic workflows), which ensures reliability; the LLM integration is a future enhancement to add adaptive, context-aware planning.

- **API Endpoints:** (Note: The Orchestrator uses a base path `/api/...` for its endpoints)
- `POST /api/forecast/run` – Manually trigger the Demand Forecast agent to run a new prediction cycle immediately. Useful for on-demand forecasting outside the schedule.
- `POST /api/triage` – Proxy endpoint to send data to the Triage agent and get results (the Orchestrator can combine triage with other actions, if needed).
- `POST /api/workflow/daily` – Trigger the full daily workflow (forecast -> schedule updates -> etc.) immediately. Meant for testing or one-click operation.
- `GET /api/agents/health` – Get health status of all agents. The Orchestrator returns a consolidated report (each agent's heartbeat), so an ops user or monitoring system can check everything in one call.

7. Federated Learning Services

Ports: 8086 (Triage FL Server), 8087 (Demand FL Server) | **Technology:** Flower Framework (flwr) v1.7.0

Enables **privacy-preserving collaborative machine learning** across multiple hospitals. These services coordinate federated learning rounds among distributed clients (each client representing a hospital's local dataset) to produce global models without centralizing data.

As detailed in the *Federated Learning Architecture* section above, there are two primary FL services corresponding to two domains: patient demand forecasting and triage acuity classification.

- **FL Demand Server (8087):** Coordinates training of a global forecasting model. Each client (hospital) trains a local ARIMA (or other time-series model) on its data; the server collects model parameters and uses an aggregation strategy (here, selecting the best model from clients) to form an improved model.
- **FL Triage Server (8086):** Coordinates training of a global triage classifier. Each hospital's `fl-triage-client` trains an XGBoost model on local triage data; the server merges these models by evaluating them and choosing the best (or potentially ensembling in future versions).
- **FL Clients:** Represented as separate processes/containers (labeled as Client1, Client2, Client3 in the architecture) to simulate multiple hospitals. In practice, each hospital would run its own FL client pointing to the central FL server. The clients handle local training and implement the Flower client API to send updates.

Key Features:

- **Privacy-Preserving:** No raw data leaves a hospital. Only model updates (e.g., learned coefficients or

gradients) are exchanged. This ensures patient data confidentiality across the network.

- **Collaborative Learning:** All participants benefit from collective data. For instance, if Hospital A has data on a rare condition that improves the triage model, Hospitals B, C, etc., will gain that improved model through federation.

- **Configurable Rounds:** The number of training rounds and other hyperparameters are configurable on the FL server. By default 5 rounds are run, but this can be tuned depending on model convergence needs.

- **Best Model Selection Strategy:** Unlike federated averaging, we employ a custom strategy: after each round, the server evaluates each client's model on a validation set (or a hold-out global set if available) and picks the best model to broadcast as the global model for the next round. This approach is robust when data is not IID or clients have varying quality of data.

- **Metrics & Monitoring:** During training, relevant metrics are logged – e.g., each round's RMSE for demand forecasting and accuracy for triage classification. These can be viewed in logs or could be reported back to the Orchestrator for a dashboard.

Dataset & Usage: As mentioned, sample CSVs are provided for three dummy hospitals in `datasets/`. One can easily simulate more by copying these or using real data. Running the FL system (via Docker or manually as shown above) will output logs indicating the progress of rounds and the performance of models.

Integration with Main Agents: After federated training, the global model artifacts can be saved (e.g., as a serialized model file or database entry). The main Demand Forecast agent and Triage agent can then load these improved models for inference. In this way, the federated training process can periodically update the models used by the hospital's live system, ensuring they become more accurate as more collective data is learned from – all without compromising privacy.

Alternate Execution: The federated clients can also be run from within the agent directories (using the included `fl_run_client.py` in the agents). This approach is useful in development if you want an agent to participate in FL while it's also serving requests. For example:

```
# Run a demand forecast agent's FL client from its directory (instead of via
separate container)
cd agents/demand_forecast
python fl_run_client.py --cid 1 --server-address localhost:8087

# Similarly for triage agent:
cd agents/triage_acuity
python fl_run_client.py --cid 1 --server-address localhost:8086
```

The above will register the agent itself as a federated learning client (using its local data and model), which is a design choice to keep FL logic close to the agents.

🔧 Development

Local Development Setup

While Docker is the easiest way to run the whole system, developers may want to run services individually for testing or to make changes. Below are instructions for setting up the development environment for different components.

Orchestrator (Node.js/TypeScript)

If you plan to develop the orchestrator:

```
cd orchestrator
npm install
npm run dev          # Start in development mode (nodemon + ts-node for hot reload)

# Alternatively, for a production-like run:
npm run build        # Compile TypeScript to JavaScript (dist folder)
npm start            # Start the compiled Node.js app
```

The orchestrator by default will listen on port 3000 (as specified in the env file). Adjust configuration in `orchestrator/src/config/` as needed.

ML Agents (Python)

Each agent service can be run locally (outside Docker) for development and debugging. For example, to run the Demand Forecast agent locally:

```
cd agents/demand_forecast # or any agent directory, e.g., staff_scheduling,
trriage_acuity, etc.
python -m venv venv        # Create a virtual environment (optional but
recommended)
# Activate the virtual environment:
# On Windows: venv\Scripts\activate
# On Mac/Linux: source venv/bin/activate

pip install -r requirements.txt # Install agent-specific dependencies
python api.py                 # Start the FastAPI server for this agent
```

By default, each agent's API will listen on the port specified in its config (e.g., demand_forecast on 8001). You may tweak configurations (like model parameters) in the agent's `config.py`. The agent will try to connect to the PostgreSQL DB and MLflow, so ensure those are running (you can use Docker for the DB/MLflow while running the agent code locally).

Federated Learning Development

For federated learning, you can similarly run servers/clients locally:

```
# In one terminal, start the FL servers:
cd federated_learning
python -m server.demand_server      # Starts FL Demand server on 8087
python -m server.triage_server      # Starts FL Triage server on 8086

# In other terminals, start a few FL clients:
python -m client.demand_client --cid 1 --server-address localhost:8087
python -m client.demand_client --cid 2 --server-address localhost:8087
python -m client.triage_client --cid 1 --server-address localhost:8086
...
```

The `--cid` parameter is a client ID (just for logging differentiation) and `--server-address` tells the client where the server is. Ensure you have appropriate data CSVs present for each client instance you run. You can observe the servers' output to verify training rounds.

Running Tests

Automated tests (unit or integration tests) can be run for each component:

```
# Orchestrator tests (Jest is set up, if applicable)
cd orchestrator
npm test

# Python agent tests (if any are provided or if you add your own)
cd agents/demand_forecast
pytest

# Similarly for other agents...
```

Each agent folder may include sample tests or you can add tests following a similar structure.

Building for Production

If you have modified the code or want to build fresh images:

```
# Build all Docker images (rebuild the entire stack)
docker-compose build

# Or build a specific service image by name:
docker-compose build orchestrator
```

```

docker-compose build demand-forecast
docker-compose build triage-acuity
... etc.

# Build without cache (for a clean rebuild of all images)
docker-compose build --no-cache

# You can also build an agent image manually outside compose if needed:
cd agents/demand_forecast
docker build -t swasthya-demand-forecast:latest .

```

The Docker images are defined to use multi-stage builds where appropriate and pin versions to ensure consistency.

Database Schema

Swasthya uses **PostgreSQL** as the primary relational database for operational data (aside from the blockchain component, which is separate). The schema includes tables corresponding to various aspects of hospital operations:

- `forecasts` – Stores patient volume forecasts (output of Demand Forecast agent) with timestamps and prediction horizons.
- `staff` – Stores staff information (doctors, nurses, etc. with their roles, shifts, etc.).
- `staff_schedules` – Stores generated staff schedules (which staff is assigned to which shift/role on which date).
- `triage_decisions` – Logs of triage outcomes (acuity level assigned, factors, timestamp, etc.).
- `er_queue` – Represents the current state of the ER queue (could be implemented as a view or table that the ER/OR agent updates).
- `or_schedules` – Stores the planned OR surgery schedule (which patient/surgery is assigned to what OR slot).
- `inpatients` – Active inpatient roster with key details (admit date, current ward, attending doctor, etc.).
- `discharge_recommendations` – Outputs from the Discharge Planning agent (patient X is recommended for discharge on date Y along with a confidence score or notes).

Additionally, some **views** may collate data for convenience, e.g., joining patient info with their latest triage score, etc.

Database initialization files:

- `data/01_create_databases.sql` – Creates the separate database for MLflow (if using a separate DB) and any required roles. The MLflow tracking server by default uses its own SQLite or can use Postgres; this script can be configured accordingly.
- `data/init_database.sql` – Contains the DDL statements to create the tables listed above (and others) and possibly insert some sample seed data (for development/demo). It's executed on container startup to set up the schema.

(Maintaining an updated schema diagram or description is important as the project evolves. The new README includes this section to make it clear how data is structured, which the original documentation lacked.)

Security

Security is paramount given the sensitive nature of health data. The project includes guidelines and configurations to ensure a secure deployment:

Production Deployment Checklist

Before deploying Swasthya in a real-world (especially production) environment, the following steps are recommended:

- [] **Credentials:** Change all default credentials (e.g., PostgreSQL `postgres` user password, any default API keys). Never use the example `.env` secrets in production.
- [] **API Keys & Auth:** If not already, configure an API authentication mechanism. At minimum, update the `API_KEY` (used for simple auth in dev/testing) in the `.env` and enforce its use in requests. For a more robust setup, consider OAuth2 or JWT auth for all API endpoints.
- [] **Enable HTTPS:** Terminate API endpoints behind HTTPS. Use TLS certificates (e.g., via NGINX reverse proxy or cloud load balancer) so that all traffic, especially any containing PHI, is encrypted in transit.
- [] **Firewall & Network:** Restrict network access to the servers. Only expose necessary ports (e.g., orchestrator, agent APIs) to trusted networks or VPN. Close or firewall off internal component ports (databases, etc.) from the public internet.
- [] **MLflow Security:** If MLflow UI is deployed, protect it (since it may contain model info derived from sensitive data). Options: enable authentication for the MLflow server or keep it accessible only within an internal network.
- [] **Audit Logging:** Enable audit logs for data access. E.g., log every time patient data is accessed or a model makes a critical recommendation, including who/what accessed it.
- [] **Regular Updates:** Keep all dependencies updated (security patches for Node, Python packages, OS packages in containers). Rebuild images or apply patches regularly to address new vulnerabilities.
- [] **Data Encryption at Rest:** Use disk encryption for databases or at least ensure backups and data dumps are encrypted. If using cloud services, enable their encryption features.
- [] **Inter-Service Security:** If deploying on cloud or across networks, consider securing inter-service communication (e.g., mutual TLS between agents and orchestrator, or placing them on a private network). Also, sanitize and validate all inter-service data payloads to avoid injection attacks even from internal sources.

By following the above, one can significantly harden the deployment beyond the out-of-the-box development setup.

HIPAA Compliance Considerations

Given that Swasthya deals with Protected Health Information (PHI), anyone deploying or extending it should consider regulatory compliance (like **HIPAA** in the US, or India's forthcoming health data protection regulations):

- **On-Premises PHI Storage:** All PHI (personally identifiable patient data) should remain on secure hospital servers. If using cloud for some components, ensure only de-identified or aggregate data goes to cloud.
- **De-Identification:** When using federated learning or any data sharing, ensure patient data is de-identified (remove or encrypt personal identifiers) unless absolutely necessary.
- **Access Controls:** Implement role-based access control in the application – e.g., only authorized users can view certain information or trigger certain actions. Maintain an audit trail of user access.
- **Encryption:** Always encrypt PHI, both at rest (database, backups) and in transit (use HTTPS, VPNs).
- **Security Audits:** Periodically conduct security risk assessments and penetration testing on the system to identify and fix vulnerabilities.
- **Documentation & Training:** Maintain documentation on how the system uses patient data, and ensure users/administrators of the system are trained in security best practices (e.g., not exposing API keys, using strong passwords, etc.).

By design, features like federated learning and blockchain support are intended to enhance compliance (by minimizing data sharing and providing immutable audit logs respectively). Implementers should leverage these features fully to meet legal requirements.

Federated Learning Privacy Benefits

It's worth highlighting how the federated learning approach in Swasthya inherently supports privacy and compliance:

- **No Data Centralization:** Hospitals keep their datasets locally; the FL server never sees raw data, only model updates.
- **Local Training Only:** All training happens behind each hospital's firewall. Only the training results (gradients or model weights) are sent out.
- **Model-Only Sharing:** The information shared is abstract and often encrypted. Techniques like Secure Aggregation can be added to ensure the FL server cannot inspect individual updates.
- **Differential Privacy (Future):** The system can be extended with differential privacy measures (adding noise to updates) if an extra layer of privacy is required.
- **Compliance-Friendly:** Since no PHI leaves the premises, federated learning can often bypass the need for complex data-sharing agreements, making multi-center collaborations easier under regulations.
- **Distributed Ownership:** The model is a collective asset rather than any one hospital's data asset, promoting a collaborative environment aligned with patient benefit.

In summary, security and privacy are first-class considerations in Swasthya's design, and the documentation explicitly includes this guidance (where the original documentation did not), reflecting a more mature readiness for real-world use.

Monitoring & Logging

Effective monitoring and logging are crucial for operating a complex multi-agent system like Swasthya. The README now includes guidance for these (which the original lacked):

Logs

Each service produces logs that can be used for debugging and performance monitoring:

- **Orchestrator Logs:** By default, the Orchestrator (Node.js) logs to console; in Docker, these logs can be viewed with `docker-compose logs orchestrator`. In a production setup, consider configuring Winston (which is included in the stack) to output to rotating log files under `orchestrator/logs/`, with different levels (info, error) separated.
- **Agent Logs:** Each Python agent logs to both console and a file (if configured). In Docker, use `docker-compose logs <service-name>` (e.g., `swasthya-demand-forecast`) to tail logs. For local dev, logs might be written to `agents/<agent_name>/logs/` (check each agent's config for logging settings).
- **Federated Learning Logs:** The FL servers and clients log their round progress and any errors. Use `docker-compose logs fl-demand-server` (and similarly for clients) to monitor training. They output metrics per round which are important to track model performance.

Setting up a centralized logging solution (like ELK stack or a cloud log aggregator) is recommended for production to collect logs from all 16 containers in one place.

MLflow Tracking

The **MLflow UI** accessible at <http://localhost:5000> is a powerful tool for monitoring the machine learning aspects:

- **View Experiments:** As agents train models (either initial training or retraining), they log experiment runs to MLflow. You can see all runs, compare metrics (e.g., forecast error rates, XGBoost model AUCs, etc.), and visualize training curves.
- **Compare Model Performance:** MLflow allows comparing multiple runs of a model. For example, if you try a Prophet vs ARIMA vs TFT (in the future) for forecasting, their metrics can be compared side by side.
- **Model Registry:** The system uses MLflow's model registry to manage champion vs challenger models. The UI will show which model version is staged as "Production" vs "Staging" for each agent's models, if that workflow is adopted.
- **Artifact Store:** Any artifacts (like trained model binaries, plots) saved by agents can be accessed via MLflow UI. For instance, the Demand agent might save a plot of forecast vs actuals for validation—this image would be an artifact you can inspect.

To use MLflow, ensure the MLflow container is running and accessible. The README includes it in Docker Compose for convenience. In a production scenario, you might secure this UI or integrate it with your CI/CD for model management.

Health Checks

For basic service monitoring, each agent (and the orchestrator) has a health check endpoint:

```
# Check orchestrator and all agents health via orchestrator's aggregate
endpoint:
curl http://localhost:3000/api/agents/health

# Sample response (JSON):
# {
#   "demand_forecast": "healthy",
#   "staff_scheduling": "healthy",
#   "error_scheduling": "healthy",
#   ...
# }

# Or check individual agent health endpoints directly:
curl http://localhost:8001/health # Demand Forecast agent health
curl http://localhost:8005/health # Triage & Acuity agent health
```

These endpoints simply return a status (and possibly version info) indicating the service is up and responding. They are used by Orchestrator's monitoring and can be integrated into external uptime monitors or load balancers (for example, Kubernetes liveness probes could hit these).

Additionally, the federated servers currently don't have an HTTP health endpoint (they operate via Flower gRPC protocol), so health is inferred from their logging activity or by checking the process. If needed, a lightweight sidecar or a simple HTTP wrapper could be added to them for health checks.

Overall, the new monitoring/logging guidance ensures that those deploying Swasthya have clear instructions on how to observe the system's behavior and detect issues, something that was missing before.

Deployment

Swasthya is designed with containerization in mind to simplify deployment. Two main deployment methods are supported out-of-the-box: **Docker Compose** for multi-container orchestration on a single host, and **Kubernetes** (via Kompose conversion or custom manifests). Additionally, a hybrid cloud/on-premise model is outlined for splitting training vs inference.

Docker Compose (Recommended for Demo/Dev and Small-Scale Deployments)

The repository provides a `docker-compose.yml` that orchestrates all services (database, orchestrator, agents, etc.). Using Docker Compose, you can easily run the entire platform on a single machine or VM.

The system includes **16 containerized services** orchestrated by Docker Compose (the original documentation mentioned 18, but in the current configuration there are 16 active containers, as broken down below):

- **Infrastructure (2 containers):**

- PostgreSQL database (data persistence)
- MLflow tracking server (experiment tracking UI)

- **Core Service (1 container):**

- Orchestrator (Node.js/TypeScript API gateway and scheduler)

- **AI Agent Services (5 containers):**

- Demand Forecast Agent (Python FastAPI)
- Staff Scheduling Agent (Python FastAPI)
- ER/OR Scheduling Agent (Python FastAPI)
- Discharge Planning Agent (Python FastAPI)
- Triage & Acuity Agent (Python FastAPI)

- **Federated Learning (8 containers):**

- FL Demand Server + 3 Demand Clients
- FL Triage Server + 3 Triage Clients

This totals 16 containers. (If needed, you can scale the number of FL clients or add future services, adjusting the count accordingly.)

Common Docker Compose commands (assuming you have Docker/Compose installed and are in the project directory):

```
# Start all services in the background (as daemons)
docker-compose up -d

# Check status of all containers
docker-compose ps

# Follow logs of a specific service (e.g., orchestrator)
docker-compose logs -f orchestrator

# Follow logs for multiple/all services
docker-compose logs -f          # logs for all, or specify multiple like '...
orchestrator demand-forecast'

# Stop all services
```

```
docker-compose down

# Stop and also remove volumes (erasing the database, etc.)
docker-compose down -v

# Restart a specific service (e.g., triage agent after code update)
docker-compose restart triage-acuity

# Scale a service (run multiple instances of an agent behind a load balancer, if
that was configured)
docker-compose up -d --scale demand-forecast=3
# Example: 3 replicas of demand forecast agent
```

Docker Compose makes it easy to run the system for demonstration or development purposes on a single host. For a production environment, consider using Kubernetes or other orchestration for better scalability and resilience.

Kubernetes (Advanced)

For production-grade deployments, Kubernetes can be used to manage the containers across a cluster of machines. While custom K8s manifests are not (yet) provided, you can generate a starting point using Kompose (a tool to convert docker-compose definitions to Kubernetes):

```
# Install kompose (if not already installed)
kompose convert -f docker-compose.yml
# This will generate YAML files for deployments, services, etc.

# Apply the generated manifests to your cluster
kubectl apply -f .
```

Be aware that the Kompose conversion may require editing. For example, setting proper PersistentVolume for PostgreSQL, configuring Services as NodePort/LoadBalancer for external access, etc. Also, secrets (like environment variables for passwords) should be managed via Kubernetes Secrets.

We plan to provide a tailored `k8s/` manifest directory in the future with optimized resource requests, autoscaling rules, and network policies.

Hybrid Deployment

The architecture supports a **hybrid deployment** model, splitting components between cloud and on-premise environments:

- **Cloud (Training & Analytics):**
 - MLflow tracking server (could be hosted in cloud for central access by multiple hospitals)
 - Intensive model training jobs (e.g., re-training large models, federated learning coordination) can be done on cloud VMs with GPUs.

- Data aggregation or analytics dashboards that benefit from cloud scalability.

- **On-Premises at Hospital (Inference & Data Collection):**

- Orchestrator and agent services run on a local server within the hospital to ensure low latency and data never leaving local storage for real-time operations.
- The PostgreSQL database with patient records stays on-prem for data ownership and compliance.
- IoT devices connect to the on-prem orchestrator/agents for real-time monitoring.

For example, a hospital could run the Orchestrator and all agent containers on an on-site machine. Periodically, the federated learning clients will connect to a cloud FL server to update models. The MLflow server might also be cloud-based, so that multiple sites log their metrics to one place for aggregate analysis. Meanwhile, all inference (live predictions, scheduling decisions) happen locally at the hospital site.

This hybrid approach leverages the best of both: cloud for heavy lifting and collaboration; edge/on-prem for data-sensitive, low-latency tasks. The README highlights this possibility to guide large deployments.

(The original documentation did not explicitly cover deployment scenarios; the updated version ensures clarity on how to actually deploy and scale the system.)

Contributing

We welcome contributions to improve Swasthya and adapt it to more use cases! If you'd like to contribute:

1. **Fork the repository** (on GitHub, click "Fork").
2. **Create a feature branch** for your changes:

```
git checkout -b feature/AmazingFeature
```

3. **Commit your changes** with clear messages:

```
git commit -m "Add AmazingFeature: description of changes"
```

4. **Push to your branch** on your fork:

```
git push origin feature/AmazingFeature
```

5. **Open a Pull Request** on the main repo: describe your changes and link any relevant issues.

Please ensure that your code follows the established structure and includes appropriate documentation/comments. If adding a new agent or major component, update the README (especially the architecture diagram and project structure) accordingly.

For major changes, it's best to open an issue first to discuss the idea with maintainers. This project is in active development, and we strive to keep documentation and implementation in sync.

License

This project is licensed under the **MIT License**. See the `LICENSE` file for details. This means you are free to use, modify, and distribute this software as long as you include the license and attribution. We believe in open innovation, especially for something as impactful as healthcare – contributions and usage are encouraged!

Technology Stack

The Swasthya project is built with a modern stack of technologies across web, AI, and DevOps:

Orchestration & Backend

- **Node.js 18** – JavaScript runtime for the Orchestrator service.
- **TypeScript 5.3** – Strongly-typed superset of JS used for the orchestrator codebase.
- **Express 4.18** – Web framework for building the Orchestrator's REST API.
- **PostgreSQL 15** – Relational database for system data (patients, schedules, etc.).
- **TypeORM** (or similar ORM) – For database migrations and object-relational mapping (if used in orchestrator).
- **Winston** – Logging library for structured logs in Node.js.
- **node-cron** – Scheduler for running cron-like jobs in Node.
- **Axios** – HTTP client for Orchestrator to call agent APIs.

Machine Learning & Optimization (Python)

- **Python 3.10** – Language for all agent services and FL services.
- **FastAPI 0.95** – High-performance Python web framework (based on Starlette) to expose agent models via REST.
- **scikit-learn 1.2** – General ML utilities (could be used for preprocessing, baseline models).
- **XGBoost 1.7 / 2.0** – Gradient boosting library used in Triage & Discharge agents for classification/regression.
- **Prophet 1.1** – Facebook Prophet for time-series forecasting in Demand Forecast agent.
- **statsmodels 0.14** – Used for ARIMA time-series models and statistical tests.
- **pmdarima** – For easier ARIMA model training (auto-ARIMA functionality).
- **Google OR-Tools 9.6** – Operations research library (CP-SAT solver) for Staff and OR scheduling optimizations.
- **PyTorch / PyTorch Forecasting** – *Planned*: Will be used for advanced forecasting (Temporal Fusion Transformer) in the future.
- **Hugging Face Transformers** – *Planned*: For NLP enhancements, possibly to use pretrained language models in triage or discharge planning.

Federated Learning

- **Flower (flwr) 1.7.0** – Federated Learning framework enabling server-client architecture for training (it abstracts communication and orchestration of rounds).
- **grpcio** – Flower uses gRPC under the hood for efficient network communication between FL servers and clients.

- **NumPy/Pandas** – Used in clients for data handling in training.
- **Custom Aggregation Strategy** – The `BestModelStrategy` implemented for selecting models (using Python logic within the Flower server callbacks).

MLOps & Monitoring

- **MLflow 2.9** – Machine Learning Lifecycle management:
 - *Tracking*: Logging parameters, metrics, and artifacts of model runs.
 - *Model Registry*: Storing and versioning trained models (with stages like Staging/Production).
 - *UI*: Web interface to view experiments.
- **Prometheus/Grafana** – *Planned*: For metrics scraping and dashboarding (not yet included, but the system can expose metrics endpoints if needed).
- **ELK Stack (Elasticsearch/Logstash/Kibana)** – *Planned*: For centralized logging and log analytics (considering the volume of logs from 16 services).

DevOps & Deployment

- **Docker** – Containerization of all components for consistent environment across development, testing, and production.
- **Docker Compose** – Orchestration of multi-container setup (as a single-node orchestrator, great for local dev or single VM deploys).
- **Kubernetes** – Support for deploying on K8s clusters (tested via Kompose conversion; proper manifests to be added in future).
- **GitHub Actions** – *Planned*: CI/CD pipeline for automated testing, building, and deploying containers.
- **Kompose** – Used to bootstrap Kubernetes resource definitions from Compose (for those moving to K8s).

Development Tools

- **Git** – Version control for the code.
- **pytest** – Python testing framework (for agent logic tests).
- **Jest** – JavaScript testing framework (to test Orchestrator logic; *planned*: adding more orchestrator tests).
- **Prettier/ESLint** – To enforce code style and quality in the TypeScript code.
- **Black/Flake8** – To enforce style in Python code (ensuring readability and consistency).
- **Visual Studio Code** – Recommended IDE with good support for both TypeScript and Python (with devcontainer settings for Docker, potentially).

*Planned future integrations include: **NVIDIA Clara** (for imaging AI acceleration), **Meta LLaMA/Code Llama** (LLMs for clinical text reasoning), and **Ethereum/Solidity** (for the blockchain health wallet). These are not yet in the core stack but are part of the project's forward-looking technology roadmap.*

Acknowledgments

- **Healthcare Professionals**: We thank the doctors, nurses, and hospital administrators who provided domain expertise and helped identify real-world requirements (their feedback shaped the Problem Statement and agent designs).

- **Open Source Community:** This project builds on many open-source libraries – thanks to contributors of Prophet, XGBoost, OR-Tools, Flower, FastAPI, and others for making such powerful tools freely available.
- **NVIDIA Clara Team:** For their research and toolkits on medical imaging AI, which inspire our planned triage enhancements.
- **Meta AI Research:** For open-sourcing advanced models like LLaMA and Kats, which inform our advanced AI strategies.
- **MumbaiHacks Hackathon:** (If applicable) Acknowledge the hackathon or event that sparked the project's inception, if this was initially developed in such a context.
- **Community & Testers:** Everyone who has tested the system and provided feedback, helping us improve both the code and documentation.

Swasthya's development is a collective effort – we appreciate all those who have indirectly contributed by creating the ecosystem that made it possible.

Support

Need help or have questions? We're here to support developers and users of Swasthya.

Getting Help

Quick Help Resources:

- **API Documentation:** Each running agent provides interactive API docs at `http://<host>:<port>/docs`. Use these to understand input/output formats and to manually try out endpoints in a web browser.
- **Configuration:** Refer to the `env.example` file for all environment variables that can be set. This project is highly configurable via environment settings (ports, debug modes, database URLs, etc.).
- **Project Structure Guide:** If you're trying to find where a certain functionality is implemented, check the *Project Structure* section above. It will guide you to the correct folder or file for various concerns (e.g., "Where is the triage model code? -> `agents/triage_acuity/model.py`").

For Developers:

- The code is documented with inline comments and type hints (in the TS code) to clarify complex sections. Don't hesitate to read through the agent or orchestrator code – we've tried to make it readable.
- The MLflow UI (`http://localhost:5000` when running) can also serve as a debugging tool – you can inspect if models are training correctly and compare runs if you adjust parameters.
- The database schema in `data/init_database.sql` can be consulted to understand what data each agent expects to read or write. For example, knowing there's an `or_schedules` table suggests how the OR scheduling agent outputs its results.

Contact & Issues

If you encounter a bug, or the system doesn't behave as described:

- **GitHub Issues:** Please open an issue on the repository's issue tracker. Include as much detail as possible: steps to reproduce, logs, and screenshots if applicable. We track issues closely and will respond.
- **Email:** You can reach out to the maintainers at `support@swasthya.example.com` for specific inquiries or if you want to collaborate.
- **Feature Requests:** For suggesting new features or enhancements (e.g., integration with another system, new agent idea, etc.), opening a discussion or issue on GitHub is the best way to start the conversation.

We aim to foster an open community around Swasthya. Whether it's troubleshooting deployment or brainstorming the next big feature to help hospitals, we're excited to hear from you!

Roadmap

Swasthya is under active development. Below is the roadmap with completed features and upcoming planned enhancements. The roadmap ensures transparency about which ambitious features from the project vision are implemented now and which are slated for future releases.

Completed Features

- **Multi-agent AI system** – 6 specialized agents + 1 orchestrator working in concert (demand forecasting, staff scheduling, ER/OR scheduling, discharge planning, triage, supervisor).
- **MLflow integration** – Full experiment tracking and model registry for all learning components.
- **Federated Learning** – Working federated training for demand forecasting and triage models, enabling multi-hospital collaboration without data sharing.
- **Containerized deployment** – Docker Compose setup with all 16 services for easy one-command deployment.
- **Comprehensive documentation** – (This README and associated docs) Covering architecture, setup, usage, and development, making it easier to onboard new contributors.
- **PostgreSQL database with schema** – A robust data layer with all necessary tables, supporting transactional consistency and complex queries.
- **RESTful APIs with OpenAPI** – Each service exposes clear REST endpoints with auto-generated docs, simplifying integration with frontends or other systems.
- **Hybrid cloud support** – Configuration allowing separation of training vs inference workloads (verified with cloud MLflow and local agents scenario).
- **Basic security and compliance guidelines** – Added to documentation (checklist and best practices) to guide production deployments.

Planned Enhancements

- [] **Advanced Reinforcement Learning for dynamic resource allocation** – Integrate RL algorithms for Staff Scheduling and OR Scheduling agents to adapt to changing conditions in real-time,

improving over static optimization. (This was part of the original vision and is now being actively researched.)

- [] **LLM-driven Decision Support** – Incorporate Large Language Models (e.g., Meta’s LLaMA or GPT-4) into the Supervisor and/or agents for reasoning and explanations. For example, an LLM in the Orchestrator could summarize system status and suggest improvements, or explain to clinicians why the AI made a particular recommendation.
- [] **Blockchain-based Health Wallet integration** – Implement the blockchain layer for patient records with a working Ethereum smart contract and integrate it with the Orchestrator/agents. Patients will have a decentralized health identity (via Aadhaar linking) and providers can read/write to it through Swasthya, bringing the vision of interoperable health records to life.
- [] **IoT Vitals Monitoring** – Develop an IoT integration service or extend an agent to continuously ingest vital sign data (from wearables, ICU machines). This includes real-time processing of data streams and alerting mechanisms in the Orchestrator (e.g., if a patient’s blood pressure spikes, auto-notify a doctor and re-run triage).
- [] **Integration with EHR systems (HL7/FHIR)** – Enable Swasthya to pull/push data from existing hospital Electronic Health Record systems using standards like HL7 v2 or FHIR. This will allow easy adoption in hospitals by interfacing with their current patient databases and ADT systems.
- [] **Mobile App for Clinicians/Staff** – A companion mobile application or responsive web app that staff can use to view schedules, get notifications (e.g., “ER patient acuity high, please attend”), and interact with the system on the go.
- [] **Real-time Predictive Analytics Dashboard** – Develop a front-end dashboard showing key metrics (current ER wait time, predicted admissions, bed occupancy forecasts, etc.) in real time, for hospital administrators. This likely involves a Web UI that consumes the orchestrator’s APIs.
- [] **Kubernetes Deployment Manifests** – Provide official K8s YAML manifests and Helm charts for deploying Swasthya on a cluster, including production-ready configurations (autoscaling rules, persistent volumes, secrets management, etc.).
- [] **Advanced NLP for Clinical Notes** – Extend the Triage or a new agent to analyze unstructured clinical notes or referral letters using advanced NLP (Transformer models). This can extract additional features for triage or help in discharge planning by summarizing a patient’s hospital course.
- [] **Automated Model Retraining Pipelines** – Implement a pipeline (possibly using CI/CD or Apache Airflow) to periodically retrain models as new data comes in, and detect model performance drift. This ensures the AI stays up-to-date with changing hospital patterns (e.g., seasonal changes in patient volumes).
- [] **Multi-Language Support for Triage** – Enable the triage NLP to understand multiple languages (important in India’s multilingual context). This might involve training models for Hindi, Marathi, etc., or using translation pipelines so that patients can be triaged in their native language.
- [] **Advanced Visualization for Forecasting & Scheduling** – Provide richer visual outputs like interactive charts of forecast vs actual, Gantt charts for OR schedules, etc., possibly integrated into the dashboard or via the MLflow artifact UI.
- [] **Integration with Ambulance Routing Systems** – Extend the network to pre-hospital care by connecting with ambulance dispatch systems. For instance, use predictive analytics to route ambulances to hospitals with available capacity, and feed en route patient vitals to the hospital’s triage agent ahead of arrival.

(And more – this list is not exhaustive and will evolve. The key is that originally advertised features like blockchain and IoT, which are not yet implemented, are transparently listed here so stakeholders know they are planned. Community feedback may reprioritize some items.)

Built with ♥ for better healthcare delivery. Swasthya aims to bring cutting-edge technology to the frontlines of medicine, and we're excited to continue this journey in open collaboration. Here's to healthier hospitals and happier patients!
