🧠 1. Doctor Search Module

🔍 Problem:

Users enter vague queries like “heart pain in Kurnool” — keyword search fails.

✅ AI Solution:

* **Semantic Search** using embeddings (e.g. sentence-transformers)
* **NER + Intent Detection** to extract location, specialization, symptoms

⚙️ Tools:

* spaCy or transformers for NER
* FAISS for fast vector search

🚀 Performance:

* Precompute doctor profile embeddings
* Use FAISS with HNSW indexing for sub-second retrieval

💰 Cost Reduction:

* Host locally with FastAPI + Uvicorn
* Avoid cloud NLP APIs (e.g. OpenAI) unless caching

🧠 2. Doctor Recommendation Module

🔍 Problem:

Users don’t know which doctor to choose — static filters aren’t enough.

✅ AI Solution:

* **Collaborative Filtering** (based on user-doctor interactions)
* **Content-Based Filtering** (based on doctor attributes)

⚙️ Tools:

* LightFM, Surprise, or custom cosine similarity

🚀 Performance:

* Cache top-N recommendations per user
* Use batch inference for dashboard rendering

💰 Cost Reduction:

* Train offline, serve via lightweight API
* Use SQLite/PostgreSQL for interaction logs

🧠 3. Review Sentiment Analysis

🔍 Problem:

Reviews are unstructured — hard to quantify doctor quality.

✅ AI Solution:

* **Sentiment Classification** (positive/negative/mixed)
* **Aspect Extraction** (e.g. “wait time”, “communication”)

⚙️ Tools:

* VADER, TextBlob, or fine-tuned BERT

🚀 Performance:

* Use ONNX-optimized models for fast inference
* Run sentiment scoring asynchronously (Celery)

💰 Cost Reduction:

* Use rule-based fallback for short reviews
* Store sentiment scores in DB for reuse

🧠 4. Appointment Optimization

🔍 Problem:

Manual scheduling leads to overlaps and no-shows.

✅ AI Solution:

* **Predictive Modeling** to forecast no-shows
* **Smart Slot Allocation** based on doctor load

⚙️ Tools:

* scikit-learn, XGBoost, or CatBoost

🚀 Performance:

* Train weekly, serve predictions via API
* Use Redis to cache slot availability

💰 Cost Reduction:

* Avoid real-time training — use batch updates
* Use lightweight models (e.g. logistic regression)

🧠 5. Emergency Services Routing

🔍 Problem:

Users need fast access to nearby emergency help.

✅ AI Solution:

* **Geospatial Matching** using coordinates
* **Availability Prediction** for ambulance/hospital load

⚙️ Tools:

* GeoPandas, OSRM, Google Maps API

🚀 Performance:

* Pre-index emergency locations
* Use bounding box filtering for fast lookup

💰 Cost Reduction:

* Use open-source routing (OSRM) over paid APIs
* Cache frequent routes and hospital lookups

🧠 6. Profile & Dashboard Analytics

🔍 Problem:

Users and doctors need insights — not just raw data.

✅ AI Solution:

* **Behavioral Analytics** (consultation patterns, review trends)
* **Engagement Scoring** (active vs passive users)

⚙️ Tools:

* Pandas, NumPy, Plotly for visual dashboards

🚀 Performance:

* Precompute metrics nightly
* Use AJAX to load dashboard widgets asynchronously

💰 Cost Reduction:

* Avoid BI tools — build lightweight dashboards in Flask
* Use static charts for low-traffic users

🔗 Integration Strategy

🔧 Backend

* Use **FastAPI** for AI endpoints
* Modularize AI logic in /ai\_modules/
* Use **Celery** for background tasks (e.g. review scoring)

💻 Frontend

* Use **AJAX** to call AI endpoints
* Dynamically update doctor cards, dashboards, reviews

🐳 Deployment

* Dockerize each AI module
* Use **Gunicorn + Uvicorn** for concurrency
* Host on **AWS EC2 or GCP Cloud Run** with autoscaling

🧱 Folder Structure

/ai\_modules/ ├── search\_engine.py ├── recommender.py ├── sentiment\_analysis.py ├── appointment\_predictor.py ├── emergency\_router.py /routes/ ├── doctor\_routes.py ├── user\_routes.py ├── service\_routes.py /templates/ ├── dashboard.html ├── doctor\_profile.html ├── emergency\_services.html

🧠 Final Tips for Cost + Performance

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |