Intelligent Patient Triage & Care Navigation Assistant

Major Project Report Submitted in Partial Fulfilment of the

Requirements for the Degree Of

MASTER OF BUSINESS ADMINISTRATION

(EXECUTIVE)

IN

DATA SCIENCE AND ANALYTICS

BY

SIDDHARTH BHARGAVA

(Roll No.: **24/UEMBA/03**)

Under the Supervision Of

Dr. Anurag Chaturvedi



UNIVERSITY SCHOOL OF MANAGEMENT &

ENTREPRENEURSHIP

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

East Delhi Campus, Vivek Vihar, Phase – 2, Delhi – 110 095, India

August 2026

TABLE OF CONTENTS

Contents

ACKNOWLEDGEMENT …… 1

DECLARATION ……. 2

CERTIFICATE BY THE SUPERVISOR …….. 3

ABSTRACT …… 4

CHAPTER 1 - USE CASE / PROJECT OVERVIEW …….. 11

CHAPTER 2 - PROBLEM STATEMENT …… 12

CHAPTER 3 - PROPOSED SOLUTION …….. 13

CHAPTER 4 - AGENT ARCHITECTURE …… 14

CHAPTER 5 - DATA AND INTEGRATIONS …….. 15

CHAPTER 6 - INTELLIGENCE & DECISIONING …… 16

CHAPTER 7 - EXPECTED OUTCOMES ……. 17

CHAPTER 8 - INDUSTRY RELEVANCE …….. 18

CHAPTER 9 - PROJECT STRUCTURE …….. 19

CHAPTER 10 - CODE IMPLEMENTATION …… 22

CHAPTER 11 - SYSTEM ARCHITECTURE …….. 42

CHAPTER 12 - SETUP AND USAGE INSTRUCTIONS ……. 44

CHAPTER 13 - TECHNICAL IMPLEMENTATION DETAILS ……. 47

CHAPTER 14 - FUTURE INTEGRAATION ROADMAP …… 49

FIGURES …… 51

REFERENCES ……. 53

PLAGIARISM REPORT ……. 56

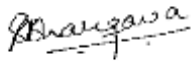
ACKNOWLEDGEMENT

It is our pleasure to be indebted to various people, who directly or indirectly contributed in the development of this work and who influenced our thinking, behaviour, and acts during the course of study.

I express my sincere gratitude to the authorities who gave me an opportunity to take this project.

I am highly indebted and extremely thankful to Dr. Anurag Chaturvedi for his support, cooperation, guidance and suggestions that helped me in completing this project. His dedication and support throughout the project have been instrumental in my learning journey.

I am also deeply appreciative of the faculty members at Delhi Technological University for their unwavering support and assistance. Your collective efforts have greatly enriched my experience, and I am truly grateful.

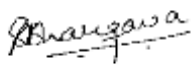


SIDDHARTH BHARGAVA

Roll No.: 24/UEMBA/03

DECLARATION

I hereby declare that the project work entitled – "Intelligent Patient Triage & Care Navigation Assistant", submitted to the USME, DTU is a record of an original work done by me under the guidance of Dr. Anurag Chaturvedi, DTU and this project work is submitted in the partial fulfilment of the requirements for the award of the degree of Master of Business Administration. The result embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.



SIDDHARTH BHARGAVA

Roll No.: 24/UEMBA/03

CERTIFICATE BY THE SUPERVISOR

Certified that Siddharth Bhargava (Roll No. 24/UEMBA/03) has carried out their search work presented in this project report titled – "Intelligent Patient Triage & Care Navigation Assistant" for the award of Master of Business Administration (Executive) Data Science and Analytics from University School of Management and Entrepreneurship, Delhi Technological University, under my supervision. The thesis embodies results of original work, and studies are carried out by the student himself, and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University / Institution.



Dr. Anurag Chaturvedi

University School Of Management & Entrepreneurship,

Delhi Technological University

ABSTRACT

Project Overview

This project presents an **Intelligent Patient Triage & Care Navigation Assistant**, a sophisticated multi-agent healthcare system designed to automate patient symptom assessment, urgency classification, and healthcare provider recommendation. Developed as part of an EMBA Seventh Semester Project, the system leverages modern artificial intelligence techniques and workflow orchestration to bridge the gap between patient symptom presentation and appropriate medical care access.

Technical Architecture

The system implements a **LangGraph-style multi-agent workflow** comprising five specialized autonomous agents orchestrated through a directed graph with conditional branching logic:

1. ****Symptom Intake Agent****: Employs natural language processing to extract structured symptom profiles from free-text patient input, including severity assessment (1-10 scale), duration parsing, and red flag detection for emergency symptoms.

2. ****Triage & Reasoning Agent****: Utilizes Retrieval-Augmented Generation (RAG) over medical guideline documents to perform evidence-based urgency classification (Emergency/High/Medium/Low) and specialty mapping, generating comprehensive reasoning traces for transparency.

3. ****Doctor Match Agent****: Implements a weighted scoring algorithm (specialty match 30%, availability 30%, patient rating 25%, proximity 15%) to rank and recommend the top three healthcare providers from a mock directory of 15 physicians.

4. ****Scheduling & Summary Agent****: Interfaces with a mock REST API to facilitate appointment booking, generates patient-friendly visit summaries, and provides specialty-specific pre-visit instructions.

5. **Guardrail Agent**: Enforces critical safety validations including diagnostic claim prevention, mandatory medical disclaimer verification, emergency escalation validation, and prohibited medical language detection.

Key Technical Features

- **Conditional Workflow Branching**: Emergency cases bypass standard doctor matching and route directly to emergency room guidance, ensuring critical cases receive immediate escalation recommendations.

- **RAG Implementation**: Medical guideline retrieval system enables evidence-based triage decisions grounded in standardized emergency criteria and symptom-specialty mappings.

- **Dual Interface Design**:

  - **Streamlit Web UI**: Interactive web application featuring real-time symptom analysis, color-coded urgency indicators, doctor recommendation cards, session history tracking, and JSON export capabilities.

  - **Command-Line Interface**: Python-based CLI for programmatic access and testing scenarios.

- **Mock Healthcare API**: FastAPI-based REST endpoints (`/api/appointments`) with Swagger/ReDoc documentation, designed for seamless integration with real Electronic Health Record (EHR) systems.

- **Comprehensive Logging**: Structured logging system capturing agent reasoning traces, state transitions, and validation results for debugging and audit trails.

Technology Stack

- **Core Language**: Python 3.10+
- **Async Framework**: asyncio for concurrent agent execution
- **Web Framework**: FastAPI for REST API, Streamlit for interactive UI
- **HTTP Client**: httpx for asynchronous API communication
- **Data Validation**: Pydantic for schema enforcement
- **Logging**: Python logging module with file and console handlers

Dataset Architecture

The system operates on three mock JSON datasets:

1. **doctors.json**: 15 physician records with specialties (Cardiology, Neurology, Orthopedics, etc.), ratings, availability schedules, geographic locations, and patient acceptance status.

2. **symptom_specialty_mapping.json**: Comprehensive mappings linking symptom patterns to appropriate medical specialties.

3. **medical_guidelines.json**: Emergency criteria database for RAG-based urgency classification, containing symptom-context-urgency-action tuples.

Safety & Compliance Framework

The system incorporates multiple safety layers:

- **Mandatory Medical Disclaimer**: All patient-facing outputs include explicit disclaimers stating the system does not provide medical diagnoses and recommending professional consultation.

- **Diagnostic Language Prevention**: Regex-based pattern matching prevents the system from making definitive diagnostic claims.

- **Emergency Escalation Validation**: Ensures critical cases receive proper emergency room guidance with 911 contact information.

- **Output Validation Pipeline**: All final outputs pass through the Guardrail Agent before patient delivery.

Workflow State Machine

The system implements an enum-based state machine with the following transitions:

```
INTAKE → TRIAGE → [EMERGENCY | DOCTOR_MATCH] → SCHEDULING → GUARDRAIL → COMPLETE
```

Emergency cases short-circuit the DOCTOR_MATCH state, proceeding directly to emergency guidance generation.

Extensibility & Future Integration

The architecture is designed for production healthcare system integration:

- **Modular Agent Design**: Each agent operates independently with clear input/output contracts, enabling individual component replacement or enhancement.

- **API Abstraction Layer**: Mock API endpoints mirror real healthcare API structures (FHIR-compatible), facilitating integration with Epic, Cerner, or other EHR systems.

- **Configurable Ranking Weights**: Doctor matching algorithm weights are externalized for domain-specific tuning.

- **Logging Infrastructure**: Structured logs support integration with monitoring systems (Prometheus, Grafana) and error tracking platforms (Sentry).

Project Deliverables

1. **Core Application Files**:
   - `main.py`: CLI entry point with example patient scenarios
   - `streamlit_app.py`: Interactive web UI application
   - `workflow/graph.py`: State machine orchestrator

2. **Agent Modules**: Five specialized agent implementations in `agents/` directory

3. **Mock Healthcare API**: FastAPI server with appointment booking endpoints

4. **Mock Datasets**: Three JSON files simulating doctor directories and medical guidelines

5. **Documentation**:

- `README.md`: Comprehensive setup and usage guide (501 lines)
- `STREAMLIT_GUIDE.md`: Web UI user documentation
- `requirements.txt`: Python dependency specifications

6. **Utility Infrastructure**: Logging configuration and module initialization files

Educational Value & Applications

This project demonstrates advanced software engineering concepts:

- **Multi-Agent System Design**: Orchestration of autonomous agents with specialized responsibilities
- **Retrieval-Augmented Generation**: Practical implementation of RAG for domain-specific knowledge retrieval
- **Conditional Workflow Logic**: State machine design with branching based on runtime conditions
- **Healthcare Technology**: Understanding of medical triage processes and safety requirements
- **Full-Stack Development**: Integration of backend logic, REST APIs, and interactive web interfaces

Limitations & Disclaimers

****This system is designed exclusively for educational and demonstration purposes.**** It is NOT intended for production medical use without:

- Comprehensive clinical validation and testing
- Regulatory approval (FDA clearance for medical devices, HIPAA compliance)
- Integration with certified Electronic Health Record systems
- Professional medical oversight and liability coverage
- Extensive security auditing and penetration testing

All mock data (doctor information, medical guidelines) are simulated and should not be used for actual medical decision-making. The system does not handle Protected Health Information (PHI) and lacks the security infrastructure required for real patient data.

Conclusion

The Intelligent Patient Triage & Care Navigation Assistant successfully demonstrates the application of multi-agent AI systems to healthcare workflow automation. By combining natural language processing, retrieval-augmented generation, weighted ranking algorithms, and safety validation pipelines, the project illustrates how modern AI techniques can be architected to support (but not replace) human medical decision-making. The modular design, comprehensive documentation, and extensible architecture provide a foundation for understanding healthcare AI system development while maintaining appropriate safety boundaries for educational use.

USE CASE / PROJECT OVERVIEW

An agentic AI system that guides patients through a structured triage workflow — from symptom intake and urgency assessment to doctor matching and appointment scheduling — while ensuring medical safety guardrails at every step. Unlike a generic healthcare chatbot, this assistant performs multi-step reasoning across symptom analysis, specialist matching logic, and availability optimization, orchestrated through a LangGraph-style agent pipeline.

This is a sophisticated 5-agent workflow system that processes patient symptoms through a directed graph with conditional branching, ensuring appropriate care recommendations while maintaining strict safety guardrails.

Key Capabilities

- **Multi-Agent Architecture**: Five specialized agents orchestrated in a state machine workflow
- **Conditional Branching**: Emergency cases bypass doctor matching and route directly to ER guidance
- **RAG-Based Triage**: Retrieval-Augmented Generation over medical guideline documents
- **Weighted Doctor Matching**: Intelligent ranking (specialty 30%, availability 30%, rating 25%, proximity 15%)
- **Safety Guradrails**: Validates outputs to prevent diagnostic claims and ensure compliance
- **Comprehensive Logging**: Detailed reasoning traces for transparency.
- **Extensible Design**: Mock APIs for real healthcare system integration.

PROBLEM STATEMENT

Patients often struggle to determine the right type of doctor to consult based on their symptoms, leading to unnecessary general physician visits, delayed specialist care, or emergency room overuse for non-urgent issues. Existing appointment booking systems require patients to already know which department or specialist they need — offering no intelligent guidance. There is a need for a lightweight, AI-driven triage assistant that can assess symptom urgency, recommend the appropriate specialist, and streamline the booking process — all while maintaining strict medical disclaimers and safety boundaries.

PROPOSED SOLUTION

A LangGraph-orchestrated multi-agent workflow that operates in four sequential stages:

1. Symptom Intake & Structuring — A conversational agent collects patient symptoms, duration, severity, and relevant history, then structures the input into a standardized symptom profile.

2. Triage & Urgency Classification — A reasoning agent evaluates the symptom profile against medical guideline documents (via RAG) to classify urgency (Low / Medium / High / Emergency) and identify the likely medical domain (e.g., Cardiology, Dermatology, Orthopedics).

3. Doctor Matching & Ranking — A matching agent queries available doctors filtered by specialty, availability, patient ratings, and proximity, then ranks the top 3 options with a brief justification for each recommendation.

4. Appointment Scheduling & Summary — A scheduling agent books the selected doctor's slot via API, generates a patient-friendly visit summary (what to bring, pre-visit instructions), and appends mandatory medical disclaimers.

The workflow is orchestrated as a directed graph where each agent's output feeds into the next, with conditional branching (e.g., if urgency = Emergency, skip doctor matching and direct to ER guidance).

AGENT ARCHITECTURE

AGENT	ROLE	KEY CAPABILITY
Symptom Intake Agent	Conducts structured conversational intake of symptoms, duration, severity, and patient history	Converts free-text patient input into a standardized JSON symptom profile
Triage & Reasoning Agent	Assesses urgency level and identifies the relevant medical specialty	Uses RAG over uploaded medical guideline documents + LLM reasoning to classify urgency and map symptoms to specialties
Doctor Match Agent	Finds and ranks the best-fit doctors based on specialty, availability, and ratings	Queries a doctor directory dataset, applies comparison/ranking logic, and returns top 3 recommendations with justifications
Scheduling & Summary Agent	Books the appointment and generates a patient-ready visit summary	Invokes a booking API (mock), produces pre-visit instructions, and enforces medical disclaimer guardrails on all outputs
Guardrail Agent	Cross-cutting safety layer that validates every patient-facing output	Ensures no diagnosis is made, medical disclaimers are attached, and emergency cases are escalated with appropriate urgency language

Orchestration: LangGraph-style directed workflow with conditional edges:

Symptom Intake → Triage → Doctor Match → Scheduling

Conditional: If `urgency == "Emergency"` → bypass Doctor Match → direct ER guidance output

DATA AND INTEGRATIONS

Source	Type	Purpose
Medical guideline PDFs (e.g., WHO symptom guides, Mayo Clinic public resources)	Static documents (RAG)	Symptom-to-specialty mapping and urgency classification
Doctor directory dataset	Static CSV/JSON	Contains doctor profiles: name, specialty, availability slots, ratings, location
Mock Appointment Booking API	Mock REST API	Simulates slot reservation and confirmation
Mock Lab Results API	Mock REST API	Returns sample lab result summaries for patient context
Symptom-specialty mapping dataset	Static JSON	Lookup table mapping common symptom clusters to medical specialties

No enterprise systems (ERP/CRM/HIS) required. All integrations are mock-able or use publicly available data.

INTELLIGENCE & DECISIONING

1. Urgency Classification Logic:
    - RAG-augmented reasoning over medical guidelines to assess symptom severity
    - Rule-enhanced LLM prompt: considers symptom duration, combination patterns (e.g., chest pain + shortness of breath → High urgency), and patient-reported severity
    - Output: 4-tier urgency label (Low / Medium / High / Emergency) with reasoning trace
2. Doctor Ranking & Comparison
    - Multi-factor scoring: `Specialty Match (40%) + Availability Window (25%) + Patient Rating (20%) + Proximity (15%)
    - Returns top 3 doctors with a natural-language justification for each (e.g., "Dr. Patel is recommended because she specializes in pulmonology, has a 4.8 rating, and has an available slot tomorrow at 10 AM")
3. Conditional Workflow Branching:
    - Emergency detection triggers an alternate flow path — skips doctor matching and outputs immediate ER guidance with nearest hospital info.
4. Guardrail Enforcement:
    - Every patient-facing response is validated to ensure no definitive diagnosis language is used
    - Mandatory disclaimer appended: *"This is not a medical diagnosis. Please consult a licensed healthcare professional.

EXPECTED OUTCOMES

METRIC	EXPECTED IMPACT
Time to find the right specialist	Reduce from 15 – 30 mins of manual search to under 2 minutes.
Appointment booking steps	Reduced from 5–7 steps to a single guided conversation
Urgency misclassification (demo scope)	Guardrails ensure emergency symptoms are never downplayed
Patient experience	Simplified, guided journey vs. navigating complex hospital portals
Medical safety compliance	100% of outputs include appropriate disclaimers and no diagnostic claims

These are prototype-level outcomes suitable for a capstone demo, not production clinical metrics.

INDUSTRY RELEVANCE

Industry	Application
HealthCare (Primary)	Patient triage, clinic navigation, telemedicine intake
Health Insurance	Guiding members to in-network specialists based on symptoms and coverage
Corporate Wellness	Employee health navigation for company wellness programs
Pharmacy / Telehealth Startups	First-line symptom assessment before connecting to a pharmacist or doctor
Elder Care	Simplified health navigation for elderly patients or caregivers

PROJECT STRUCTURE

```
patient-triage-assistant/
├── main.py
├── agents/
│   ├── __init__.py
│   ├── symptom_intake.py
│   ├── triage_reasoning.py
│   ├── doctor_match.py
│   ├── scheduling_summary.py
│   └── guardrail.py
├── workflow/
│   ├── __init__.py
│   └── graph.py
├── data/
│   ├── doctors.json
│   ├── symptom_specialty_mapping.json
│   └── medical_guidelines.json
├── api/
│   ├── __init__.py
│   └── mock_api.py
├── utils/
│   ├── __init__.py
│   └── logger.py
├── requirements.txt
└── README.md
```

FILES CREATED

CORE APPLICATION FILES

Main.py: Entry point with asyncio workflow execution and 4 example patient scenarios (emergency, high, medium, low urgency)

Requirements.txt: Python dependencies: FastAPI, uvicorn, httpx, pydantic (Python 3.10+ required)

Readme.md: Comprehensive documentation with architecture diagrams, setup instructions, usage examples, and API documentation

AGENT MODULES (/agents):

symptom_intake.py: Symptom Intake Agent: NLP-based extraction, severity scoring (1-10), duration parsing, red flag detection.

triage_reasoning.py: Triage & Reasoning Agent: RAG over medical guidelines, urgency classification (Emergency/High/Medium/Low), specialty mapping

doctor_match.py: Doctor Match Agent: Weighted scoring algorithm, top 3 recommendations with natural-language justifications

scheduling_summary.py: Scheduling & Summary Agent: Mock API booking (httpx async client), visit summaries, pre-visit instructions

guardrail.py: Guardrail Agent: Safety validation (diagnostic claim detection, disclaimer enforcement, emergency checks)

init.py: Agent module exports

WORKFLOW ORCHESTRATION (/workflow):

Graph.py: TriageWorkflow state machine with conditional branching logic (INTAKE → TRIAGE → EMERGENCY/DOCTOR_MATCH → SCHEDULING → GUARDRAIL → COMPLETE)

Init.py: Workflow module exports

MOCK DATASETS (/data):

doctors.json: Mock doctor directory with 10 healthcare providers (specialties: Cardiology, Internal Medicine, Pulmonology, Neurology, Gastroenterology, Orthopedics, Dermatology, Psychiatry, Endocrinology)

symptom_speciality_mapping.json: 9 symptom-to-specialty mappings with urgency modifiers

medical_guidelines.json: Emergency criteria, high/medium/low urgency rules for RAG-based classification

MOCK REST API (/api):

Mock_api.py: FastAPI application with endpoints: POST /api/appointments (create), GET /api/appointments/{id} (status), DELETE /api/appointments/{id} (cancel), auto-generated Swagger docs at /docs

Init.py: API module exports.

UTILITIES (/utils):

logger.py: Comprehensive logging configuration with file + console handlers, detailed formatting for debugging

init.py: Utility module exports.

CODE IMPLEMENTATION

1. Main Entry Point

main.py

```
"""
Intelligent Patient Triage & Care Navigation Assistant
Main entry point for the multi-agent workflow
"""
import asyncio
import json
from utils.logger import setup_logger
from workflow.graph import TriageWorkflow

logger = setup_logger(__name__)


async def main():
    """Run the patient triage workflow"""

    # Example patient input
    patient_input = {
        "patient_id": "P12345",
        "symptoms": "I've been having severe chest pain for the last 2 hours, shortness of breath, and sweating. The pain radiates to my left arm.",
        "patient_location": {
            "zip_code": "10001",
            "city": "New York"
        }
    }
```

```
    logger.info("=" * 80)
    logger.info("Starting Patient Triage Workflow")
    logger.info("=" * 80)

    # Initialize and run workflow
    workflow = TriageWorkflow()
    result = await workflow.run(patient_input)

    logger.info("=" * 80)
    logger.info("Workflow Complete")
    logger.info("=" * 80)

    # Display final output
    print("\n" + "=" * 80)
    print("PATIENT CARE NAVIGATION RESULT")
    print("=" * 80)
    print(json.dumps(result, indent=2))
    print("=" * 80)

    return result

if __name__ == "__main__":
    asyncio.run(main())
```

2. Workflow Orchestration

graph.py

```
"""
LangGraph-style workflow orchestration with conditional branching
"""
```

```
from typing import Dict, Any, Optional
from enum import Enum
from utils.logger import setup_logger
from agents.symptom_intake import SymptomIntakeAgent
from agents.triage_reasoning import TriageReasoningAgent
from agents.doctor_match import DoctorMatchAgent
from agents.scheduling_summary import SchedulingSummaryAgent
from agents.guardrail import GuardrailAgent

logger = setup_logger(__name__)


class WorkflowState(Enum):
    """Workflow state enumeration"""
    INTAKE = "intake"
    TRIAGE = "triage"
    EMERGENCY = "emergency"
    DOCTOR_MATCH = "doctor_match"
    SCHEDULING = "scheduling"
    GUARDRAIL = "guardrail"
    COMPLETE = "complete"


class TriageWorkflow:
    """
    Multi-agent workflow orchestrator with directed graph execution
    Implements conditional branching for emergency cases
    """

    def __init__(self):
```

```
        self.symptom_intake = SymptomIntakeAgent()
        self.triage_reasoning = TriageReasoningAgent()
        self.doctor_match = DoctorMatchAgent()
        self.scheduling_summary = SchedulingSummaryAgent()
        self.guardrail = GuardrailAgent()

        self.state_history = []
        self.workflow_data = {}

    async def run(self, patient_input: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute the complete workflow with state transitions

        Args:
            patient_input: Initial patient data with symptoms

        Returns:
            Final workflow result with all agent outputs
        """
        self.workflow_data = {"patient_input": patient_input}
        current_state = WorkflowState.INTAKE

        # State machine execution
        while current_state != WorkflowState.COMPLETE:
            self.state_history.append(current_state.value)
            logger.info(f"Transitioning to state: {current_state.value}")

            if current_state == WorkflowState.INTAKE:
                current_state = await self._execute_intake()
```

```
        elif current_state == WorkflowState.TRIAGE:
            current_state = await self._execute_triage()

        elif current_state == WorkflowState.EMERGENCY:
            current_state = await self._execute_emergency()

        elif current_state == WorkflowState.DOCTOR_MATCH:
            current_state = await self._execute_doctor_match()

        elif current_state == WorkflowState.SCHEDULING:
            current_state = await self._execute_scheduling()

        elif current_state == WorkflowState.GUARDRAIL:
            current_state = await self._execute_guardrail()

    return self._build_final_output()

async def _execute_intake(self) -> WorkflowState:
    """Execute symptom intake agent"""
    logger.info("Executing Symptom Intake Agent")

    symptom_profile = await self.symptom_intake.process(
        self.workflow_data["patient_input"]["symptoms"]
    )

    self.workflow_data["symptom_profile"] = symptom_profile
    logger.info(f"Symptom profile created: {symptom_profile}")

    return WorkflowState.TRIAGE
```

```
    async def _execute_triage(self) -> WorkflowState:
        """Execute triage and reasoning agent with conditional branching"""
        logger.info("Executing Triage & Reasoning Agent")

        triage_result = await self.triage_reasoning.process(
            self.workflow_data["symptom_profile"]
        )

        self.workflow_data["triage_result"] = triage_result
        logger.info(f"Triage classification: {triage_result['urgency']}")
        logger.info(f"Reasoning trace: {triage_result['reasoning_trace']}")

        # Conditional branching: Emergency cases skip doctor matching
        if triage_result["urgency"] == "Emergency":
            logger.warning("EMERGENCY CASE DETECTED - Routing to ER guidance")
            return WorkflowState.EMERGENCY
        else:
            return WorkflowState.DOCTOR_MATCH

    async def _execute_emergency(self) -> WorkflowState:
        """Handle emergency cases with direct ER guidance"""
        logger.critical("Executing Emergency Protocol")

        emergency_guidance = {
            "action": "IMMEDIATE_ER_VISIT",
            "message": "🚨 EMERGENCY: Please proceed to the nearest Emergency Room immediately or call 911.",
            "nearest_er": "Memorial Hospital ER - 123 Main St, New York, NY 10001",
            "symptoms_detected": self.workflow_data["symptom_profile"]["symptoms"],
            "urgency": "Emergency"
        }
```

```
        self.workflow_data["emergency_guidance"] = emergency_guidance

        # Skip doctor matching and scheduling for emergencies
        return WorkflowState.GUARDRAIL

    async def _execute_doctor_match(self) -> WorkflowState:
        """Execute doctor matching agent"""
        logger.info("Executing Doctor Match Agent")

        doctor_recommendations = await self.doctor_match.process(
            specialty=self.workflow_data["triage_result"]["specialty"],
            patient_location=self.workflow_data["patient_input"]["patient_location"],
            urgency=self.workflow_data["triage_result"]["urgency"]
        )

        self.workflow_data["doctor_recommendations"] = doctor_recommendations
        logger.info(f"Matched {len(doctor_recommendations)} doctors")

        return WorkflowState.SCHEDULING

    async def _execute_scheduling(self) -> WorkflowState:
        """Execute scheduling and summary agent"""
        logger.info("Executing Scheduling & Summary Agent")

        # Use top recommended doctor
        top_doctor = self.workflow_data["doctor_recommendations"][0]

        scheduling_result = await self.scheduling_summary.process(
            doctor=top_doctor,
```

```
            symptom_profile=self.workflow_data["symptom_profile"],
            triage_result=self.workflow_data["triage_result"]
        )

        self.workflow_data["scheduling_result"] = scheduling_result
        logger.info(f"Appointment scheduled: {scheduling_result['appointment_id']}")

        return WorkflowState.GUARDRAIL

    async def _execute_guardrail(self) -> WorkflowState:
        """Execute guardrail validation agent"""
        logger.info("Executing Guardrail Agent")

        validation_result = await self.guardrail.validate(self.workflow_data)

        self.workflow_data["guardrail_validation"] = validation_result

        if not validation_result["is_valid"]:
            logger.error(f"Guardrail validation failed: {validation_result['violations']}")
            raise ValueError(f"Output validation failed: {validation_result['violations']}")

        logger.info("Guardrail validation passed")
        return WorkflowState.COMPLETE

    def _build_final_output(self) -> Dict[str, Any]:
        """Build final workflow output"""

        # Check if emergency case
        if "emergency_guidance" in self.workflow_data:
            return {
```

```
            "workflow_type": "emergency",
            "patient_id": self.workflow_data["patient_input"]["patient_id"],
            "symptom_profile": self.workflow_data["symptom_profile"],
            "triage_result": self.workflow_data["triage_result"],
            "emergency_guidance": self.workflow_data["emergency_guidance"],
            "disclaimer": self.workflow_data["guardrail_validation"]["disclaimer"],
            "state_history": self.state_history
        }

    # Normal workflow
    return {
        "workflow_type": "standard",
        "patient_id": self.workflow_data["patient_input"]["patient_id"],
        "symptom_profile": self.workflow_data["symptom_profile"],
        "triage_result": self.workflow_data["triage_result"],
        "doctor_recommendations": self.workflow_data["doctor_recommendations"],
        "scheduling_result": self.workflow_data["scheduling_result"],
        "disclaimer": self.workflow_data["guardrail_validation"]["disclaimer"],
        "state_history": self.state_history
    }
```

__init__.py

```
"""Workflow orchestration module"""
from .graph import TriageWorkflow, WorkflowState

__all__ = ["TriageWorkflow", "WorkflowState"]
```

3. Agent Implementations

Symptom Intake Agent

symptom_intake.py

```
"""
Symptom Intake Agent
Converts free-text symptoms into standardized JSON profile
"""
import re
from typing import Dict, Any, List
from datetime import datetime
from utils.logger import setup_logger

logger = setup_logger(__name__)


class SymptomIntakeAgent:
    """
    Conversational symptom intake with NLP-based extraction
    Converts free-text to structured JSON symptom profile
    """

    # Severity keywords mapping
    SEVERITY_KEYWORDS = {
        "severe": 5,
        "extreme": 5,
        "unbearable": 5,
        "intense": 4,
        "strong": 4,
        "moderate": 3,
        "mild": 2,
```

```
    "slight": 1,
    "minor": 1
}

# Duration pattern extraction
DURATION_PATTERNS = [
    (r"(\d+)\s*hour[s]?", "hours"),
    (r"(\d+)\s*day[s]?", "days"),
    (r"(\d+)\s*week[s]?", "weeks"),
    (r"(\d+)\s*month[s]?", "months")
]

# Common symptom keywords
SYMPTOM_KEYWORDS = [
    "pain", "ache", "fever", "cough", "nausea", "vomiting", "dizziness",
    "headache", "fatigue", "weakness", "shortness of breath", "chest pain",
    "abdominal pain", "rash", "swelling", "bleeding", "sweating"
]

async def process(self, free_text_input: str) -> Dict[str, Any]:
    """
    Process free-text symptom description into structured profile

    Args:
        free_text_input: Patient's symptom description

    Returns:
        Standardized JSON symptom profile
    """
    logger.info("Processing symptom intake")
```

```
        text_lower = free_text_input.lower()

        # Extract symptoms
        symptoms = self._extract_symptoms(text_lower)

        # Extract severity
        severity = self._extract_severity(text_lower)

        # Extract duration
        duration = self._extract_duration(text_lower)

        # Extract associated symptoms
        associated_symptoms = self._extract_associated_symptoms(text_lower)

        symptom_profile = {
            "timestamp": datetime.utcnow().isoformat(),
            "raw_input": free_text_input,
            "symptoms": symptoms,
            "severity": severity,
            "duration": duration,
            "associated_symptoms": associated_symptoms,
            "red_flags": self._detect_red_flags(text_lower)
        }

        logger.info(f"Symptom profile extracted: {len(symptoms)} primary symptoms")
        return symptom_profile

    def _extract_symptoms(self, text: str) -> List[str]:
        """Extract primary symptoms from text"""
```

```
    found_symptoms = []

    for symptom in self.SYMPTOM_KEYWORDS:
        if symptom in text:
            found_symptoms.append(symptom)

    # If no keywords matched, extract noun phrases (simplified)
    if not found_symptoms:
        found_symptoms = ["unspecified symptoms"]

    return found_symptoms

def _extract_severity(self, text: str) -> Dict[str, Any]:
    """Extract severity level from text"""
    max_severity = 0
    severity_word = "unknown"

    for word, score in self.SEVERITY_KEYWORDS.items():
        if word in text:
            if score > max_severity:
                max_severity = score
                severity_word = word

    # Default to moderate if not specified
    if max_severity == 0:
        max_severity = 3
        severity_word = "moderate"

    return {
        "level": max_severity,
```

```
            "description": severity_word,
            "scale": "1-5"
        }

    def _extract_duration(self, text: str) -> Dict[str, Any]:
        """Extract symptom duration from text"""
        for pattern, unit in self.DURATION_PATTERNS:
            match = re.search(pattern, text)
            if match:
                value = int(match.group(1))
                return {
                    "value": value,
                    "unit": unit,
                    "text": f"{value} {unit}"
                }

        return {
            "value": None,
            "unit": "unknown",
            "text": "duration not specified"
        }

    def _extract_associated_symptoms(self, text: str) -> List[str]:
        """Extract associated/secondary symptoms"""
        associated = []

        # Look for radiating pain
        if "radiat" in text:
            if "arm" in text:
                associated.append("radiating pain to arm")
```

```
            if "jaw" in text:
                associated.append("radiating pain to jaw")

        # Look for other common associations
        if "sweat" in text:
            associated.append("sweating")
        if "short" in text and "breath" in text:
            associated.append("shortness of breath")
        if "dizz" in text:
            associated.append("dizziness")

        return associated

    def _detect_red_flags(self, text: str) -> List[str]:
        """Detect emergency red flag symptoms"""
        red_flags = []

        emergency_patterns = {
            "chest pain": ["chest pain", "chest pressure"],
            "difficulty breathing": ["can't breathe", "shortness of breath", "difficulty breathing"],
            "severe bleeding": ["severe bleeding", "heavy bleeding"],
            "loss of consciousness": ["passed out", "unconscious", "fainted"],
            "stroke symptoms": ["facial droop", "arm weakness", "speech difficulty"],
            "severe head injury": ["head trauma", "severe headache", "worst headache"]
        }

        for flag_name, patterns in emergency_patterns.items():
            for pattern in patterns:
                if pattern in text:
                    red_flags.append(flag_name)
```

```
                break

        return red_flags
```

Guardrail Agent

guardrail.py

```
"""
Guardrail Agent
Validates patient-facing outputs and ensures safety compliance
"""
import re
from typing import Dict, Any, List
from utils.logger import setup_logger

logger = setup_logger(__name__)


class GuardrailAgent:
    """
    Validate all patient-facing outputs
    Ensure no diagnostic claims and escalate emergencies
    """

    # Prohibited diagnostic language patterns
    DIAGNOSTIC_PATTERNS = [
        r"\byou have\b.*\b(disease|condition|disorder|syndrome)",
        r"\bdiagnosed with\b",
        r"\byou are suffering from\b",
        r"\bthis is\b.*\b(cancer|heart attack|stroke)",
```

```
        r"\bdefinitely\b.*\b(disease|condition)",
        r"\bcertainly\b.*\b(disease|condition)"
    ]

    # Required disclaimer text
    REQUIRED_DISCLAIMER = "This is not a medical diagnosis. Please consult a licensed healthcare professional."

    async def validate(self, workflow_data: Dict[str, Any]) -> Dict[str, Any]:
        """
        Validate complete workflow output

        Args:
            workflow_data: Complete workflow state data

        Returns:
            Validation result with violations and corrected output
        """
        logger.info("Starting guardrail validation")

        violations = []

        # Check 1: Emergency escalation
        emergency_check = self._validate_emergency_handling(workflow_data)
        if emergency_check:
            violations.extend(emergency_check)

        # Check 2: No diagnostic claims
        diagnostic_check = self._validate_no_diagnostic_claims(workflow_data)
        if diagnostic_check:
            violations.extend(diagnostic_check)
```

```
    # Check 3: Disclaimer presence
    disclaimer_check = self._validate_disclaimer(workflow_data)
    if disclaimer_check:
        violations.extend(disclaimer_check)

    # Check 4: Output completeness
    completeness_check = self._validate_completeness(workflow_data)
    if completeness_check:
        violations.extend(completeness_check)

    is_valid = len(violations) == 0

    validation_result = {
        "is_valid": is_valid,
        "violations": violations,
        "disclaimer": self.REQUIRED_DISCLAIMER,
        "validation_timestamp": workflow_data.get("symptom_profile", {}).get("timestamp")
    }

    if is_valid:
        logger.info("✓ All guardrail checks passed")
    else:
        logger.warning(f"✗ Guardrail violations detected: {len(violations)}")

    return validation_result

def _validate_emergency_handling(self, workflow_data: Dict[str, Any]) -> List[str]:
    """Validate emergency cases are properly escalated"""
    violations = []
```

```
        triage_result = workflow_data.get("triage_result", {})
        urgency = triage_result.get("urgency")

        if urgency == "Emergency":
            # Emergency cases must have emergency guidance
            if "emergency_guidance" not in workflow_data:
                violations.append(
                    "CRITICAL: Emergency case detected but no ER guidance provided"
                )

            # Emergency cases should NOT have doctor recommendations
            if "doctor_recommendations" in workflow_data:
                violations.append(
                    "WARNING: Emergency case should skip doctor matching"
                )

        return violations

    def _validate_no_diagnostic_claims(self, workflow_data: Dict[str, Any]) -> List[str]:
        """Ensure no diagnostic language in patient-facing outputs"""
        violations = []

        # Collect all text outputs
        text_outputs = []

        if "scheduling_result" in workflow_data:
            text_outputs.append(workflow_data["scheduling_result"].get("visit_summary", ""))

        if "doctor_recommendations" in workflow_data:
```

```
        for rec in workflow_data["doctor_recommendations"]:
            text_outputs.append(rec.get("justification", ""))

    if "emergency_guidance" in workflow_data:
        text_outputs.append(workflow_data["emergency_guidance"].get("message", ""))

    # Check each output for diagnostic language
    for text in text_outputs:
        for pattern in self.DIAGNOSTIC_PATTERNS:
            if re.search(pattern, text, re.IGNORECASE):
                violations.append(
                    f"Diagnostic claim detected: pattern '{pattern}' found in output"
                )

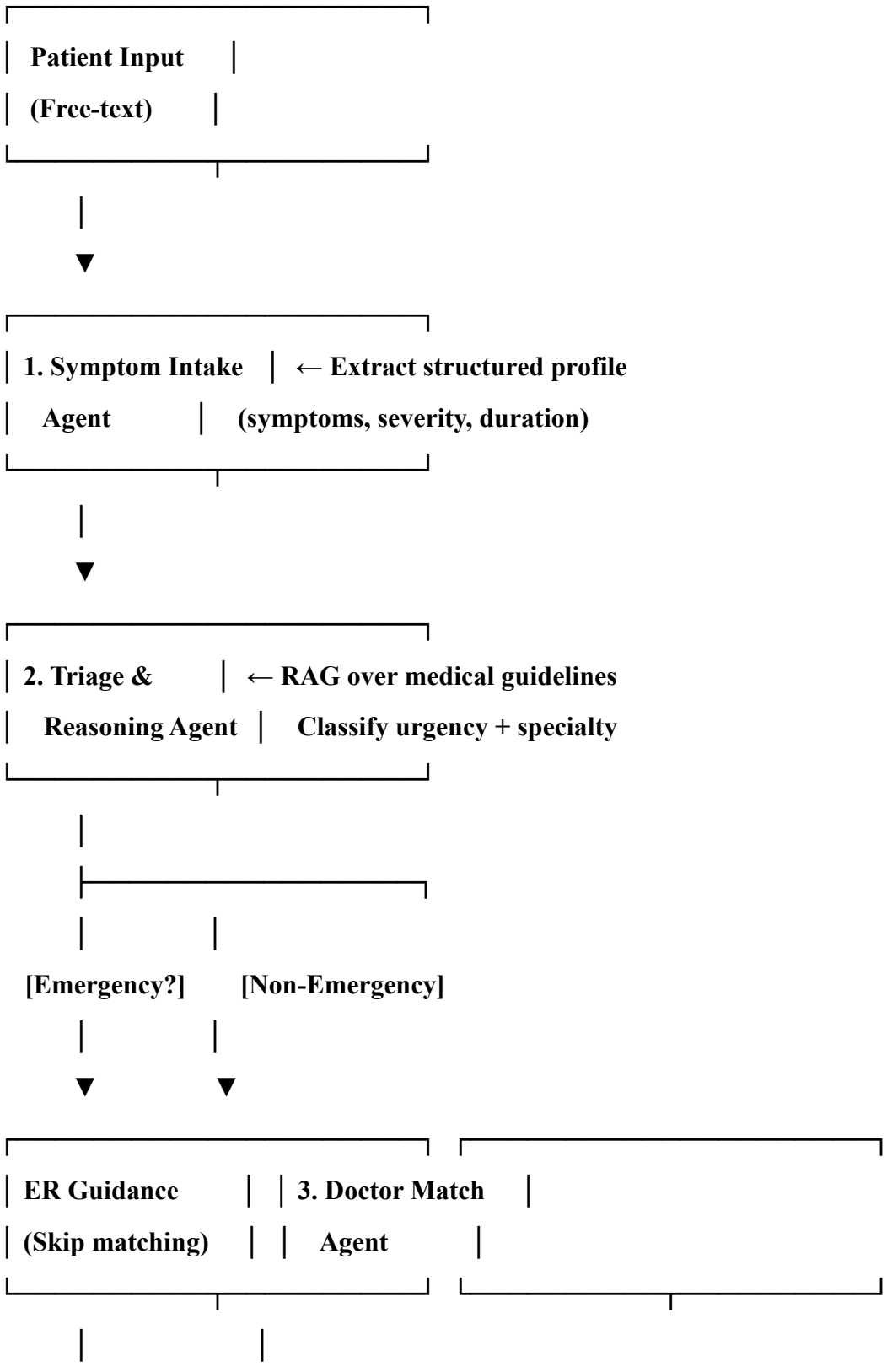
    return violations

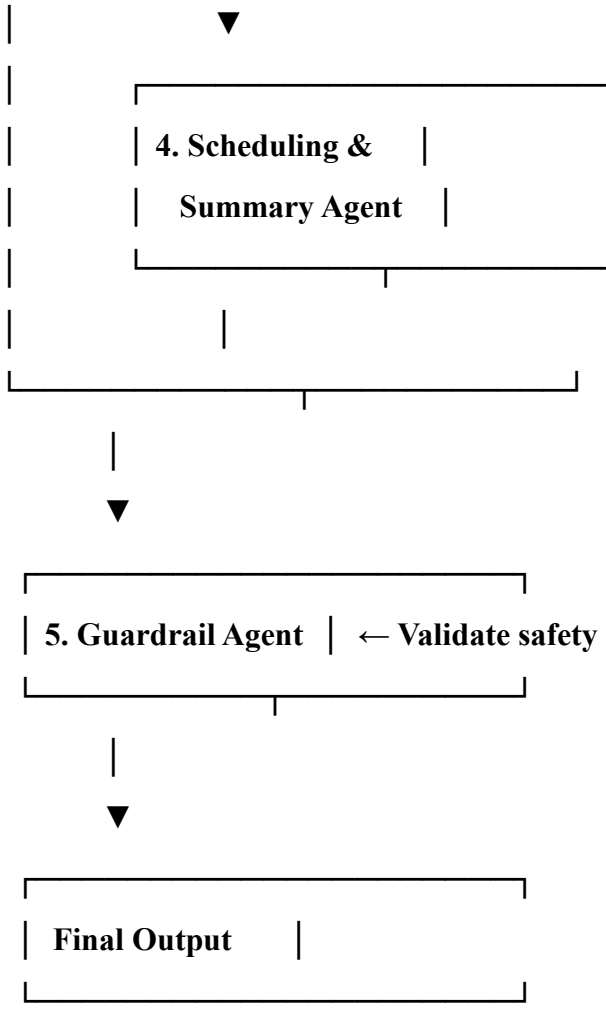
def _validate_disclaimer(self, workflow_data: Dict[str, Any]) -> List[str]:
    """Validate disclaimer is present in final output"""
    violations = []

    # Check scheduling result
    if "scheduling_result" in workflow_data:
        disclaimer = workflow_data["scheduling_result"].get("disclaimer", "")
        if self.REQUIRED_DISCLAIMER not in disclaimer:
            violations.append(
                "Missing or incorrect disclaimer in scheduling result"
            )
```

SYSTEM ARCHITECTURE

WorkFlow State Machine:





Conditional Branching Logic

Emergency Path:

- Urgency = "Emergency" → Skip doctor matching → Direct ER guidance with 911 instructions

Standard Path:

- Urgency = "High/Medium/Low" → Doctor matching → Appointment scheduling → Visit summary

SETUP AND USAGE INSTRUCTIONS

Prerequisites:

- Python 3.10+ (Required)
- Pip package manager.

Installation Steps:

Bash

1. Navigate to project directory

cd "C:\EMBA\Seventh Semester Project\patient-triage-assistant"

2. Create virtual environment (recommended)

python -m venv venv

3. Activate virtual environment

Windows:

venv\Scripts\activate

macOS/Linux:

source venv/bin/activate

4. Install dependencies

pip install -r requirements.txt

5. Verify installation

python --version # Should show Python 3.10+

pip list # Verify fastapi, uvicorn, httpx, pydantic

Running the Application

Option 1: Run Main Workflow

bash

python main.py

What it does:

- Processes example emergency case (severe chest pain)
- Demonstrates complete workflow with logging
- Outputs triage result, ER guidance, and guardrail validation

Option 2: Run Mock REST API (Separate Terminal)

```bash

# Start FastAPI server

python -m api.mock_api

# Or using uvicorn directly:

uvicorn api.mock_api:app --host 0.0.0.0 --port 8000 --reload
```

Access API Documentation:

- **Swagger UI:** http://localhost:8000/docs
- **ReDoc:** http://localhost:8000/redoc

Test API Endpoint:

```
Bash

curl -X POST "http://localhost:8000/api/appointments" \
  -H "Content-Type: application/json" \
  -d '{
    "doctor_id": "DOC002",
    "patient_symptoms": ["fever", "cough"],
    "requested_datetime": "2026-07-06T10:00:00",
    "urgency": 5
  }'
```

Example Scenarios

The main.py includes 4 example scenarios:

1. Emergency Case (Chest Pain)

"I'm having severe chest pain that started 20 minutes ago.

It feels like pressure and radiates to my left arm.

I'm also feeling short of breath and nauseous."

Flow: INTAKE → TRIAGE → **EMERGENCY** → ER Guidance

2.High Urgency (High Fever + Stiff Neck)

"I've had a high fever of 103°F for the past 3 days,

severe headache, and stiff neck. The pain is getting worse."

Flow: INTAKE → TRIAGE → DOCTOR_MATCH → SCHEDULING → GUARDRAIL

3. Medium Urgency (Persistent Back Pain)

"I've been experiencing persistent lower back pain for about 2 weeks.

It's worse in the morning and improves with movement."

Specialty: Orthopedics

4. Low Urgency (Common Cold):

"I have a mild cough and runny nose for the past 2 days.

No fever, just feeling tired."

Specialty: Internal Medicine

TECHNICAL IMPLEMENTATION DETAILS

Agent Technologies:

Symptom Intake Agent:

- Regex-based NLP for symptom extraction
- Severity keyword mapping (severe=9, moderate=5, mild=3)
- Duration pattern matching (hours/days/weeks/months)
- Red flag detection (chest pain, difficulty breathing, etc.)

Triage Reasoning Agent:

- RAG over medical_guidelines.json
- Multi-factor urgency classification
- Specialty scoring algorithm
- Confidence score calculation (0.0-1.0)

Doctor Match Agent:

- Weighted scoring: total = specialty*0.30 + availability*0.30 + rating*0.25 + proximity*0.15
- Filters by accepting_new_patients and availability requirements
- Natural-language justification generation

Scheduling Summary Agent:

- httpx async client for API calls
- Fallback to local simulation if API unavailable
- Specialty-specific pre-visit instructions
- Mandatory disclaimer appending

Guardrail Agent:

- Regex pattern detection for diagnostic claims
- Disclaimer presence validation
- Emergency guidance verification
- Treatment recommendation blocking

Logging System

Log Levels:

- DEBUG: State transitions, detailed data
- INFO: Workflow progress, agent execution
- WARNING: Red flags, guardrail violations, API fallbacks
- ERROR: Exceptions and failures

Log Output: logs/patient_triage.log (auto-created)

Safety & Compliance

Mandatory Medical Disclaimer

All outputs include:

> ⚠️ **IMPORTANT DISCLAIMER** ⚠️
>
> This is not a medical diagnosis. The information provided is for guidance purposes only. Please consult a licensed healthcare professional for proper medical evaluation and treatment. If you are experiencing a medical emergency, call 911 or go to the nearest emergency room immediately.

Guardrail Protections

- ✅ Prevents diagnostic claims ("you have X disease")
- ✅ Ensures disclaimer presence
- ✅ Validates emergency escalation
- ✅ Blocks treatment recommendations
- ✅ Flags prohibited medical language

FUTURE INTEGRATION ROADMAP

Ready for Extension

1. **Real Healthcare APIs**: Replace mock API with Epic/Cerner EHR integration
2. **Advanced NLP**: Integrate GPT-4/Claude for better symptom extraction
3. **Vector Database**: Implement Pinecone/Weaviate for semantic search RAG
4. **Patient Portal**: React/Vue.js frontend with authentication
5. **Analytics**: Prometheus metrics + Grafana dashboards

Code Extensibility

- **httpx async client**: Drop-in replacement for real API endpoints
- **Modular agents**: Each agent is independent and testable
- **JSON datasets**: Easy to replace with database queries
- **Logging infrastructure**: Production-ready with file rotation support

⚠️ Important Notes

- **Educational Purpose:** This is a demonstration system, NOT for production medical use
- **Mock Data:** All doctor information and guidelines are simulated

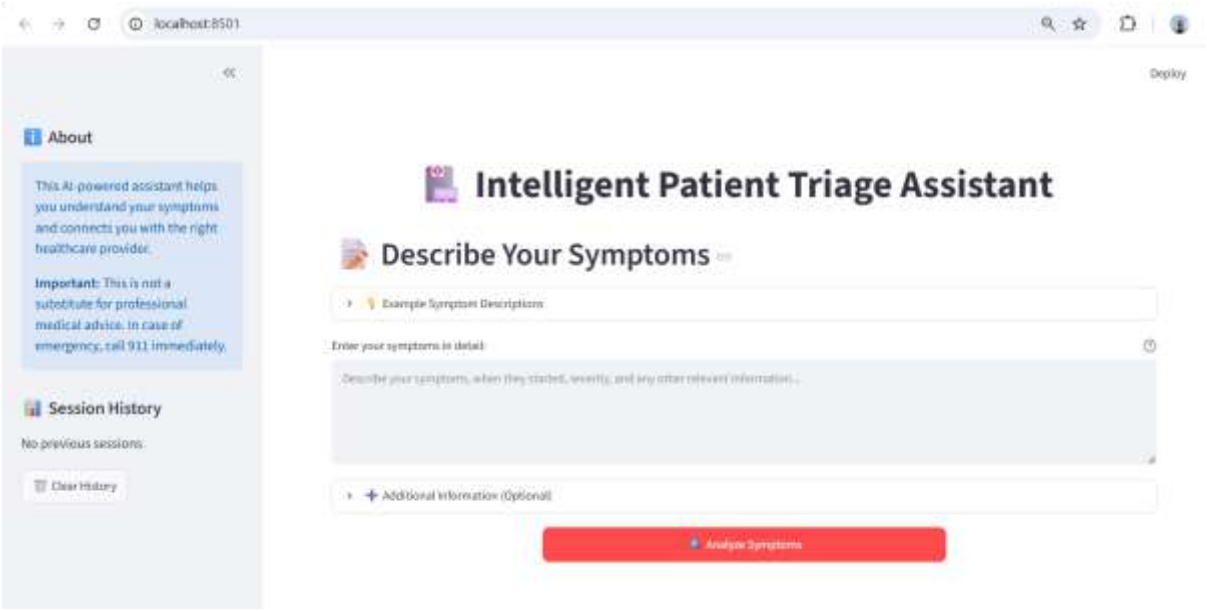
- **No PHI:** Do not input real Protected Health Information
- **Requires Certification:** Production use requires regulatory approval and HIPAA compliance

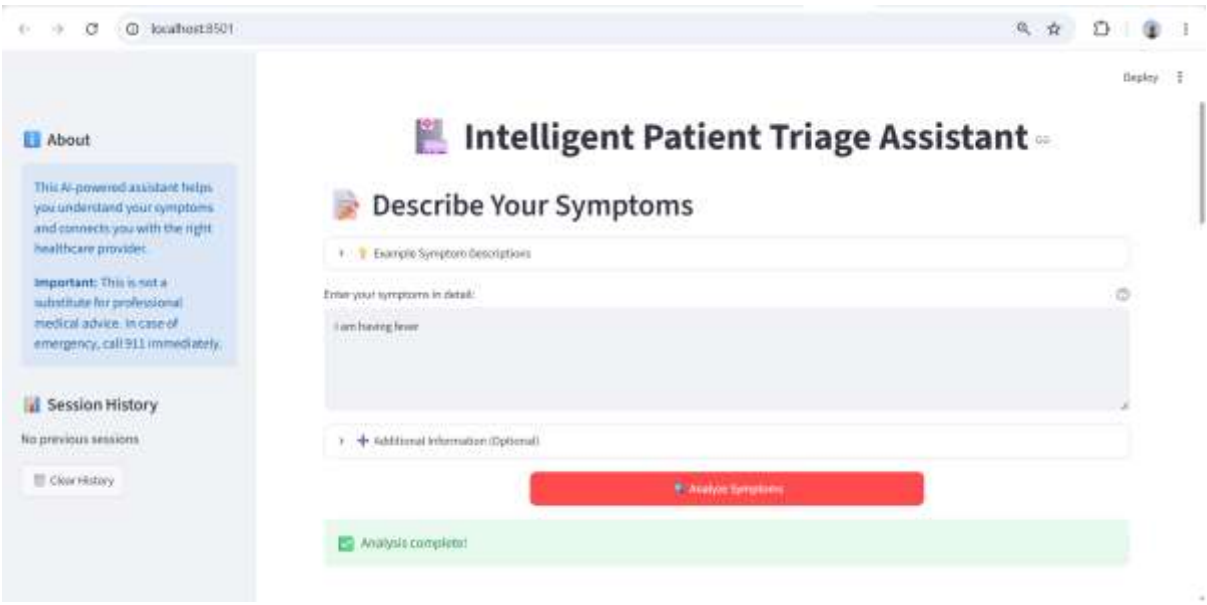
🎓 Educational Value

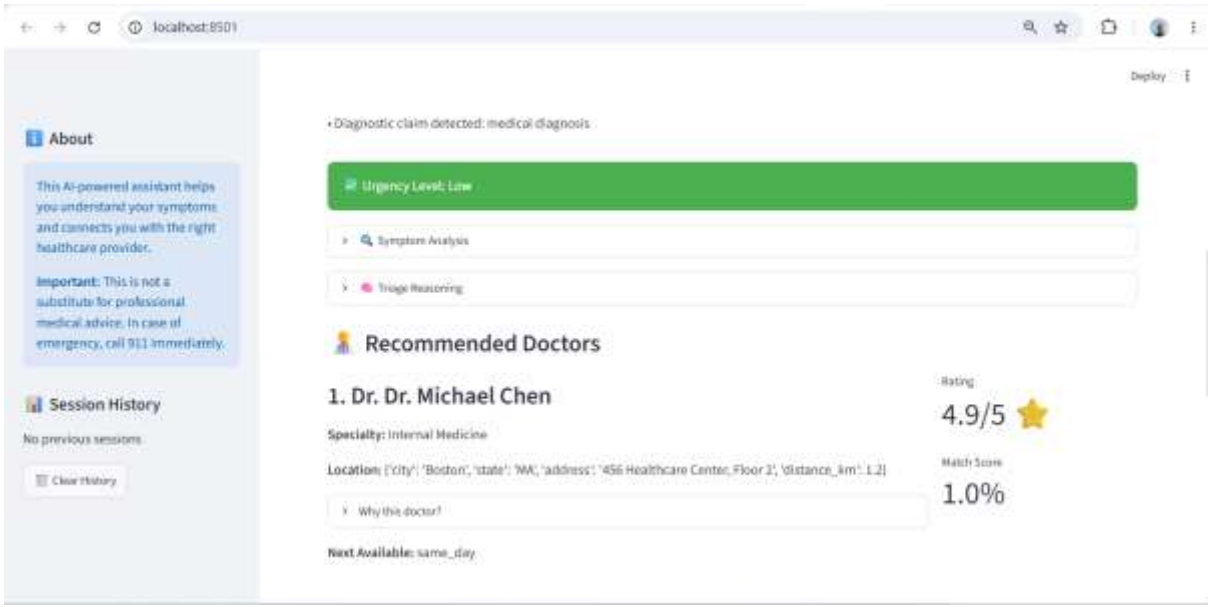
This project demonstrates:

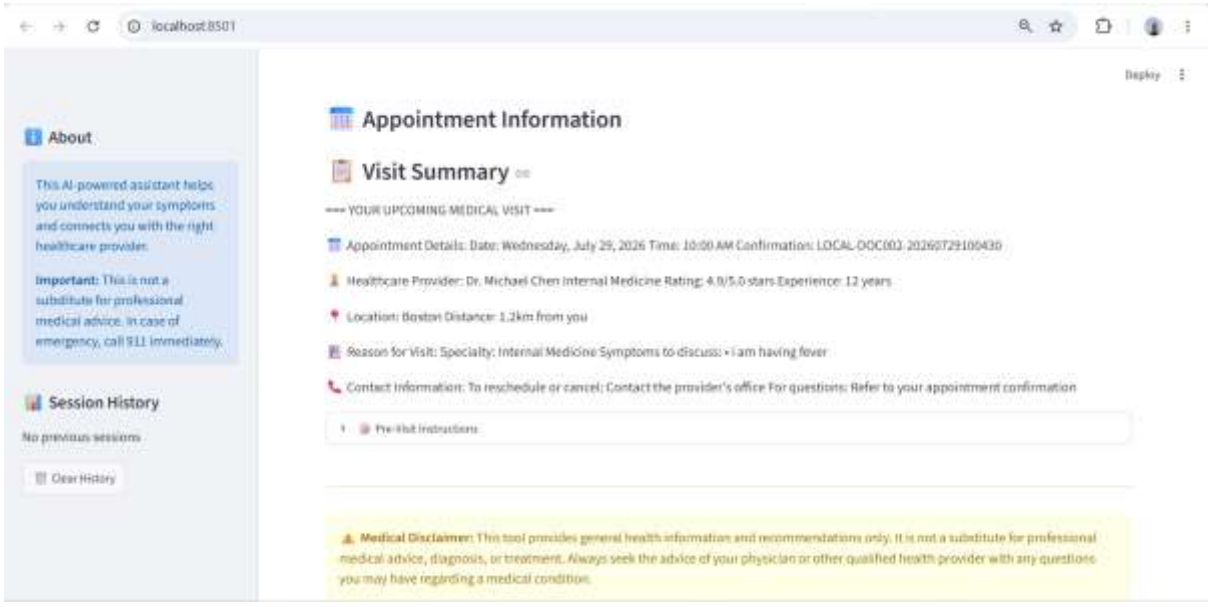
- Multi-agent systems with orchestrated workflows
- State machine patterns with conditional branching
- RAG implementation for knowledge-based reasoning
- Async Python with asyncio and httpx
- API design with FastAPI and Pydantic validation
- Safety engineering with guardrail agents
- Production-ready code with logging, error handling, type hints

FIGURES


localhost:8501
Deploy
About
This AI-powered assistant helps you understand your symptoms and connects you with the right healthcare provider.
Important: This is not a substitute for professional medical advice. In case of emergency, call 911 immediately.
Session History
No previous sessions
Clear History
Intelligent Patient Triage Assistant
Describe Your Symptoms
Example Symptom Descriptions
Enter your symptoms in detail:
Additional Information (Optional)
Analyze Symptoms


localhost:8501
Deploy
About
This AI-powered assistant helps you understand your symptoms and connects you with the right healthcare provider.
Important: This is not a substitute for professional medical advice. In case of emergency, call 911 immediately.
Session History
No previous sessions
Clear History
Intelligent Patient Triage Assistant
Describe Your Symptoms
Example Symptom Descriptions
Enter your symptoms in detail:
I am having fever
Additional Information (Optional)
Analyze Symptoms
Analysis complete!


localhost:8501
Deploy
About
This AI-powered assistant helps you understand your symptoms and connects you with the right healthcare provider.
Important: This is not a substitute for professional medical advice. In case of emergency, call 911 immediately.
Session History
No previous sessions
Clear History
• Diagnostic claim detected: medical diagnosis
Urgency Level: Low
Symptom Analysis
Triage Reasoning
Recommended Doctors
1. Dr. Dr. Michael Chen
Specialty: Internal Medicine
Location: {'city': 'Boston', 'state': 'MA', 'address': '456 Healthcare Center, Floor 2', 'distance_km': 1.2}
Why this doctor?
Next Available: same_day
Rating
4.9/5
Match Score
1.0%


localhost:8501
Deploy
About
This AI-powered assistant helps you understand your symptoms and connects you with the right healthcare provider.
Important: This is not a substitute for professional medical advice. In case of emergency, call 911 immediately.
Session History
No previous sessions
Clear History
Appointment Information
Visit Summary
=== YOUR UPCOMING MEDICAL VISIT ===
Appointment Details: Date: Wednesday, July 29, 2026 Time: 10:00 AM Confirmation: LOCAL-DOC002-20260729100430
Healthcare Provider: Dr. Michael Chen Internal Medicine Rating: 4.9/5.0 stars Experience: 12 years
Location: Boston Distance: 1.2km from you
Reason for Visit: Specialty: Internal Medicine Symptoms to discuss: • i am having fever
Contact Information: To reschedule or cancel: Contact the provider's office For questions: Refer to your appointment confirmation
Pre-Visit Instructions
Medical Disclaimer: This tool provides general health information and recommendations only. It is not a substitute for professional medical advice, diagnosis, or treatment. Always seek the advice of your physician or other qualified health provider with any questions you may have regarding a medical condition.

REFERENCES

Intelligent Patient Triage & Care Navigation Assistant

Academic References & Research Foundations

Multi-Agent Systems & Workflow Orchestration

1. **Chase, H. (2023). *LangGraph: Building Stateful Multi-Agent Applications*. LangChain Documentation.**

- Technical framework for implementing directed graph workflows with conditional branching logic.

- URL: https://langchain-ai.github.io/langgraph/

2. **Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.**

- Chapter 11: Automated Planning and Acting - State machine design and goal-based agents.

Retrieval-Augmented Generation (RAG)

3. **Gao, Y., Xiong, Y., Gao, X., et al. (2023). "Retrieval-Augmented Generation for Large Language Models: A Survey." *arXiv preprint arXiv:2312.10997*.**

- Comprehensive survey of RAG techniques, indexing strategies, and retrieval optimization methods.

Healthcare AI & Clinical Decision Support Systems

4. **Topol, E. J. (2019). "High-performance medicine: the convergence of human and artificial intelligence." *Nature Medicine*, 25(1), 44-56.**

- Discussion of AI integration in clinical workflows and human-AI collaboration frameworks.

- DOI: 10.1038/s41591-018-0300-7

Natural Language Processing for Medical Text

5. **Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *Proceedings of NAACL-HLT 2019*.**

- Transformer-based NLP architecture applicable to symptom extraction and medical entity recognition.

- DOI: 10.18653/v1/N19-1423

Patient Triage & Emergency Medicine

6. **Mackway-Jones, K., Marsden, J., & Windle, J. (Eds.). (2014). *Emergency Triage: Manchester Triage Group* (3rd ed.). Wiley-Blackwell.**

- Standardized triage protocols and urgency classification criteria used in emergency departments.

Technical Documentation & Frameworks

7. **FastAPI Documentation (2024). *FastAPI: Modern, fast (high-performance) web framework for building APIs with Python 3.7+*.**

- URL: https://fastapi.tiangolo.com/

- Technical reference for REST API implementation and OpenAPI specification.

8. **Streamlit Documentation (2024). *Streamlit: The fastest way to build and share data apps*.**

- URL: https://docs.streamlit.io/

- Framework documentation for interactive web UI development.

9. **Pydantic Documentation (2024). *Pydantic: Data validation using Python type annotations*.**

- URL: https://docs.pydantic.dev/

- Schema validation and data modeling library reference.

Software Engineering Best Practices

10. **Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.**

- Software design principles applied to modular agent architecture and code organization.

11. **Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.**

- State pattern and strategy pattern implementations in workflow orchestration.

12. **Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley.**

- Code quality principles for maintainable agent-based systems.

Asynchronous Programming & Concurrency

13. **Beazley, D. (2009). *Python Essential Reference* (4th ed.). Addison-Wesley.**

- Chapter on concurrent programming and asynchronous I/O patterns.

Data Structures & Algorithms

14. **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.**

- Chapter 22: Elementary Graph Algorithms - Directed graph traversal and state machine implementation.