BIL 481 – Quality Assurance Document

Group Members

Helen Parlar – Artificial Intelligence Engineering

Mete Kılıç – Artificial Intelligence Engineering

Ahmet Çakmak – Medicine & Artificial Intelligence Engineering

TASK MATRIX

Section	**Description**	**Contributor**
1. QA Strategy	Overview of testing approach, automated vs. manual testing strategies.	Helen Parlar
2. Quality Factors	Definition of performance, security, reliability, and usability metrics.	Ahmet Çakmak
3. Test Plan	Definition of specific test cases (Database, API, UI) and expected outcomes.	Mete Kılıç
4. Bug Tracking	Description of the issue tracking process via GitHub.	All Members

Table of Contents

1. Quality Assurance Strategy

1.1. Overview

The quality assurance strategy for the "Smart Clinical Appointment System" focuses on ensuring data integrity, system reliability, and accurate AI-driven recommendations. Since the system handles booking transactions and medical symptom analysis, our primary goal is to prevent data conflicts (e.g., double bookings) and ensure the AI provides relevant polyclinic suggestions.

1.2. Testing Methodologies

We will employ a "Shift-Left" testing approach, integrating testing early in the development lifecycle.

- **Unit Testing:** Used to verify individual backend components, specifically the AI prompt generation logic and Django model constraints.
- **Integration Testing:** Validates the interaction between the Django Backend, MySQL Database, and the OpenAI API. This is critical for ensuring that appointment data is correctly committed to the database.
- **Usability Testing:** Conducted manually to ensure that patients can easily navigate the symptom entry and booking flow without confusion.
- **Database Integrity Testing:** Specifically targets the database constraints (Unique Keys, Foreign Keys, Check Constraints) to ensure invalid data cannot be entered.

1.3. Automated vs. Manual Testing

- **Automated Testing:** Backend API endpoints and database constraints will be tested using Django's built-in test runner (manage.py test) and Postman collections.
- **Manual Testing:** The Frontend user interface (React/Vue), error message clarity, and the subjective quality of AI responses will be tested manually by the team.

2. Quality Factors & Metrics

- We have defined four key quality factors derived from our "Real-Life Constraints" analysis.

Quality Factor	Description	Measurement Metric
Reliability (Data Integrity)	The system's ability to prevent conflicting data entries,specifically double-booked appointments.	**Zero (0)** occurrences of overlapping appointments for the same doctor/time slot (Enforced by DB Unique Constraints).
Performance	The speed at which the system analyzes symptoms and retrieves doctor schedules.	Average API response time for symptom analysis **< 5 seconds**.
Security	Adherence to role-based access control and data validation rules.	**100%** of registered Doctor accounts must match the @doc.com email pattern.
Usability	The ease with which a new patient can complete the appointment process.	Average time to book an appointment **< 3 minutes** (measured via user trials).

3. Test Plan

The following test cases cover the critical paths of the application, including database constraints, AI logic, and user flow.

Test Case ID	Scenario	Steps	Expected Result
TC-01	**Concurrency / Double Booking Prevention**	1. User A sends a request to book **Dr. Smith** at **14:00**. 2. User B sends a request to book **Dr. Smith** at **14:00** at the exact same time.	The database processes the first request successfully. The second request is rejected with a **"Slot already booked"** error due to the OneToOne relationship and UNIQUE constraint on the schedule_id.
TC-02	**Doctor Role Security Verification**	1. An admin attempts to register a new user with the role **"Doctor"**. 2. The email provided is test_doctor@gmail.com.	The system rejects the registration. The Database CHECK constraint (chk_email_role) fails because the email does not end with the required @doc.com domain.
TC-03	**AI Logic & JSON Response Parsing**	1. User logs in and enters the symptom: *"Şiddetli kulak ağrısı ve baş dönmesi"* (Severe ear pain and dizziness). 2. User clicks "Analyze".	The backend sends the prompt to **gpt-4o-mini** and receives a valid **JSON** object (e.g., {"clinics": ["KBB"]}). The system parses this JSON and correctly redirects the user to the **KBB (ENT)** clinic page.

Test Case ID	Scenario	Steps	Expected Result
TC-04	**Invalid Data Integrity (Foreign Key)**	1. A direct API call is made to create an appointment. 2. The payload includes a doctor_id (e.g., 9999) that does not exist in the Users/Doctors table.	The operation fails with a **Foreign Key Constraint Violation** error. No orphan record is created in the Appointments table, ensuring data consistency.
TC-05	**Symptom Checklist Retrieval**	1. User manually selects **"Kardiyoloji"** (Cardiology) from the dropdown list. 2. System requests the symptom checklist.	The system queries the Symptoms table and retrieves the specific 10 symptoms (e.g., "Göğüs ağrısı", "Çarpıntı") mapped to the Cardiology clinic ID.

4. Bug Tracking

4.1. Workflow We utilize **GitHub Issues** for tracking and managing defects found during testing.

4.2. Reporting Format Each bug report must include the following details:

- **Title:** Concise summary of the error (e.g., *"AI API timeout on long symptom text"*).
- **Labels:** bug, backend, frontend, database, critical.
- **Description:** Detailed explanation of the issue.
- **Steps to Reproduce:**
    1. Go to page X.
    2. Enter data Y.
    3. Click button Z.
- **Expected vs. Actual Result:** What should have happened vs. what actually happened.
- **Screenshots/Logs:** Visual proof or server logs of the error.

4.3. Resolution Process

1. Issue is assigned to a team member via the Task Matrix.
2. Developer fixes the issue and pushes the code to a feature branch.
3. Pull Request is created referencing the Issue ID (e.g., Fixes #12).
4. The issue is closed only after verification.