Part A – System Design (Written):

1. System Architecture

Type: RESTful Web Application (Client-Server Model)

Components:

- **Frontend:**
  - Built with HTML, CSS, and JavaScript (or React)
  - Interfaces for teachers to create assignments, students to submit assignments, and teachers to view submissions
  - Communicates with backend via RESTful APIs
- **Backend:**
  - Developed using FastAPI (or Django/Flask)
  - Manages business logic, data validation, authentication, and API endpoints
  - Role-based access control (Teacher, Student)
- **Database:**
  - Uses SQLite for development/testing (can switch to PostgreSQL in production)
  - Stores Users, Assignments, and Submissions in relational tables
- **Authentication & Authorization:**
  - Uses JWT (JSON Web Token) for secure, stateless authentication
  - Access permissions enforced based on user roles (Teacher/Student)

2. Core Entities and Relationships

Entity	Attributes	Relationships
User	id (PK), name, email, password, role (student/teacher)	Teacher or Student
Assignment	id (PK), title, description, created_by (FK: User), created_at	Created by Teacher
Submission	id (PK), assignment_id (FK), student_id (FK), content/file, submitted_at	Submitted by Student for an Assignment

3. API Endpoints

Method	Endpoint	Access	Description
POST	/signup	Open	Register a new user (teacher/student)
POST	/login	Open	Login and receive JWT token
POST	/assignments	Teacher	Create a new assignment
POST	/assignments/{id}/submit	Student	Submit assignment for given ID

GET	/assignments/{id}/submissions	Teacher	View all submissions for a specific assignment

4. Authentication Strategy

- **JWT Authentication:**
  - On successful login, a JWT token is issued containing user role
  - Each protected API validates the token for authenticity and checks role-based access
  - Passwords securely stored (hashed) in the database
- **Role Permissions:**
  - Teacher: Create assignments, view submissions
  - Student: Submit assignments

5. Scaling Strategies

- Database Scaling: Use PostgreSQL with indexing for better query performance
- Containerization: Deploy backend using Docker for consistency across environments
- Load Balancing: Use a load balancer with backend replicas for high traffic
- Caching: Implement Redis for frequently accessed data like submissions count
- Microservices: Split core functionalities (auth, assignment, submission) into independent services as the system grows
- Cloud Storage: Integrate with AWS S3 for file uploads
- Monitoring: Add monitoring tools like Prometheus and Grafana for system health checks