**The exercise described does not strictly require production-ready code, but it does expect a very high standard of code quality. You should aim for a well-engineered prototype that’s maintainable, testable, and deployable — but not necessarily a full-scale production system.**

**We are not looking for a POC that somehow works and meets the requirement but a well-designed and organized system (even if it is not feature complete).  
  
Important Notes:  
• Don’t worry if some features are incomplete – we care more about how well you’ve structured your app, even if not everything works perfectly.  
• Focus on Quality: If something doesn’t work fully, don’t leave it unfinished or hacky. Show us that you can design an elegant solution that is easy to maintain and extend.  
• Keep it Modular: Even small features can be modular and easy to extend.  
• Performance: Think about how this system would behave with large data sets**

**MANDATORY INSTRUCTIONS**

* **Code should be modular and structured (not all files should be in one folder, should reusable and modules can be tested independently)**
* **Modules and Packages should be distinct**
* **Authentication should be implemented**
* **Unit test cases for all modules/components are mandatory**
* **Detailed documentation step by step procedure on how to run your application is also mandatory**

**NestJS backend application for user and document management**

1. **NestJS Backend (User Management and Document Management)**
   * **Purpose:** Create a backend service using NestJS to manage user authentication, document management, and ingestion controls.
   * **Key APIs:**
     + **Authentication APIs**: Register, login, logout, and handle user roles (admin, editor, viewer).
     + **User Management APIs**: Admin-only functionality for managing user roles and permissions.
     + **Document Management APIs**: CRUD operations for documents, including the ability to upload documents.
     + **Ingestion Trigger API**: Allows triggering the ingestion process in the backend, possibly via a webhook or API call.
     + **Ingestion Management API**: Tracks and manages ongoing ingestion processes.
   * **Tools/Libraries**:
     + TypeScript for consistent type management.
     + Database integration (Postgres recommended).
     + JWT for authentication, with role-based authorization.
     + Microservices architecture to facilitate interaction between NestJS and the Python backend.

**Evaluation Criteria**

1. **Code Quality and Structure**:
   * Your code should be modular: avoid placing everything in a single file or folder.
   * Use packages and reusable components so parts of the system (e.g., models, routes, services) can be independently tested or updated.  
      Implement basic authentication, error handling, and unit tests for all core modules and endpoints.
   * TypeScript usage with strong object-oriented principles.
   * Clean, well-documented, and easy-to-understand code structure.
2. **Data Modeling and Design**:
   * Design a robust database schema, including generating a large dataset (e.g., 1000+ users with roles, 100000+ entities).
   * Demonstrate methods to create realistic test data.
3. **API Development and Testing**:
   * REST API design and automated testing.
   * Microservices architecture integration to handle the Python backend for ingestion.
4. **Authentication and Authorization**:
   * Implementation of JWT-based authentication with role-based access control.
   * Demonstration of secure and scalable authentication for high volumes of users.
5. **Additional Skills**:
   * Knowledge of microservices and inter-service communication.
   * Problem-solving skills and scalability considerations for handling large datasets and user traffic.

**End-of-Development Showcase Requirements**

At the end of the development, candidates should demonstrate the following:

1. **Design Clarity**:
   * Show a clear design of classes, APIs, and databases, explaining the rationale behind each design decision.
   * Discuss non-functional aspects, such as API performance, database integrity, and consistency.
2. **Test Automation**:
   * Showcase functional and performance testing.
   * Cover positive and negative workflows with good test coverage (70% or higher).
3. **Documentation**:
   * Provide well-documented code and create comprehensive design documentation.
4. **3rd Party Code Understanding**:
   * Explain the internals of any 3rd-party code used (e.g., libraries for LLM or authentication).
5. **Technical Knowledge**:
   * Demonstrate knowledge of HTTP/HTTPS, security, authentication, authorization, debugging, monitoring, and logging.
6. **Advanced Concepts**:
   * Showcase advanced concepts like RxJS, NgRx, and ORM where applicable.
   * Usage of design patterns in code.
7. **Test Data Generation**:
   * Demonstrate skills in generating large amounts of test data to simulate real-world scenarios.
8. **Deployment and CI/CD** (Applicable to All Components):
   * **Dockerization**: Dockerize each service, making it easily deployable and portable.
   * **Deployment Scripts**: Provide deployment scripts to run the application on Docker or Kubernetes, compatible with any cloud provider (e.g., AWS, Azure, GCP).
   * **CI/CD Pipeline**: Implement a CI/CD pipeline for each component to automate testing, building, and deployment.

You do not need to set up a full AWS production environment. Instead, pick one of the following deployment methods to show you're able to get the app running in a cloud-like setup:

You can choose any one of these:  
- Docker-based deployment: Write a Docker file and Docker Compose (if needed). This is acceptable as a "cloud-ready" setup.  
- GitHub Repo with README: Push your code to GitHub and include a detailed README explaining how to run the app locally or deploy it.  
- CI/CD or Infrastructure Plan: Describe, in a clear and reproducible way, how the project would be deployed in the cloud (e.g., using AWS EC2, ECS, or RDS).  
- Optional AWS Deployment: If you're comfortable, set up an AWS Free Tier environment with EC2 + RDS. Not mandatory.

The goal is to see that your code is structured to be deployed, even if it’s not fully live in the cloud.