Name: Vidya Sinha , Dhruvi Patel

Project title : The Accreditation and Data Management Assistant

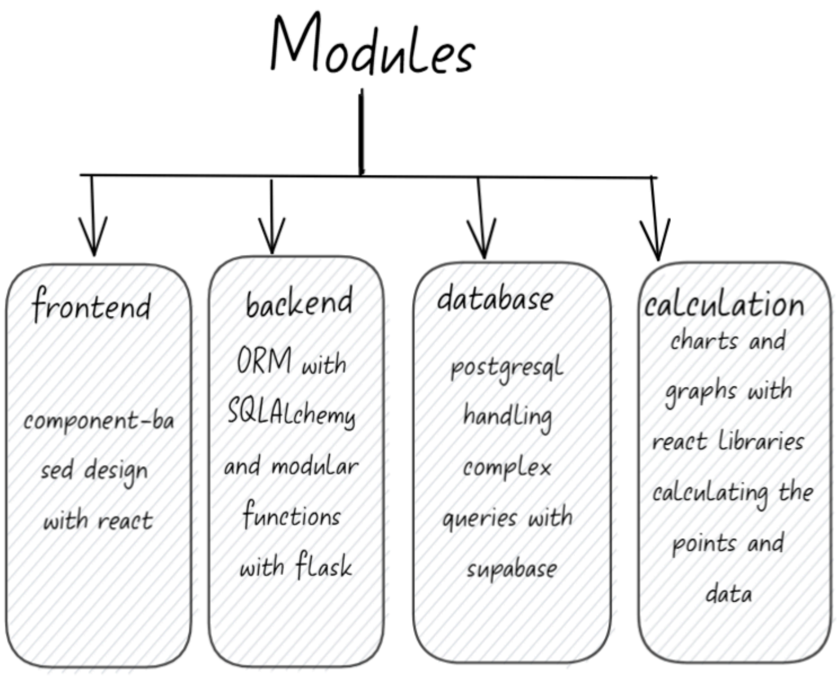
System Design and Architecture

Introduction

The Accreditation and Data Management Assistant is designed to automate the entire process of collecting, validating, and computing scores for higher-education accreditation. It centralizes institutional data (faculty, students, research, infrastructure), calculates accreditation scores based on predefined criteria, and provides real-time dashboards for administrators and accreditation bodies.

The system is designed to be modular, scalable, secure, and maintainable, ensuring flexibility to accommodate multiple accreditation frameworks and future expansion. Now lets look into the details how we are achieving this :-

The whole system is divided into 4 modules :


Modules
frontend
component-based design with react
backend
ORM with SQLAlchemy and modular functions with flask
database
postgresql handling complex queries with supabase
calculation
charts and graphs with react libraries calculating the points and data

So the major role of frontend will have :

1. User interface for administrators, HoDs, deans, and accreditation officers.
2. Dashboard (score visualization, compliance status) and forms for data entry, file upload in different formats and and reporting export PDFs, CSVs for SAR.
3. Technology used are React with TypeScript, Tailwind CSS (UI design) as React ensures component-based design for reusability. Tailwind CSS makes responsive UI design.

Backend have responsibilities like ;

1. Core logic, business rules, and A LOT of API services.
2. For API services we have RESTful APIs for every micro service.
3. For Authentication & Authorization we used JWT-based role management managing the security which ensures secure and stateless authentication with Flask,
4. We have used SQLAlchemy ORM which simplifies database interactions and supports complex queries and its easy to shift the whole thing to any technology easily.

Database we have managed on Supabase because we needed:

1. Secure storage of all institutional, student, faculty, and accreditation-related data.
2. database tables have Users, Institutions, Departments, Courses, Faculty, Students, Publications, Placements and 10 more tables according to criterias.
3. Relationships we have managed with proper normalization for avoiding redundancy as the same data was being used in many different criterias so we couldnt have made table with respective of criterias of bodies.
4. For file storage of documents and evidence we have used supabase storage services. Made different buckets like student and faculties to manage the storage of files separately.
5. Supabase provides secure, cloud-based hosting, automatic backups, and file storage.

Now as this system as several calculations we needed

1. Score dashboards with charts and graph of yearly updated documents and details.
2. Calculation for qualitative data as well as quantitative data with unique formula given for every sub criterias.

List of functional and non functional requirements:
FUCNTIONAL:

1. Enter course outcomes, research publications, and other academic achievements.
2. Validate student performance data.
3. Access dashboards to view departmental statistics.
4. Manage student admissions and enrollment data.
5. Access verified institutional data.

6. Monitor system usage and dashboard analytics.
7. View personal academic records and performance metrics.
8. Upload required documents (10th, 12th, registration, entrance exams).

NON FUNCTIONAL:

1. Performance wise: System should handle 1000+ students/year without noticeable lag; heavy operations (file upload, bulk retrieval) optimized.
2. Scalabilty wise : it should be able to scale horizontally (multiple servers) and vertically (more CPU/memory) as data grows.
3. Intuitive UI/UX for HoD and deans, and admin users; dashboards with clear visualizations.
4. JWT authentication, encryption for sensitive data, role-based access control, input sanitization, file validation.

Thes some of the APIs that we are calling are:
GET: /criteria4/admissions, /criteria4/academic performance-index, /criteria4/events, /criteria5/faculty-cadre etc
POST: criteria4/student/admissions/upload-docs, criteria4/student-upload
PUT: criteria4/update-facultydetails

For Security & Rate Limiting:

1. Use JWT for authentication across all endpoints.
2. Sensitive operations like file uploads use rate limiting (eg. 10req/min) to prevent abuse.
3. Input validation and sanitization are applied on all endpoints.

ESTIMATIONS:

Assuming 1000 students per year:

In File Storage

Each student has ~4 documents (10th, 12th, GUJCET, registration)
Average file size = 2 MB
Annual storage growth: 1000 students × 4 × 2 MB = 8 GB/year

To ensure the system can efficiently handle increased user load and data volume, several scalability strategies have been implemented. Load balancing is achieved by deploying the Flask backend behind an Nginx server, which distributes incoming requests across multiple application servers. This setup enables the system to manage concurrent requests effectively while maintaining responsiveness. Additionally, session management is handled using Redis, allowing user sessions to be shared seamlessly across servers and improving overall reliability during peak usage.

Database scalability is addressed through indexing of frequently queried fields, which significantly improves query performance for large datasets. Read replicas are used to offload high-volume read operations, such as dashboard queries and report generation, ensuring that the primary database remains responsive. For future growth, sharding can be implemented to partition large datasets across multiple database instances, further enhancing performance and maintaining low latency.

API performance optimization includes implementing rate limiting on heavy endpoints, such as file uploads (`/upload-documents`) and bulk performance data retrieval (`/academic-performance`), to prevent system overload. Frequently accessed queries are cached using Redis, reducing database access times and improving response speed. Furthermore, asynchronous processing is used for file uploads via background workers (e.g., Celery or Flask-RQ), which prevents long-running tasks from blocking API responses and ensures a smooth user experience.

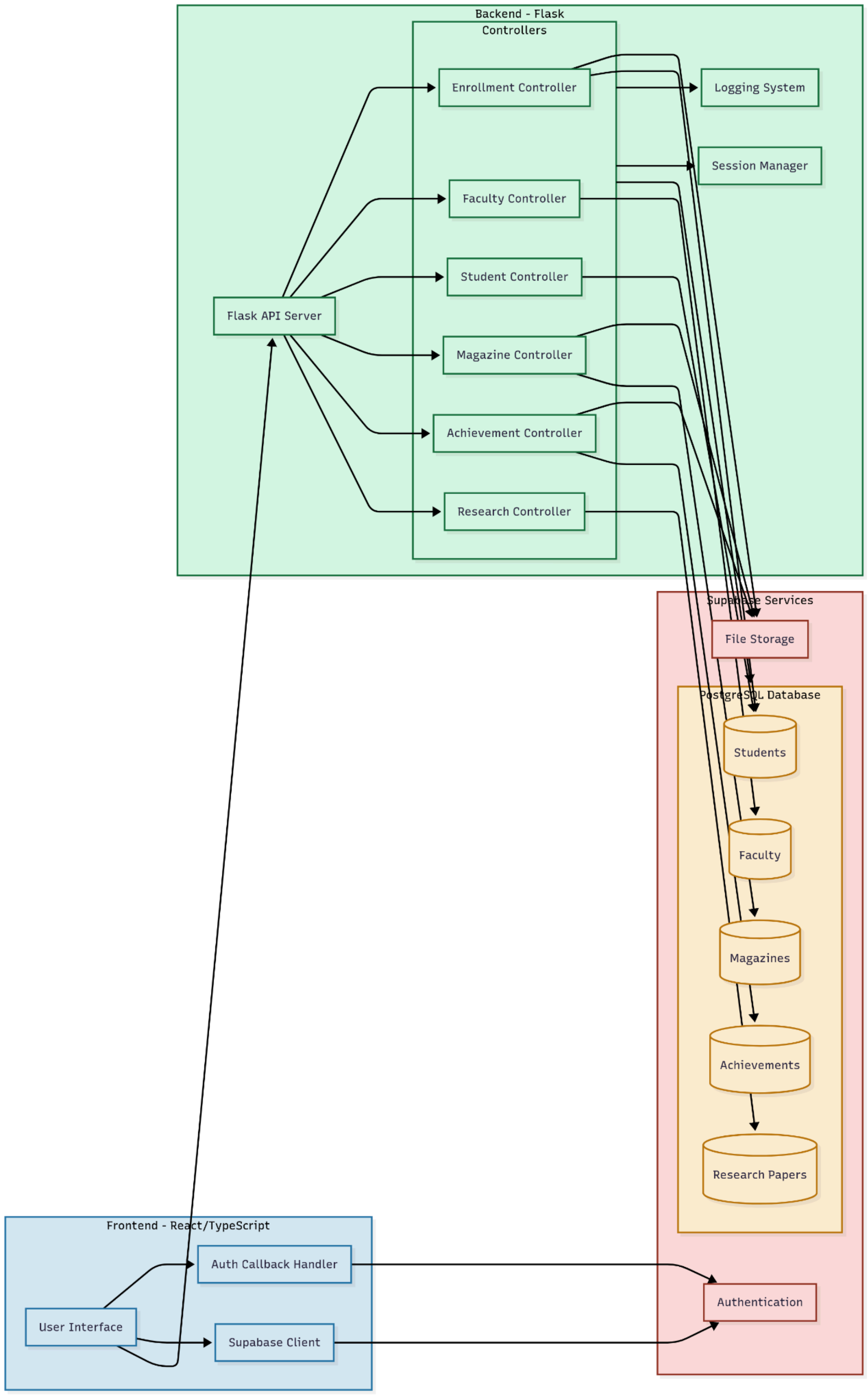
Now lets look into scalability , on how we can scale if get more and more data over the years:

Since we are having limited users we dont have to take accountability for user load but we have to consider the increasing data.

A. For horizontal scaling RESTful APIs are stateless, allowing horizontal scaling by deploying multiple backend instances behind a load balancer.
B. Caching frequently accessed data (Redis / in-memory caching) to reduce database queries.
C. Read replicas for high-volume read operations.
D. Partitioning / Sharding if tables grow very large (e.g., multi-year institutional data).

E. Indexes on frequently queried fields for faster access
F. Lazy loading of components and data fetching optimizes performance for larger institutions.

Here is the diagram to demonstrate it:



The proposed architecture establishes a clear and scalable flow between the Frontend, Backend, and Supabase services, ensuring seamless integration of user interactions, data processing, and secure storage. By aligning the system left-to-right, we achieve a cleaner separation of concerns: the React/TypeScript frontend handles user interactions and authentication, the Flask backend manages business logic through dedicated controllers, and Supabase services provide reliable data storage, file management, and authentication. Additional components like the logging system and session manager further enhance maintainability, security, and monitoring. Overall, this architecture not only minimizes complexity and arrow collisions in the design but also demonstrates a modular, transparent, and efficient approach well-suited for higher education accreditation and evaluation systems.