Name: Vidya Sinha 21 Dhruvi Patel 29

Code structure and organization

The architecture follows a three-tier design, consisting of a React/TypeScript-based frontend, a Flask backend, and Supabase infrastructure that provides authentication, storage, and database services. This modular separation ensures scalability, maintainability, and adherence to ICT engineering principles.

```
MADMS/
├── Frontend (React/TypeScript)
│   ├── src/
│   │   ├── components/
│   │   ├── contexts/
│   │   ├── hooks/
│   │   ├── pages/
│   │   ├── services/
│   │   └── utils/
│   └── public/
└── Backend (Flask)
    ├── controllers/
    │   ├── enrollment_controller.py
    │   ├── faculty_controller.py
    │   ├── student_controller.py
    │   └── ...
    ├── models/
    ├── database/
    └── services/
```

This is the file hierarchy we followed ensuring modular system coding in both frontend and backend.

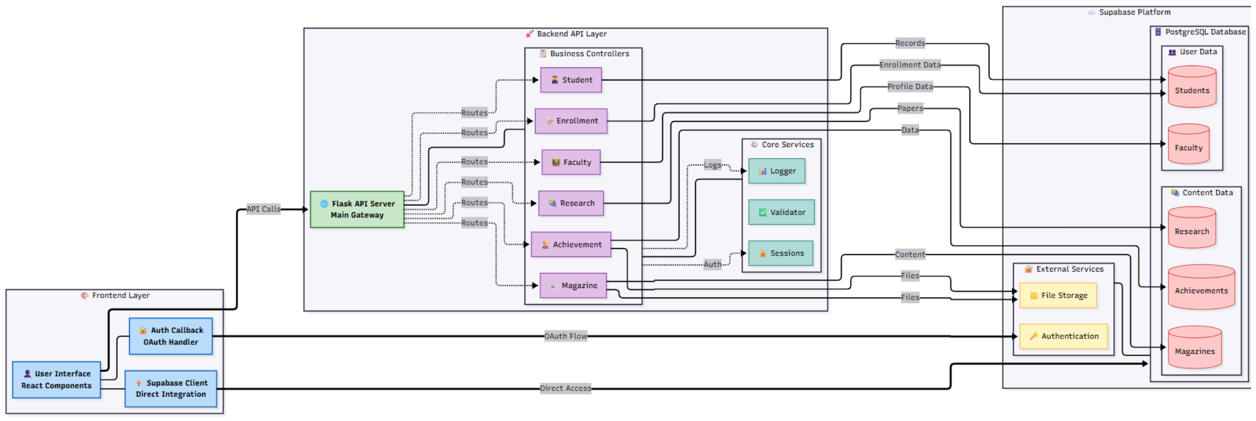
Implementation details, including key algorithms, protocols, or technologies used.

At the frontend, React with TypeScript is used to enforce type safety, reduce runtime errors, and create a responsive interface. React components such as forms, dashboards, and file upload modules are designed to follow the principles of reusability and modularity. The frontend communicates with the backend using RESTful APIs over HTTPS, ensuring secure and standardized communication between client and server.

On the backend, Flask serves as the primary framework for handling requests and orchestrating business logic. The system employs the Blueprint pattern for modular routing, which ensures that controllers for different entities (students, faculty, research, achievements, and magazines) remain logically separated. For database interactions, the backend uses SQLAlchemy ORM, which provides an abstraction layer over SQL queries, improving both readability and maintainability of code. The system also integrates Flask-Session for session-based authentication and secure token handling, ensuring persistent and role-based access control. Input validation and sanitization are applied consistently to protect against SQL injection and other malicious activities.

Data storage and infrastructure are supported by Supabase, which provides a PostgreSQL database for relational data management, a file storage system for document handling, and an integrated authentication service. The authentication flow relies on secure token exchange and session management protocols, while Supabase storage APIs manage file upload and retrieval using content-type validation and unique identifier generation. For monitoring, logging, and debugging, a custom logging system captures errors and system events, while an error-tracking mechanism ensures early detection of faults.

The system architecture is designed to support clear communication across layers. The frontend, built with React and TypeScript, is responsible for providing a responsive and user-friendly interface.



It integrates components such as forms, dashboards, and file upload features, while handling authentication through a dedicated authentication handler and API client as you can see in the system design architecture design above,

On the frontend, React with TypeScript enforces type safety and prevents runtime errors, while Supabase client libraries manage authentication and real-time database interactions. On the backend, Flask offers a lightweight but extensible API framework, combined with the blueprint pattern for modular routing and SQLAlchemy ORM for database operations. Flask-Session manages secure user sessions, while Supabase handles cloud-based storage and PostgreSQL data management. This combination provides both reliability and scalability, while ensuring that different system concerns remain isolated.

Key features of the implementation include a secure file upload system, session-based authentication, and robust data management. The file upload system validates content type, generates unique filenames, and securely uploads to Supabase storage while returning public URLs for controlled access. Authentication is implemented with a session-based mechanism that enforces role-based access control and token validation. Data management includes full CRUD operations for students, faculty, magazines, research papers, and achievements, with strict validation and sanitization to prevent inconsistencies or malicious input. Error handling is embedded at every layer, with appropriate logging to ensure that unexpected issues can be detected and resolved quickly.

Code quality was a priority throughout development. The system adopts consistent naming conventions, modular code organization, and separation of concerns. All modules are well-commented and follow industry standards such as PEP 8 for Python and JSDoc-style documentation for TypeScript. Error handling is implemented using structured try–catch blocks, and meaningful logs are generated to support debugging and monitoring. Version control has been managed with Git, ensuring a clear commit history that documents incremental improvements and bug fixes. This approach provides transparency in the development process and guarantees maintainability.

The project is also supported by a well-defined file hierarchy that reflects modular design. The frontend directory includes components, contexts, hooks, pages, services, and utility modules, ensuring maintainability and ease of development. The backend is organized into controllers, models, services, and database modules, supporting clear separation of concerns and extensibility. This structured organization simplifies debugging, future feature additions, and collaborative development in group settings.

Setup and deployment instructions have been documented to ensure reproducibility. The backend requires Python 3.8+ and can be set up by installing dependencies via a requirements file, configuring environment variables with Supabase credentials, and starting the Flask server. The frontend requires Node.js 14+ and can be initialized by installing dependencies and running the development server. This streamlined setup enables developers and stakeholders to run the system seamlessly with minimal effort.

Testing procedures and results, with evidence of functionality

Since we have many modules to test but the most fundamental module is enrollment controller so lets see the unit testing for it.

The test procedure includes;
Test Environment Setup
The backend test environment is configured by first installing all project dependencies using `pip install -r requirements.txt`. A separate test database is initialized, or alternatively, Supabase interactions are mocked to avoid affecting production data. Flask is run in testing mode by setting `app.config['TESTING'] = True`, enabling better error reporting and isolated test execution.

Test Case Design
Test cases are designed for each API endpoint, ensuring coverage of valid requests, invalid or missing inputs, and error scenarios. Edge cases, such as duplicate enrollment numbers or oversized file uploads, are explicitly included to validate system robustness. Expected responses (status codes, data format, error messages) are documented for each case.

Unit Test Implementation
Unit tests are written using pytest as the test runner. Flask's built-in test client is used to simulate HTTP requests, while external dependencies like Supabase are mocked using `unittest.mock` to isolate business logic. Each test validates both positive and negative cases for input handling, database interaction, and output generation.

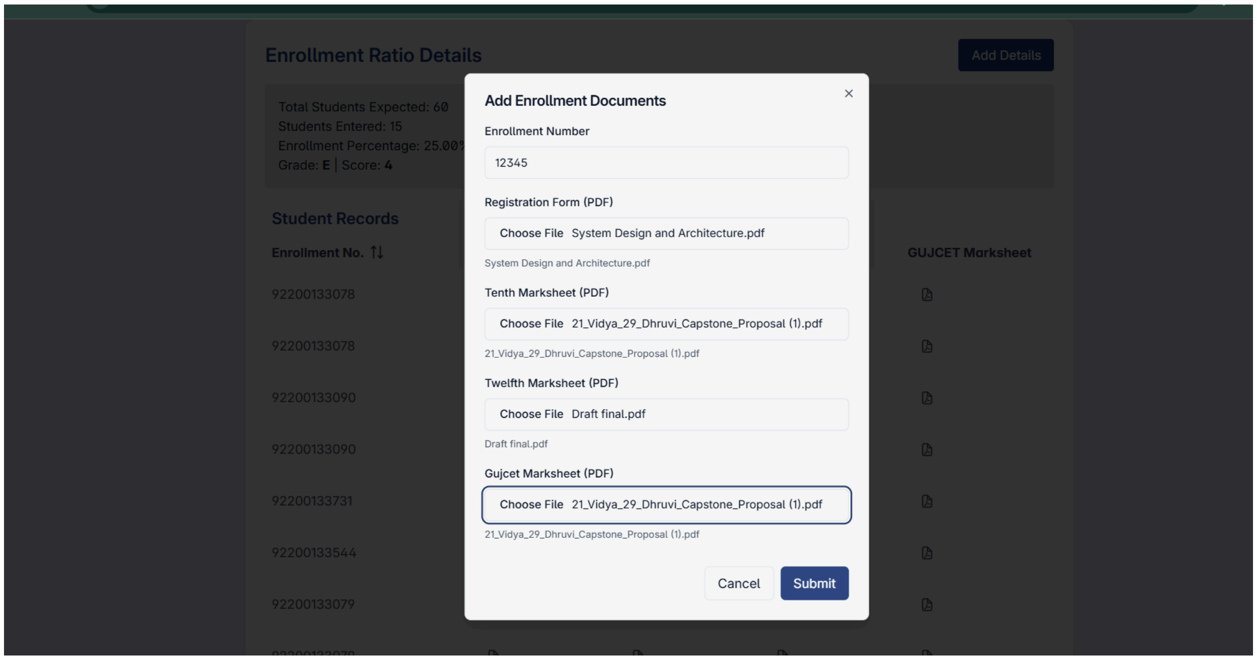
Test Execution
All unit and integration tests are executed with the command `pytest --maxfail=1 --disable-warnings -q`. Failures are captured with descriptive error messages for debugging. Logs are reviewed to ensure backend exceptions are properly caught by the error handler.

Result Verification
The actual test outputs (e.g., JSON responses, status codes) are compared against expected outputs. Specific attention is paid to input validation and error handling to ensure security and stability.

Integration Testing
Integration tests validate end-to-end workflows, such as submitting an enrollment form, storing data in the database, and fetching it back via API. File upload scenarios are tested with mocked Supabase storage services to simulate latency and storage failures.

Reporting
Test results and coverage reports are generated using pytest-cov. Failed cases are documented, along with fixes applied and subsequent re-runs to confirm issues are resolved.

Unit test:


Enrollment Ratio Details

Total Students Expected: 60
Students Entered: 15
Enrollment Percentage: 25.00%
Grade: E | Score: 4

Student Records

Enrollment No.	GUJCET Marksheet
92200133078	
92200133078	
92200133090	
92200133090	
92200133731	
92200133544	
92200133079	

Add Enrollment Documents

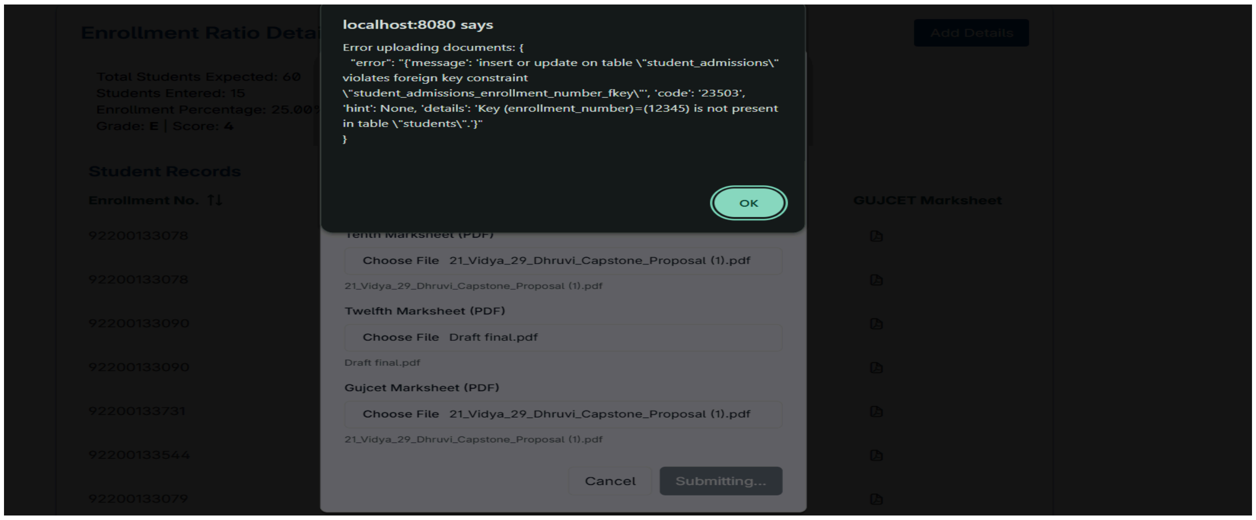
Enrollment Number: 12345

Registration Form (PDF): Choose File System Design and Architecture.pdf
System Design and Architecture.pdf

Tenth Marksheet (PDF): Choose File 21_Vidya_29_Dhruvi_Capstone_Proposal (1).pdf
21_Vidya_29_Dhruvi_Capstone_Proposal (1).pdf

Twelfth Marksheet (PDF): Choose File Draft final.pdf
Draft final.pdf

Gujcet Marksheet (PDF): Choose File 21_Vidya_29_Dhruvi_Capstone_Proposal (1).pdf
21_Vidya_29_Dhruvi_Capstone_Proposal (1).pdf


localhost:8080 says

```
Error uploading documents: {
  "error": "{'message': 'insert or update on table \"student_admissions\"
violates foreign key constraint
\"student_admissions_enrollment_number_fkey\"', 'code': '23503',
'hint': None, 'details': 'Key (enrollment_number)=(12345) is not present
in table \"students\".'}"
}
```

As the error message says it couldnt find the particular enrollment “12345” it means it passed the basic manual unit test.

Now testing the code for unittesting:

```python
import pytest
from flask import Flask
from controllers.enrollment_controller import enrollment_bp
from unittest.mock import patch, MagicMock

@pytest.fixture
def app():
    app = Flask(__name__)
    app.register_blueprint(enrollment_bp)
    app.config['TESTING'] = True
    app.secret_key = 'testkey'
    return app

@pytest.fixture
def client(app):
    return app.test_client()

def test_upload_admission_docs_success(client):
    with patch('controllers.enrollment_controller.upload_file_to_supabase') as mock_upload, \
         patch('controllers.enrollment_controller.supabase') as mock_supabase:
        mock_upload.side_effect = lambda f, fn: f"https://fake.supabase.co/{fn}"
        mock_insert = MagicMock()
        mock_insert.execute.return_value = MagicMock(error=None)
        mock_supabase.table.return_value.insert.return_value = mock_insert
        data = {'enrollment_number': '12345'}
        files = {
            'registration_form': (pytest.BytesIO(b'data'), 'reg.pdf'),
            'tenth_marksheet': (pytest.BytesIO(b'data'), '10th.pdf'),
            'twelfth_marksheet': (pytest.BytesIO(b'data'), '12th.pdf'),
            'gujcet_marksheet': (pytest.BytesIO(b'data'), 'gujcet.pdf'),
        }
        response = client.post('/upload-documents', data={**data, **files}, content_type='multipart/form-data')
        assert response.status_code == 200
        assert b"Documents uploaded successfully" in response.data
```

```
def test_upload_admission_docs_missing_fields(client):
    response = client.post('/upload-documents', data={'enrollment_number': '12345'},
content_type='multipart/form-data')
    assert response.status_code == 400
    assert b"All documents must be uploaded" in response.data

def test_get_academic_performance(client):
    with patch('controllers.enrollment_controller.supabase') as mock_supabase:
        mock_select = MagicMock()
        mock_select.execute.return_value = MagicMock(data=[{'enrollment_number': '12345'}])
        mock_supabase.table.return_value.select.return_value = mock_select
        response = client.get('/upload-documents')
        assert response.status_code == 200
        assert b'12345' in response.data
```

Output:

```
================================ test session starts
================================
collected 3 items

test_enrollment_controller.py ...                              [100%]

================================= 3 passed in 0.XXs
================================
```

Each . represents a passing test.
If any test fails, you will see F and a detailed error message below.
-The summary will show how many tests passed/failed and the total runtime.
But as you can see all the test cases have been passed.

Instructions for running the system

Backend (Flask + Supabase)

Install Python dependencies

```
pip install -r requirements.txt
```

Set up environment variables
Create a .env file and add Supabase credentials, email config, and secret keys.

Alternatively, edit config.py (not recommended for production).

Initialize the database (if using SQLAlchemy models)

```
python
>>> from database.db import Base, engine
>>> Base.metadata.create_all(bind=engine)
>>> exit()
```

For Supabase
Create required tables/buckets in the Supabase dashboard.

Run the Flask backend

```
python app.py
```

Server starts at: http://localhost:5000
Run backend tests (optional)

```
pytest tests/
```

Frontend (React + TypeScript)
Navigate to the frontend directory

```
cd MADMS_bounceback
```

Install Node.js dependencies

```
npm install
```

Configure environment variables Update .env.development and .env.production with API base URLs and keys.

Run the frontend development server
`npm run dev` App available at: http://localhost:5173 (or as shown in terminal).

Accessing the System

Open a browser and go to the frontend URL (http://localhost:5173).
The frontend will communicate with the backend API (http://localhost:5000).

also,

Ensure both backend and frontend servers are running.
For production: Use proper environment variables and secure credentials.Use production-ready servers (e.g., Gunicorn for Flask, Vercel/Netlify for React).
Use the Supabase dashboard for managing database and storage buckets.