1. **Cloud Architecture & Network Design**

Amazon Web Services has been selected as our cloud service provider based on a few factors in a comprehensive evaluation of cost efficiency, service breadth, scalability capabilities, and security features. AWS allows our team to pay only for resources that are consumed without upfront capital investments. The platform's auto-scaling and elastic load balancing features ensure our application can adjust to carrying traffic loads, providing optimal performance during slower periods as it scales down to control costs.

Another major factor in our decision is that the breadth and maturity of AWS services make it an ideal choice for our comprehensive application architecture. With over 200 featured services such as storage, networking, databases, analytics, and security, AWS provides all the necessary components we need to design, deploy, and manage our infrastructure. With security being a top priority, AWS maintains enterprise-grade security standards with compliance certifications including SOC 2, ISO 27001, and GDPR, ensuring we maintain enterprise-level security and meet regulatory requirements. The platform's global infrastructure with a 99.99% uptime SLA guarantees high availability and reliability for our users.

Selected AWS Services
EC2 for hosting the Node.js application

- S3 for object storage and backups
- RDS for managed MySQL database
- VPC for isolated networking
- IAM for access management
- ELB for load balancing and availability

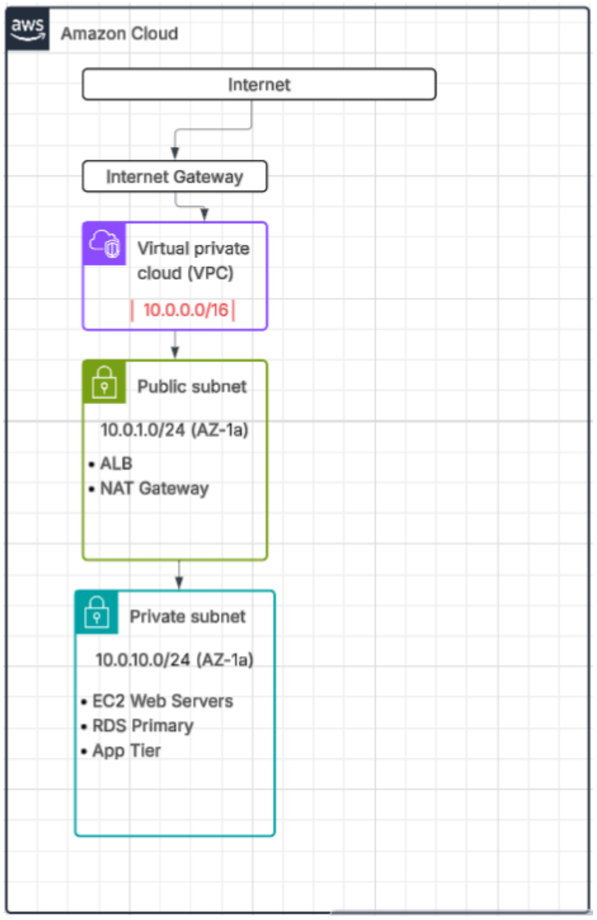
Network Architecture

We'll establish a Virtual Private Cloud (VPC) with redundant subnets distributed across multiple Availability Zones to maintain high availability. This architecture isolates public-facing components from internal systems, enforcing strong security boundaries while maintaining necessary connectivity.

Key components:

- 1 public subnet (for application servers/load balancers in separate AZs)
- 1 private subnet (for the database tier in separate AZs)
- **Internet Gateway** attached to the VPC to enable inbound/outbound traffic for public subnets.
- **Route Tables** are configured to manage traffic flows, ensuring private subnets route outbound traffic via the NAT Gateway while public subnets have direct Internet Gateway access.
- **Security Groups** with strict ingress/egress rules to enforce a least-privilege model.

Security Group Rules:
- HTTP (80) and HTTPS (443): Open to the world for web access.
- MySQL (3306): Only accessible from application server security groups (not public).
- ICMP (ping): Restricted to specific admin IP ranges for diagnostics.
- RDP (3389) or SSH (22): Locked to specific admin IPs for secure management access



Port Configuration:

- 80 / 443 (Web): Open for public HTTP and HTTPS traffic to the application server.
- 22 (SSH) / 3389 (RDP): Limited to administrative IP ranges for secure server management.
- 3306 (MySQL): Accessible only from the application server within the VPC.
- ICMP: Restricted to specific administrative IPs for troubleshooting connectivity

2. Application Design

The application is designed using a modular, scalable, and cloud-ready architecture that separates concerns across three primary layers: the frontend, backend, and database. Each layer leverages modern, widely supported technologies that ensure ease of development, maintainability, and seamless deployment within the AWS cloud ecosystem.

Frontend Layer

The frontend is built using React.js, a popular JavaScript library for constructing dynamic user interfaces. React's component-based architecture is ideal for building reusable UI components, such as course cards, search bars, and registration forms. React's use of the Virtual DOM ensures efficient rendering and responsiveness, particularly important for mobile and web-based access.
To manage state and data flow, React integrates easily with libraries such as Redux or Context API. HTTP requests to the backend are handled via Axios or native fetch() calls. For styling and layout, we plan to utilize frameworks such as React Bootstrap or Material UI, depending on team preference and UI design consistency.

Backend Layer

The backend is powered by Node.js, a fast and efficient JavaScript runtime environment that handles asynchronous operations well, making it suitable for high-traffic applications such as course registration systems. The backend exposes a RESTful API using the Express.js framework, which provides routing, middleware integration, and error handling. By default, the Node/Express server will run on port 3000 during development. The React frontend, typically hosted on port 5173 (when using Vite) or 3000 (for Create React App), communicates with the backend through this port. In production, reverse proxies or load balancers (e.g., via NGINX or AWS Application Load Balancer) will handle routing and port forwarding as necessary.
Endpoints will be created for user login, course search, student registration, and administrative functions. JSON will be used as the standard data format for all API responses and requests, ensuring clean data exchange with the React frontend.

Middleware and Supporting Packages

Several middleware tools will be used to support backend functionality:

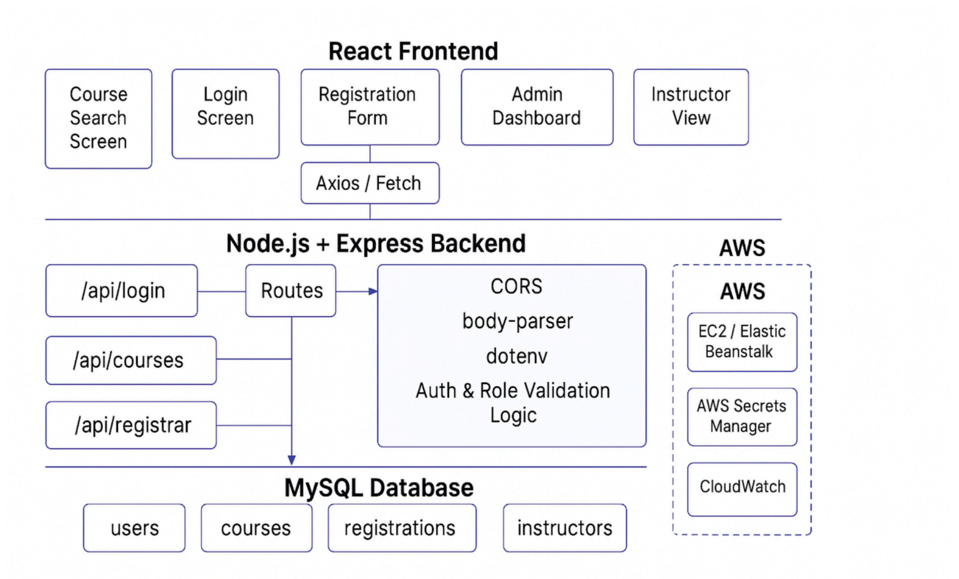
- cors: Enables cross-origin resource sharing to allow frontend-backend communication.
- body-parser: Parses incoming request bodies for POST and PUT requests.
- dotenv: Loads environment variables securely from a .env file for local development and AWS Secrets Manager for production.

Package dependencies will be managed using npm (Node Package Manager), streamlining the process of adding, updating, and maintaining open-source modules.

Runtime and Hosting Environment

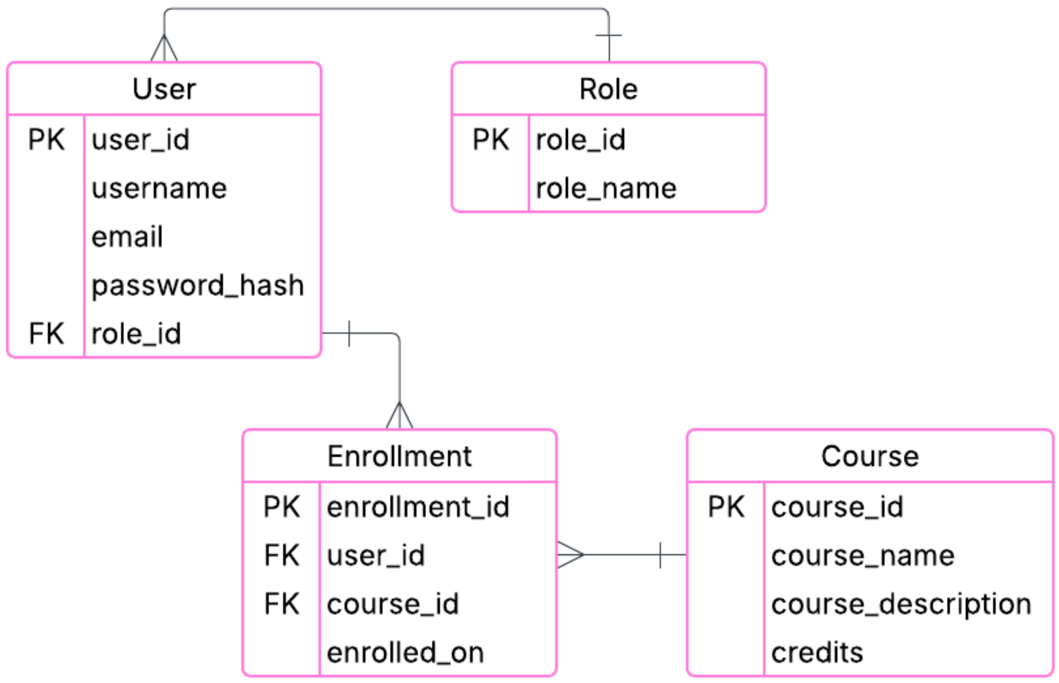
The application will run in an AWS-hosted environment, using either EC2 instances or Elastic Beanstalk for simplified orchestration. Application logs will be centralized for monitoring via CloudWatch, and autoscaling can be enabled based on CPU or memory usage for production environments. Security will be enforced through API token management and user role segmentation (student, admin, instructor). Application secrets and credentials will be managed through AWS Secrets Manager or .env files (during development).

Figure 1: Component Layer Architecture for Online Course Registration System



3. Database Design & Data Visualization Tool

ERD:



For our student course registration system application, we will utilize MySQL, hosted via Amazon RDS. MySQL is a suitable option for our project due to its structured data, which includes courses, enrollments, and users. RDS also automates backups, scaling, and maintenance. Lastly, it integrates seamlessly with Node.js. For a data visualization tool, we chose to use Power BI. Power BI provides interactive dashboards, scheduled refreshes, and analytics for enrollment and course trends. Concerning AI and data integration, it offers AI-powered Q&A visualizations, natural language queries, and direct MySQL connectors. My team members are familiar with Power BI from other courses. Additionally, Power BI's interface is intuitive with drag-and-drop functionality and prebuilt templates. Power BI Desktop is free, which is ideal for projects with tight timelines and budget constraints. Unlike Tableau, Power BI does not have a trial period limitation, enabling continuous access throughout the project.

4. Testing and Quality Assurance Process

To ensure the course registration system operates reliably and meets functional expectations, our team will adopt a multi-layered testing strategy that aligns with Agile development practices. This includes unit testing, integration testing, and end-to-end (E2E) testing. Each testing method is aligned with a specific sprint phase to validate production code from both a technical and user-centric perspective.

Unit Testing: To validate individual components or modules of the application, we will use unit testing. This involves testing specific pieces of the code, such as buttons, forms, or backend functions, in isolation to ensure they work correctly on their own. We will use Mocha and Chai for backend testing and React Testing Library for frontend components. By testing each unit separately during Sprint 2, we can catch and fix minor bugs early, which helps keep the overall system stable and easier to maintain as we build out the rest of the application.

Integration Testing: Our approach to integration testing focuses on ensuring that different modules of the system, such as the front end, back end, and database, communicate and function properly when connected. We will utilize tools such as Postman for manual API testing and Jest for automated integration testing. For example, we will test whether data entered in the registration form on the frontend is successfully processed by the backend and correctly stored in the database. This testing will take place during Sprint 3 and is essential for confirming that the system's components work seamlessly as a whole.

End-to-end (E2E) Testing: Our end-to-end testing strategy is designed to simulate fundamental user interactions and ensure that the entire application functions correctly from start to finish. Using tools like Cypress or Selenium, we will test complete user workflows such as a student searching for a course, registering, and receiving a confirmation, or an administrator accessing session data. These tests will be conducted during Sprint 4 and will help us identify any issues in navigation, data flow, or user experience. This final layer of testing ensures that all components work together smoothly and that the system provides a reliable and intuitive experience for its users.

Authentication and Authorization Process

For this project, the application is designed to be accessible to all users without requiring authentication. This decision aligns with the assignment's scope, which specifies that authentication and authorization processes are not necessary at this stage. However, throughout the project, our team has prioritized laying the foundation for realistic and secure expansion in future phases.

We have already defined two core user roles: Student and Administrator, within our system design. These roles are reflected in the structure of different parts of the application. For example, the React front end includes features tailored to each role, such as course search and enrollment for students and session or student management views for administrators. Even though login is not currently enforced, this role-based design ensures that the application can be extended later to support absolute access control.

In the backend, we use Node.js and Express, which support scalable middleware and security integration. We've also integrated port restrictions and security group rules into our AWS network configuration to ensure that only essential ports (such as HTTP, HTTPS, and MySQL) are open and that administrative access (like SSH) is limited to specific IP addresses. These choices reflect best practices in cloud security and mirror what would be done in a production environment.

Although authentication methods like token-based login, LDAP are not implemented in this, our architecture supports them. For instance, our use of environment variables and AWS Secrets Manager shows that we've already planned for secure data handling. In a real-world deployment, these tools would also manage API keys and session tokens.

5. Team Contribution Summary

Team Member	Role	Responsibilities	Estimated Time
Mia Avellanet	Database Architect & Data Visualization	Designed the relational database schema, created the ER diagram, ensured third normal form, selected MySQL and Power BI, and wrote tool justification.	2-4 hours

Rebekah Sumter	Cloud Architect & Network Engineer	Selected AWS as the cloud provider, identified services, created the network architecture diagram, and defined port configurations.	2-3 hours
Belinda McDowell	Application Developer	Defined the application architecture, selected JavaScript, Node.js, and React, configured runtime and middleware, and explained server setup.	2-3 hours
Keyla Agurto	QA Analyst & Project Manager	Wrote the testing and quality assurance plan, defined authentication roles, created the contribution summary, and provided GitHub proof of collaboration.	2 hours

7. Challenges: One challenge was ensuring that each section of the architecture document maintained a consistent tone and level of technical detail. To solve this, we shared drafts early and reviewed each other's work to make sure the final version felt unified. Another challenge was staying organized across multiple platforms, including Google Docs, GitHub, and AWS. To manage this, we created a shared checklist and timeline to track who was responsible for each task and when it was due. This helped us stay on schedule and ensured nothing was missed during the submission process.