DESIGN AND IMPLEMENTATION FOR

Kaiju Academy

Version 1.0

Prepared by
Group A2

YU Ching Hei	**1155193237**	**chyu@link.cuhk.edu.hk**
Lei Hei Tung	**1155194969**	**1155194969@link.cuhk.edu.hk**
Ankhbayar Enkhtaivan	**1155185142**	**1155185142@link.cuhk.edu.hk**
Yum Ho Kan	**1155195234**	**1155195234@link.cuhk.edu.hk**
Leung Chung Wang	**1155194650**	**1155194650@link.cuhk.edu.hk**

The Chinese University of Hong Kong
Department of Computer Science and Engineering
CSCI3100: Software Engineering

March 27, 2025

Contents

Document Revision History

Version	Revised By	Revision Date	Comments
0.1	Group A2	2024-02-27	Added: –Initial document structure –Basic content outline
0.2	Leung Chung Wang	2024-02-28	Added: –Introduction framework –Summary of requirements –Project goals and stakeholders
0.3	YU Ching Hei	2024-03-01	Added: –Initial System Architecture design –Major components outline –AWS service integration details
0.4	YU Ching Hei	2024-03-02	Added: –Data Models section –Database schema –Request/response structures
0.5	Ankhbayar Enkhtaivan	2024-03-04	Added: –Interface Design –API structure –Authentication details –Data flow diagrams
0.6	Lei Hei Tung & Yum Ho Kan	2024-03-06	Added: –Component Design –Frontend components –Backend services –Interface mockups
0.7	Leung Chung Wang	2024-03-07	Added: –Assumptions section –Technical constraints –Operational assumptions –Dependencies
1.0	Group A2	2024-03-11	Updated: –Completed all sections –Added diagrams and technical details –Final review and integration

1. Introduction

Kaiju Academy is a **web-based e-learning platform** designed to make learning programming accessible and engaging. It combines modern Learning Management System (LMS) capabilities with interactive coding features, enabling users to learn at their own pace.

1.1 Summary of Requirements

Based on the requirements specification, Kaiju Academy includes:

- **User Management & Authentication:** Role-based access control for students, educators, administrators, and moderators with secure authentication.
- **Course Creation & Management:** Tools for creating, updating, and organizing courses with various content types (videos, PDFs, quizzes, coding exercises).
- **Interactive Learning:** Real-time code execution, automated grading, and personalized learning dashboards.
- **Community Features:** Discussion forums with moderation tools and notification systems.
- **Administrative Tools:** User analytics, system monitoring, and compliance management.

1.2 Quality Goals

Goal	Description
Performance	- Response time <2 seconds for 95% of users. - Code execution results in <5 seconds for 99% of submissions.
Scalability	- Support 10,000 concurrent users. - Horizontal scaling via AWS Lambda and SurrealDB sharding.
Reliability	- 99.9% uptime with automatic failover. - Recovery from failures within 5 minutes.
Security	- AES-256 encryption for data at rest. - TLS 1.2+ for data in transit. - Role-based permissions and MFA.

Table 1.1: Quality Goals for Kaiju Academy

1.3 Stakeholders

Stakeholder	Role & Responsibilities
Administrators	- Manage system health, user roles, and backups. - Monitor performance and enforce security policies.
Students	- Enroll in courses, complete assessments, and track progress. - Participate in forums and coding competitions.
Educators	- Create and update course content. - Grade submissions and provide feedback.
Content Moderators	- Moderate discussion forums. - Enforce community guidelines and resolve disputes.

Table 1.2: Stakeholders and Their Roles

2. System Architecture

2.1 Major Components

Frontend Application: A React application providing the user interface across all devices.

Backend Services: A Rust application providing the service functions handling the requests.

SurrealDB: Primary database managing all persistent data with multi-model support.

AWS Services: We use the following AWS services to implement the infrastructure of the system:

API Gateway: Manages all external requests, authentication, and routing to appropriate backends.

Lambda Functions: Serverless computing units organized by domain.

Cognito: Handles user authentication, MFA, and role-based access control.

IAM: Manages user and resource access permissions.

S3 Storage: Houses course materials, user uploads, and media content.

CloudFront CDN: Optimizes content delivery globally for static assets and course materials.

Code Execution Service: Sandboxed environment for running user-submitted code securely.

Notification Service: Manages email, in-app, and push notifications.

2.2 Component Relationships

The following diagram illustrates the relationships between the different components of the system:

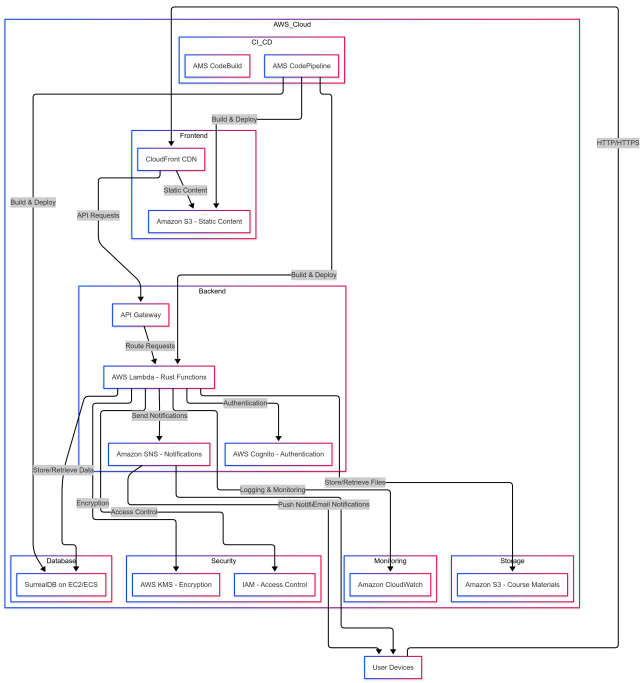


Figure 2.1: AWS Serverless Deployment Architecture

Key interactions between components include:

- Users access the system via web browsers or mobile devices, connecting to CloudFront CDN
- CloudFront delivers static content from S3 and routes API requests to API Gateway
- API Gateway manages all external requests and routes them to appropriate Lambda functions
- Lambda functions (built in Rust) handle business logic, interacting with SurrealDB for data storage

- AWS Cognito manages authentication and user identity before business logic executes
- Lambda functions use S3 for file storage and SNS for notifications
- CloudWatch provides logging and monitoring capabilities
- KMS and IAM handle encryption and access control respectively
- CI/CD pipeline with CodePipeline and CodeBuild automates deployment

2.3 Database

The database architecture of Kaiju Academy is built on SurrealDB, a multi-model database that provides flexibility in data modeling. It is hosted on AWS as a Infrastructure-as-a-Service (IaaS) using AWS EC2 for high availability and scalability, it facilitates the integration with other AWS services like Lambda, API Gateway, S3 etc. with increased efficiency and performance.

2.4 Security

2.4.1 Infrastructure Security

As our application is deployed on AWS as a Platform-as-a-Service (PaaS) using Lambda functions, the majority of infrastructure security is managed by AWS. We leverage the following AWS-managed security features:

- **Network Security:**
  - AWS-managed VPC isolation for Lambda functions
  - Built-in AWS Shield protection against DDoS attacks
  - AWS-configured security boundaries between services
- **Data Security:**
  - AWS-managed AES-256 encryption at rest
  - AWS-enforced TLS 1.2+ for data in transit
  - Automated key management through AWS KMS

2.4.2 Application Security

- **Input Validation:**
  - Request validation at API Gateway
  - Sanitization of user inputs
  - Rate limiting per user/IP
- **Code Execution Security:**
  - Dockerized isolated environments for isolated user code execution:
    * Leverage AWS Fargate to run containers without server management.
    * Ensure secure, scalable, and efficient execution of user-submitted code.
    * Package each user code submission into a Docker container.
    * Encapsulate dependencies and runtime configurations within the container.
    * Enable AWS Lambda to trigger these containers for seamless integration.
    * Maintain isolation and resource limits for each execution.
  - Our Lambda handling functions are built on Rust, which provides:
    * Memory safety through its ownership model, preventing data races and ensuring safe memory access.
    * Compile-time checks that catch potential errors before deployment, reducing runtime vulnerabilities.
    * Strict borrowing rules that enforce safe access patterns, minimizing the risk of buffer overflows.
    * A strong type system that helps prevent injection attacks by ensuring that data types are correctly handled.
    * Fast execution speed without sacrificing safety, enabling rapid processing of user-submitted code.

2.5 User Authentication and Authorization

2.5.1 Authentication Flow with AWS Cognito

- **User Registration:**
  - Cognito User Pools for email/password and social identity providers
  - Automated email verification workflows
  - Built-in MFA options (SMS, TOTP authenticator apps)
- **Session Management:**
  - Cognito-issued JWT tokens (ID, Access, and Refresh tokens)
  - Token validation via Cognito's public keys
  - Configurable token expiration policies

2.5.2 Authorization Model with AWS IAM

- **Role-Based Access Control:**
  - IAM roles mapped to Cognito user groups (Student, Educator, Admin)
  - Fine-grained IAM policies for service access
  - Temporary credentials via Cognito Identity Pools
- **Integration Points:**
  - Cognito authorizers for API Gateway endpoints
  - Resource-based policies for Lambda functions
  - IAM-based access control for AWS services (S3, DynamoDB)

2.6 Content Delivery

The content delivery infrastructure is built on AWS CloudFront CDN and S3, optimized for global distribution of both static and dynamic content.

2.7 Testability

2.7.1 Automated Testing

With the benefits of Rust's built-in support for a robust testing system, our app can undergo rigorous testing in the pipeline automatically. This automation ensures that every code change is validated through a series of tests, significantly enhancing the reliability and functionality of the application.

2.7.2 Types of Tests

Rust's testing framework allows for various types of tests, including:

- **Unit Tests:** These tests validate individual components or functions, ensuring that each part of the code behaves as expected in isolation.
- **Integration Tests:** These tests assess the interactions between different components, verifying that they work together correctly and that the overall system functions as intended.
- **Documentation Tests:** Rust allows for tests to be embedded within documentation comments, ensuring that examples remain accurate and functional as the code evolves.

2.7.3 Continuous Integration

By integrating automated testing into our continuous integration/continuous deployment (CI/CD) pipeline, we can ensure that every code change is thoroughly tested before deployment. This practice helps identify issues early in the development cycle, reducing the risk of bugs in production and maintaining high standards of quality and performance throughout the development process.

3. Data Models

3.1 Database Schema

The database design incorporates relational, document, and graph capabilities through SurrealDB's multi-model approach.

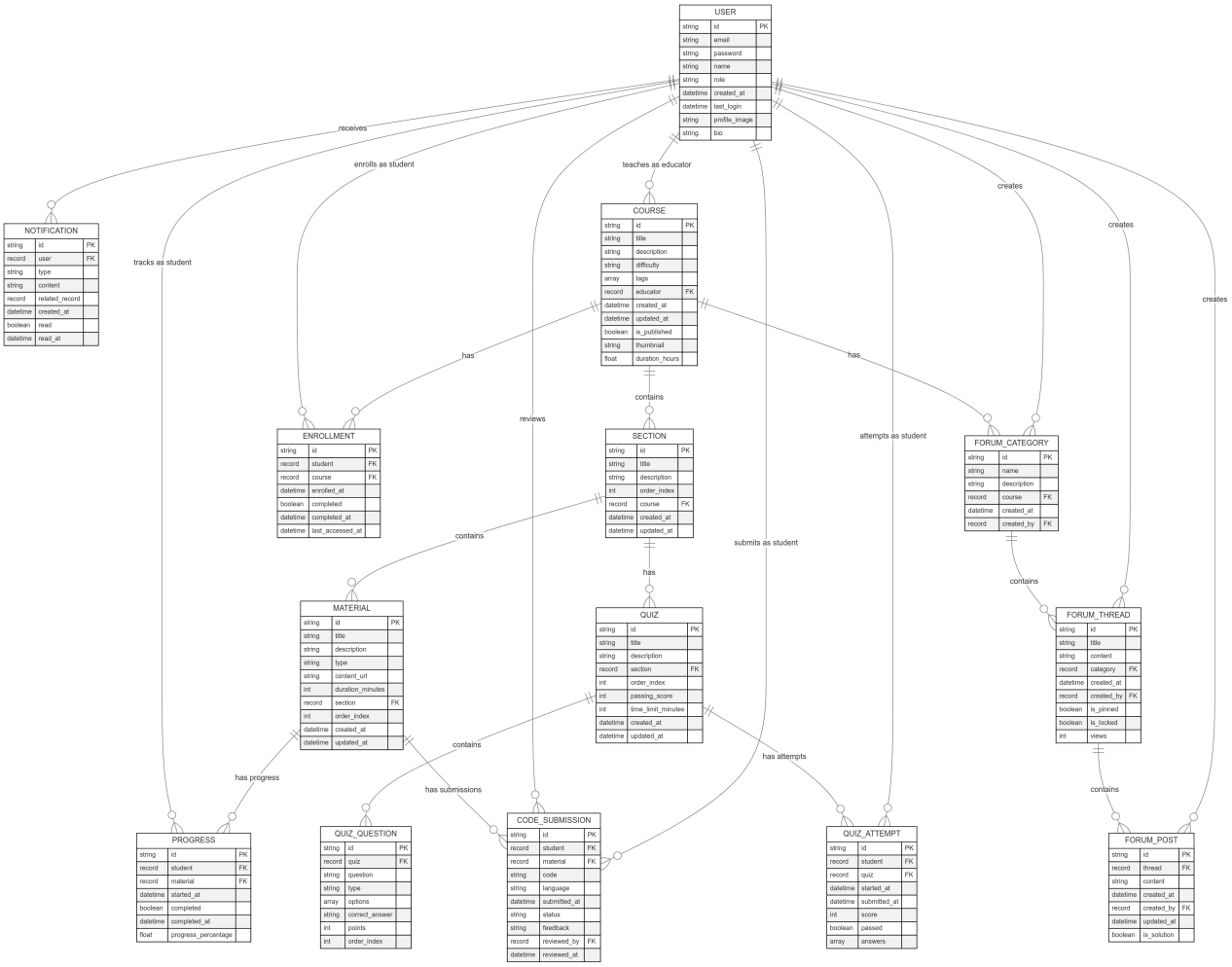


Figure 3.1: Database Schema

3.2 Request and Response Structure

AWS Lambda functions are event-driven, the request and response structure is defined in the Lambda function code. And in Rust SDK, the request and response structure is defined as

1. `LambdaEvent<ApiGatewayProxyRequest>`
2. `Result<ApiGatewayProxyResponse, Error>`

And it is an example of how we structure the response in the function handler:

```
    // other handler body code
    let response = ApiGatewayProxyResponse { // define response object
        status_code: 200,
        headers: std::collections::HashMap::new(),
        multi_value_headers: std::collections::HashMap::new(),
```

```
        body: Some(json!({ "message": "Hello from Kaiju Academy API!" }).
            to_string()),
        is_base64_encoded: Some(false),
    };
    Ok(response) // return the response as Option::Ok
```

3.3 Data Flow Sequence

The following diagram illustrates the flow of data through the system:

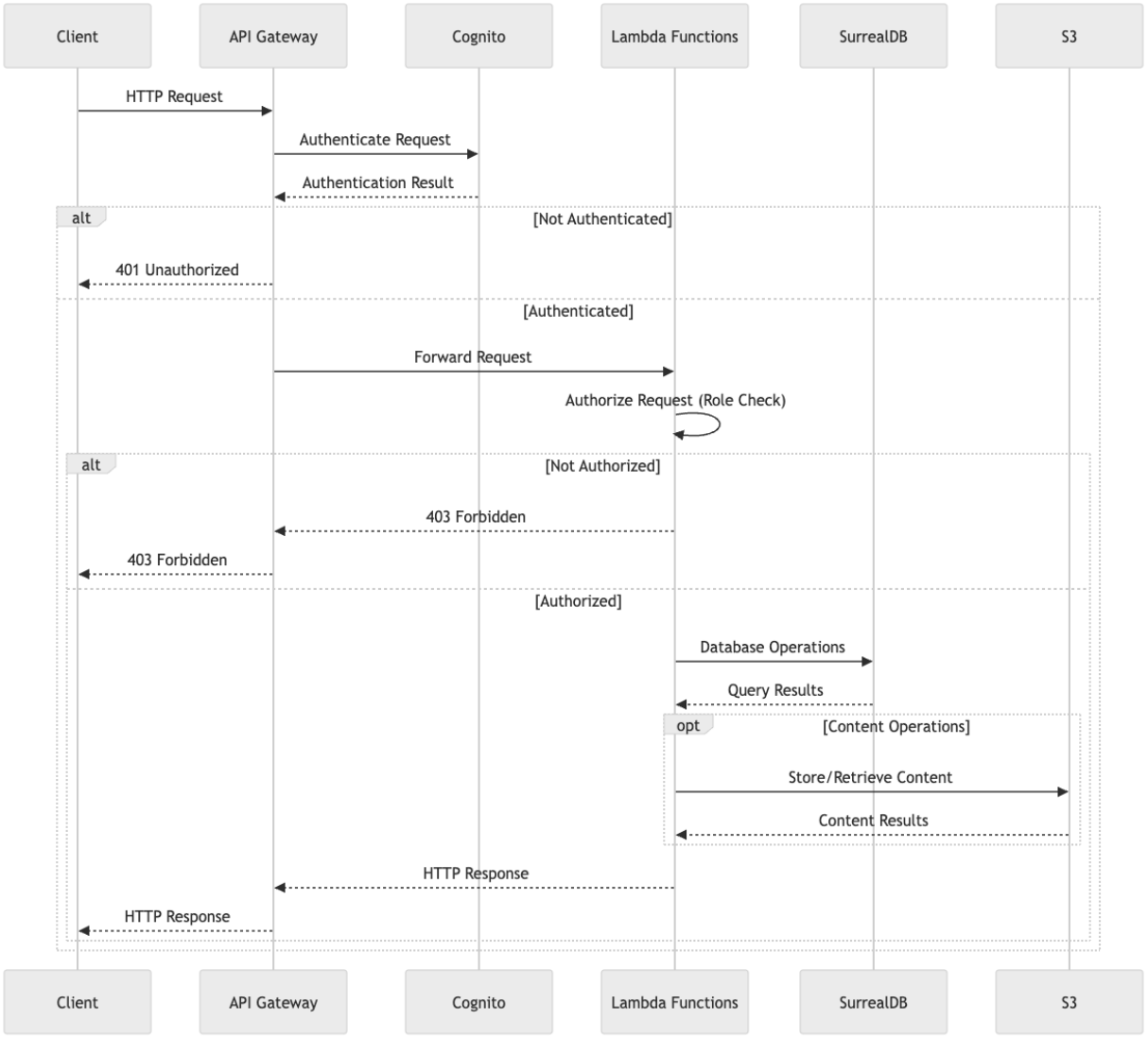


Figure 3.2: Data Flow Sequence Diagram

4. Interface Design

4.1 User Registration and Authentication

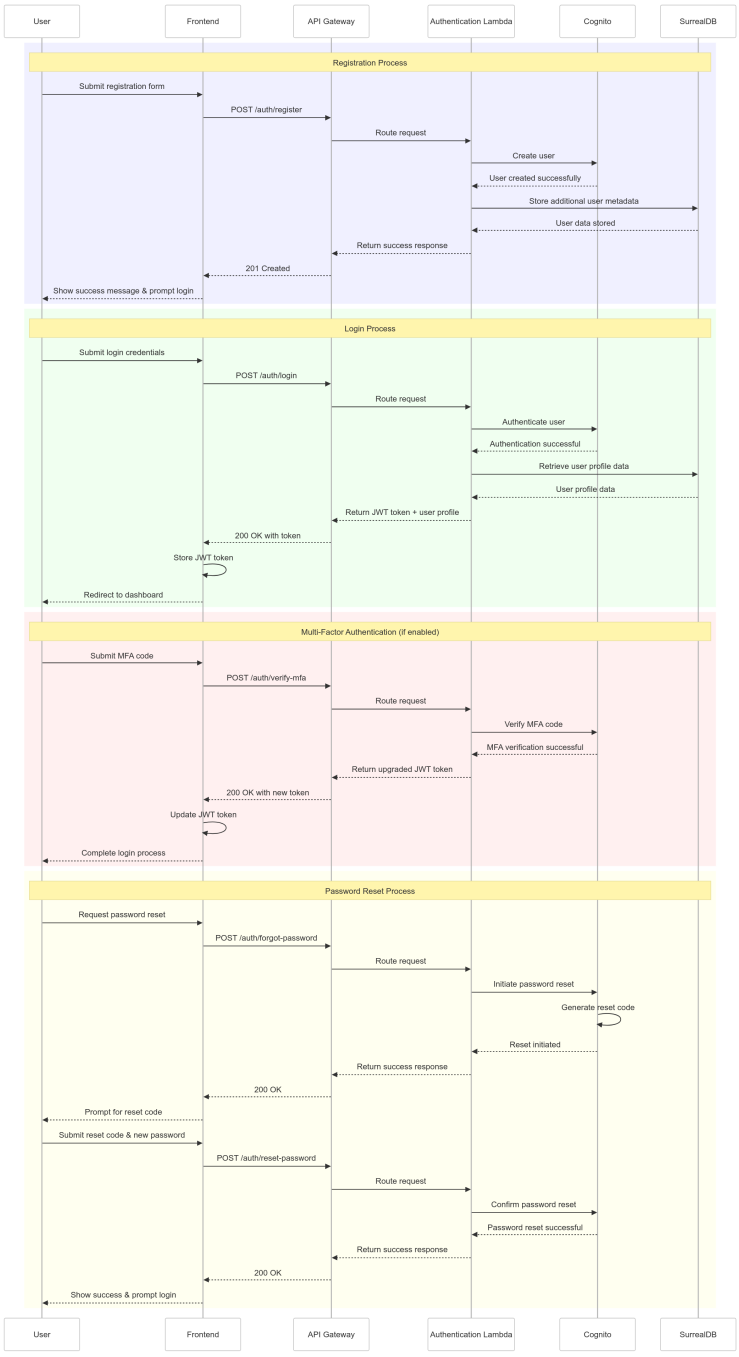


Figure 4.1: User Authentication Workflow

1. User submits registration data through frontend
2. API Gateway routes to Authentication Lambda
3. Lambda creates user in Cognito and SurrealDB
4. JWT token returned for authenticated session

4.1.1 Course Management

Course Content Management

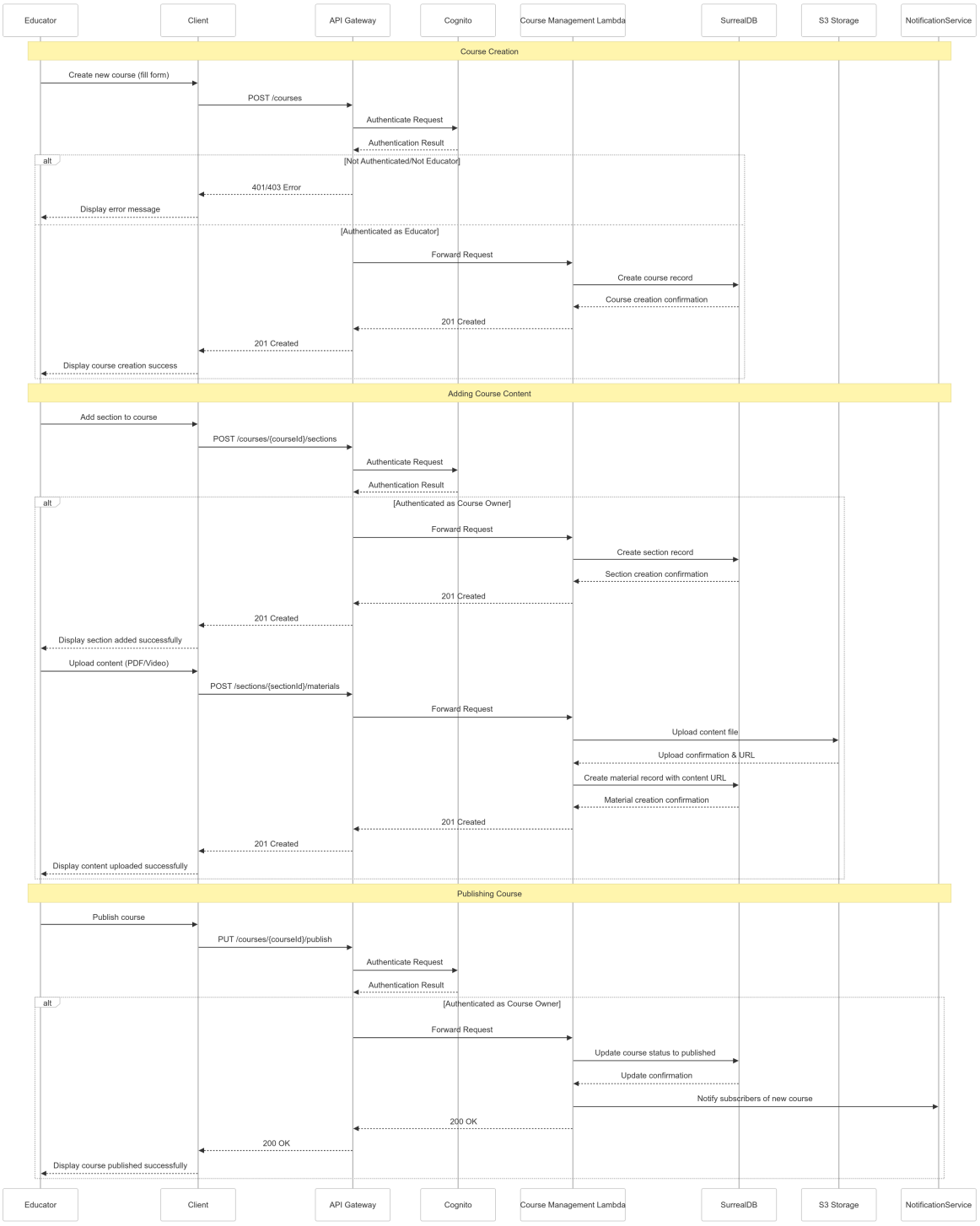


Figure 4.2: Course Management Workflow

1. Educator creates course structure through UI
2. Course metadata and structure stored in SurrealDB
3. Materials uploaded to S3 with references stored in database
4. Publishing workflow updates visibility and notifies subscribers

4.1.2 Course Access

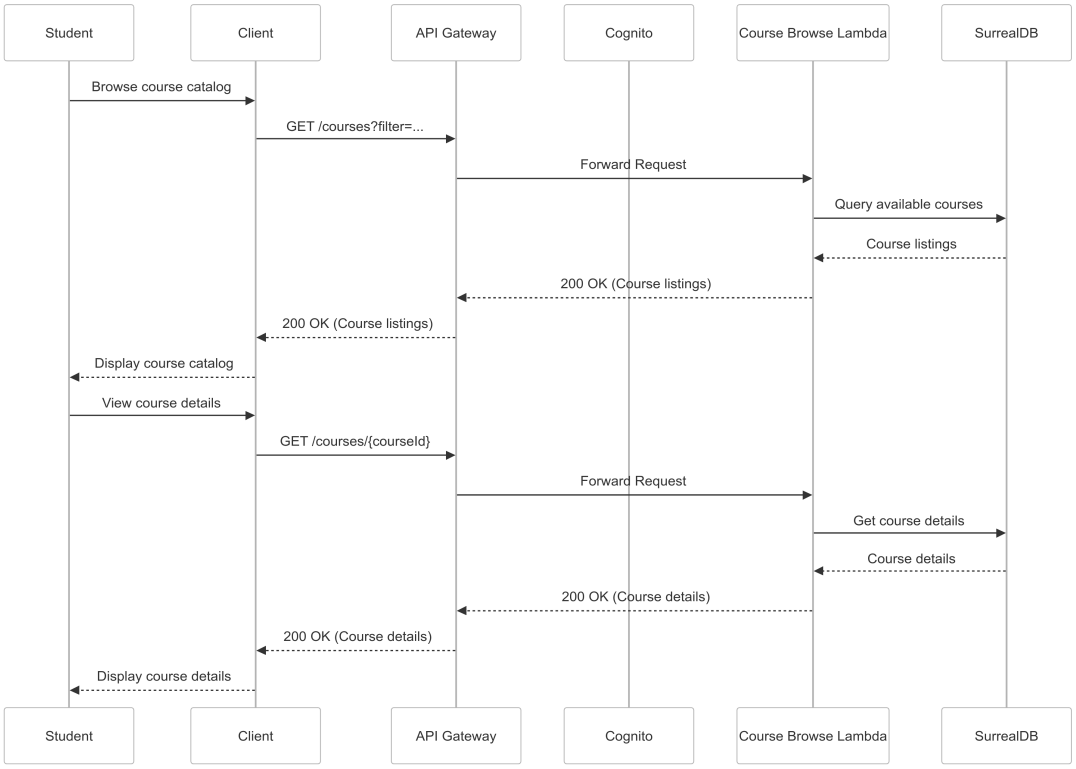


Figure 4.3: Course Access Workflow

4.1.3 Code Assessment and Grading

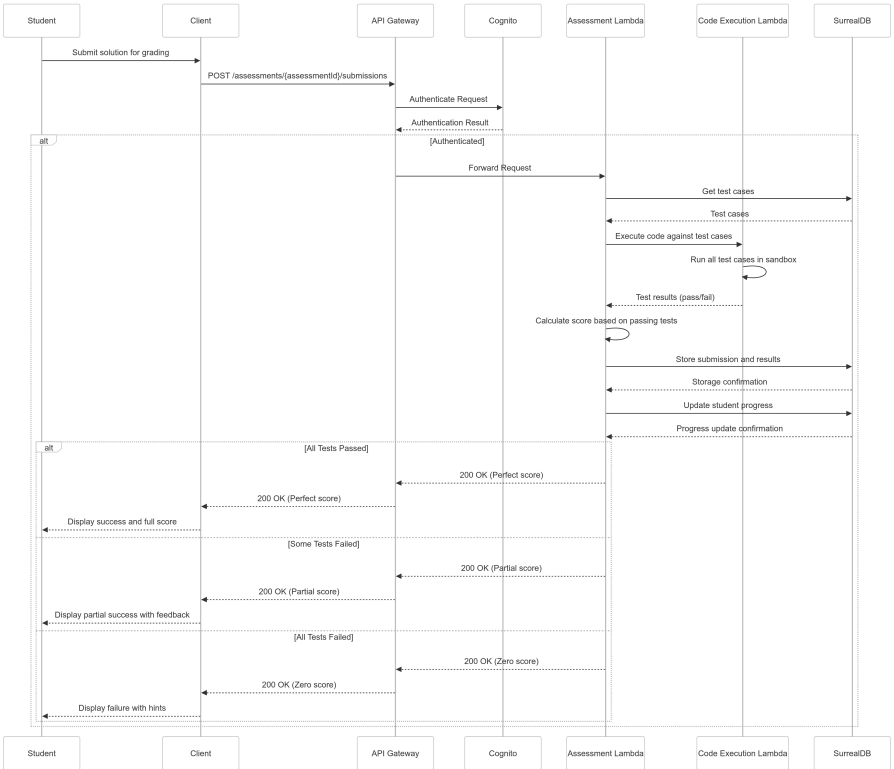


Figure 4.4: Code Auto-grading Workflow

4.1.4 Forum Moderation

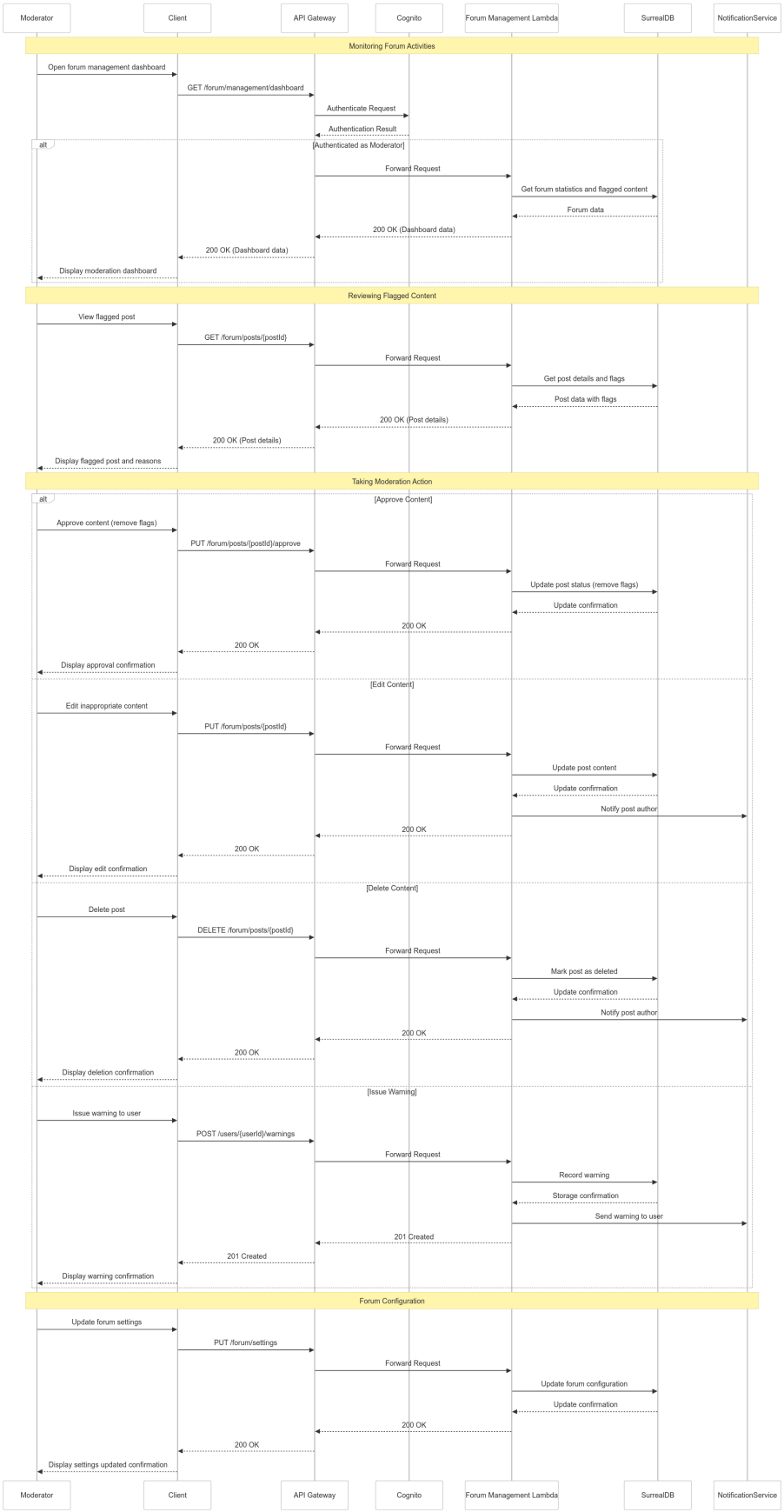


Figure 4.5: Moderator Managing Forum Process

4.2 Service Communication

4.2.1 External Interfaces

- RESTful APIs with JSON payloads
- Rate limiting (10 requests/second per user)
- Batch operations for efficiency
- Pagination for large result sets

4.2.2 Internal Communication

- AWS SNS/SQS for asynchronous event processing
- AWS Step Functions for complex workflows
- Direct Lambda invocations for synchronous operations

4.3 Security Implementation

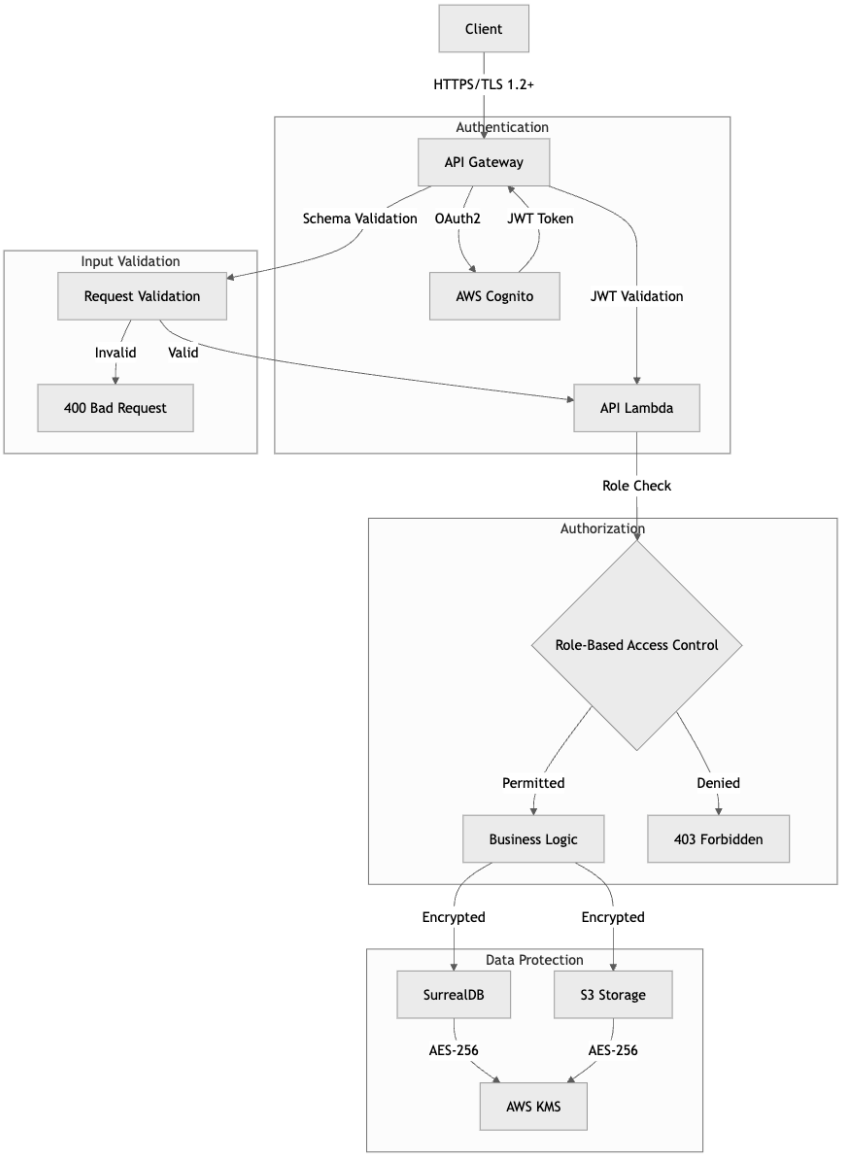


Figure 4.6: Security Implementation Diagram

4.3.1 Authentication and Authorization

- Multi-factor Authentication (MFA): Optional for all users, required for administrators
- OAuth2 Integration: Support for Google, GitHub, and Microsoft accounts
- Role-based Access Control (RBAC): Granular permissions based on user roles
- JWT for API Authentication: Secure, stateless authentication for API requests

4.3.2 Data Protection

- Encryption at Rest: AES-256 encryption for all sensitive data
- Encryption in Transit: TLS 1.2+ for all network communications
- PII Protection: Minimized collection of personally identifiable information
- Data Retention Policies: Automated purging of unnecessary data

4.4 Exception Handling

4.4.1 Expected Exceptions

We followed the standardized error model in the API Gateway and Lambda functions, expected the following error types:

- Authentication errors (401)

  Possible causes: Invalid username/password combinations or expired authentication tokens.

  Raise: 401 Unauthorized
- Authorization errors (403)

  Possible causes: Insufficient user permissions or attempts to access restricted resources.

  Raise: 403 Forbidden
- Validation errors (400)

  Possible causes: Malformed input data, missing required fields, or invalid data formats.

  Raise: 400 Bad Request
- Not found errors (404)

  Possible causes: Requests for non-existent endpoints or resources that have been deleted or never existed.

  Raise: 404 Not Found
- Database errors (500)

  Possible causes: Query syntax errors, connection timeouts, or database server outages.

  Raise: 500 Internal Server Error
- Internal server errors (500)

  Possible causes: Unhandled exceptions, misconfigurations, or resource exhaustion within the application.

  Raise: 500 Internal Server Error
- External service errors (502)

  Possible causes: Network issues, timeouts, or unavailability of downstream services.

  Raise: 502 Bad Gateway
- Rate limit errors (429)

  Possible causes: Exceeding the allowed number of requests in a specified time frame due to aggressive client behavior.

  Raise: 429 Too Many Requests

4.4.2 Error Triggering Mechanisms

- **API Request Validation:**
  - Automatic schema validation at API Gateway
  - Input parameter validation in Lambda functions
  - Business logic validation in service layer
- **Security Checks:**
  - JWT token validation
  - Permission verification against IAM policies
  - Resource ownership verification
- **External Integrations:**
  - Timeout detection for external services
  - Error propagation from downstream systems
  - Network failure detection

4.4.3 Error Handling Strategy

- **Error Processing Flow:**
  - Error detection at appropriate layer
  - Conversion to standard AppError type
  - Automatic mapping to API Gateway responses
  - Logging with correlation IDs
- **Client-Side Error Handling:**
  - Graceful degradation of UI
  - Retry mechanisms with exponential backoff
  - User-friendly error messages
  - Guided recovery steps

4.4.4 Error Handling Example

```rust
/// Convert an application error to an API Gateway response
impl From<AppError> for ApiGatewayProxyResponse {
    fn from(error: AppError) -> Self {
        let (status_code, error_type) = match &error {
            AppError::Authentication(_) => (401, "Authentication Error"),
            AppError::Authorization(_) => (403, "Authorization Error"),
            AppError::NotFound(_) => (404, "Not Found"),
            AppError::Validation(_) => (400, "Validation Error"),
            AppError::Database(_) => (500, "Database Error"),
            AppError::Internal(_) => (500, "Internal Server Error"),
            AppError::ExternalService(_) => (502, "External Service Error"),
            AppError::RateLimit(_) => (429, "Rate Limit Exceeded"),
        };
        // Some body implementation hidden...
        ApiGatewayProxyResponse {
            //ApiGatewayProxyResponse struct definition for error response
        }
    }
}
```

5. Component Design

5.1 Frontend Components

5.1.1 Navbar Component

- **Intention:** Provides top navigation links, search functionality, notification management, and user profile access.
- **Input:** User interactions (clicks, search queries), notification data from backend.
- **Output:** Navigation actions, search results, notification displays, profile dropdown.

5.1.2 Dashboard Component

- **Intention:** Displays enrolled courses, progress metrics, deadlines, and activity visualizations.
- **Input:** Course data, progress metrics, and deadlines from API.
- **Output:** Rendered UI cards, charts (line/pie), and progress indicators.

5.1.3 Code Editor Component

- **Intention:** Provides syntax highlighting, code execution environment, and result display.
- **Input:** User code, programming language (enum), test cases from backend.
- **Output:** Code package to backend for sandboxed execution and awaits results.

5.2 Backend Components

5.2.1 Authentication Service

- **Intention:** Handles user registration, login, token management, and access control.
- **Input:** User credentials (email, password), registration data, token verification requests.
- **Output:** JWT tokens, user profiles, authentication status, error messages.
- **Core Algorithm:** Bcrypt for password hashing, RSA-256 for JWT generation and validation.

5.2.2 Course Management Service

- **Intention:** Manages course creation, content organization, and publishing workflow.
- **Input:** Course objects (title, description, modules), content files, publishing requests.
- **Output:** Course IDs, organized content structures, success/error messages.
- **Core Algorithm:** Assigns `order_index` based on insertion sequence for material ordering.

5.2.3 Assessment Service

- **Intention:** Executes user code safely, grades submissions, and provides feedback.
- **Input:** User code, programming language (enum), test cases from database.
- **Output:** Execution results, scores, detailed feedback to frontend.
- **Core Algorithm:** Compares stdout with expected output and assigns partial scores for edge cases.

5.2.4 Forum Service

- **Intention:** Manages forum threads/posts and handles content moderation.
- **Input:** Post objects (content, thread ID), moderation actions.
- **Output:** Post IDs, updated thread structure, moderation log entries.
- **Core Algorithm:** Status-based content filtering and notification triggers for moderation actions.

6. User Interface Design

6.1 Design Principles

The UI design follows these core principles:

- **Consistency:** Unified design language across all pages
- **Accessibility:** WCAG 2.1 AA compliance for all users
- **Responsiveness:** Optimal experience across devices (320px and up)
- **Progressive Enhancement:** Core functionality works without JavaScript
- **Feedback:** Clear indication of system status and actions

6.2 User-wise Navigation Flow

6.2.1 Student

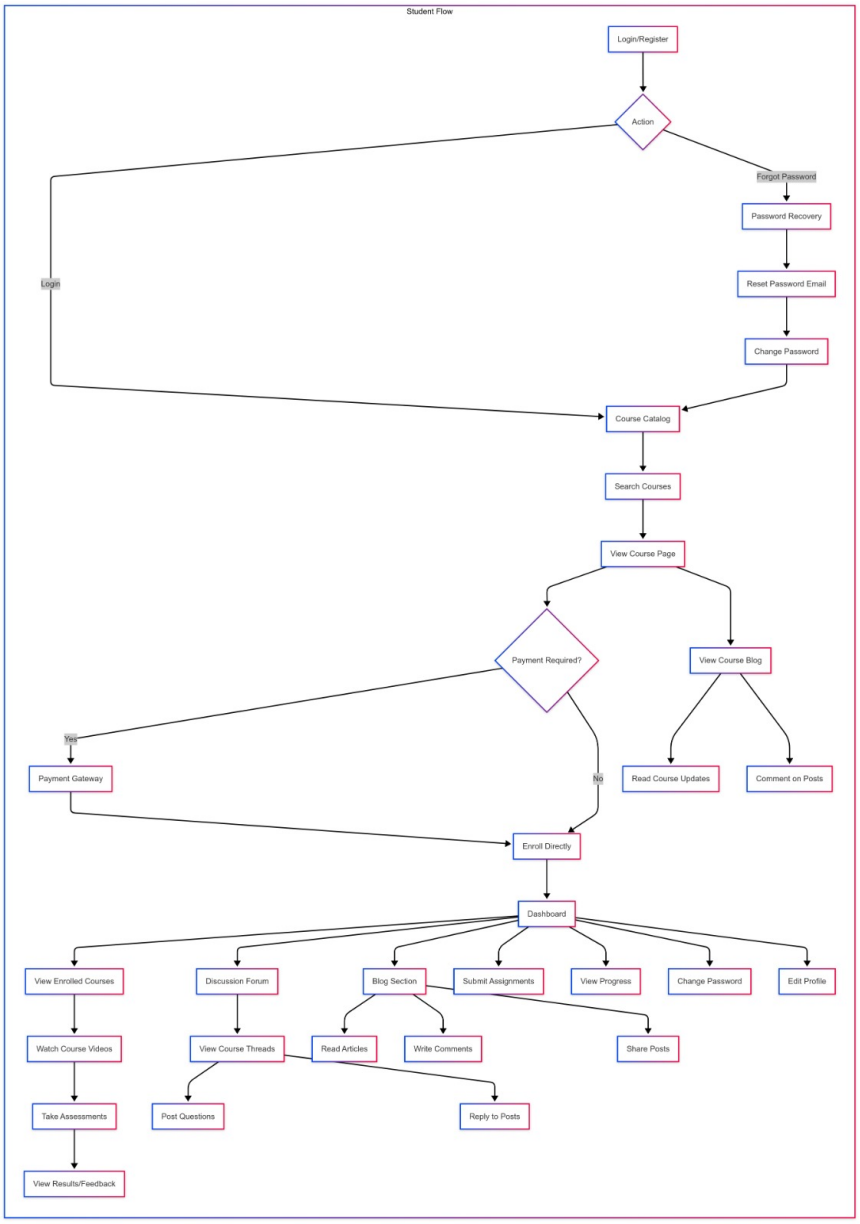


Figure 6.1: Student Navigation Flow

6.2.2 Educator

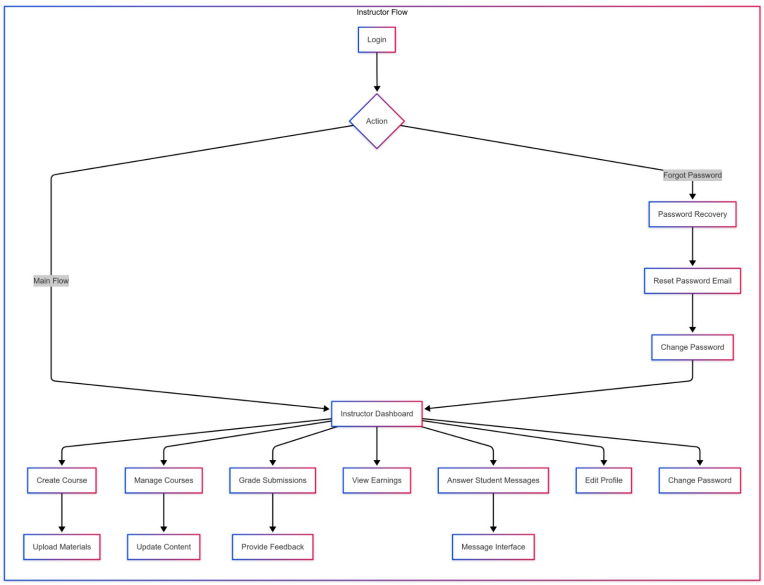


Figure 6.2: Educator Navigation Flow

6.2.3 Moderator

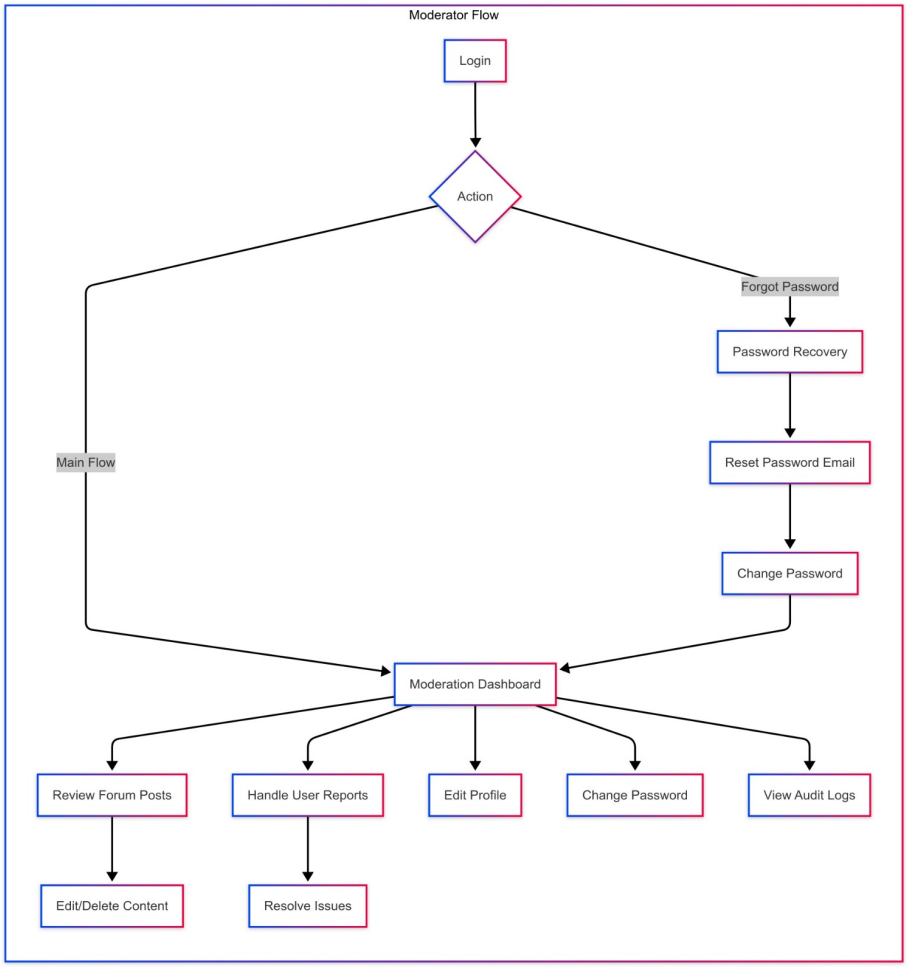
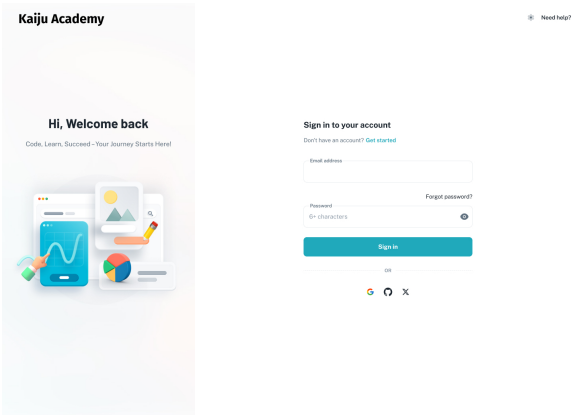


Figure 6.3: Moderator Navigation Flow

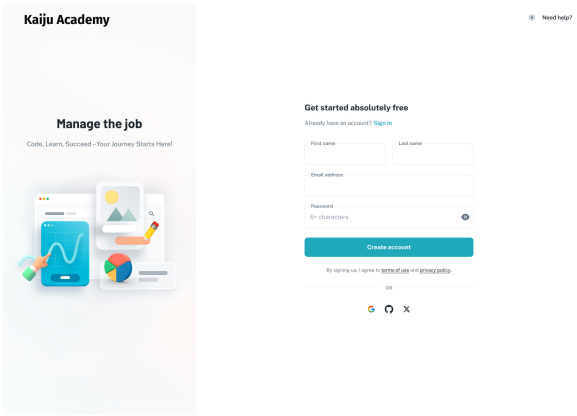
6.3 Interface Storyboards

This section showcases the key user interface screens and their relationships, depicting the actual flow of user interactions within the application.

6.3.1 Authentication Flow



(a) Sign In Screen

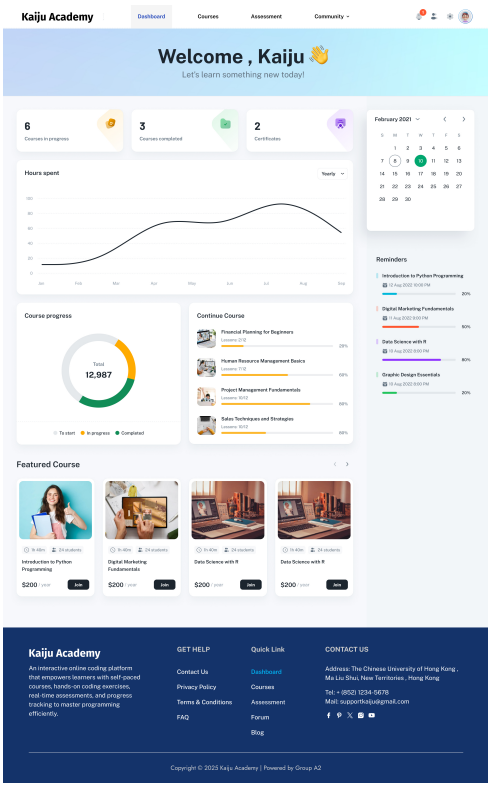


(b) Sign Up Screen

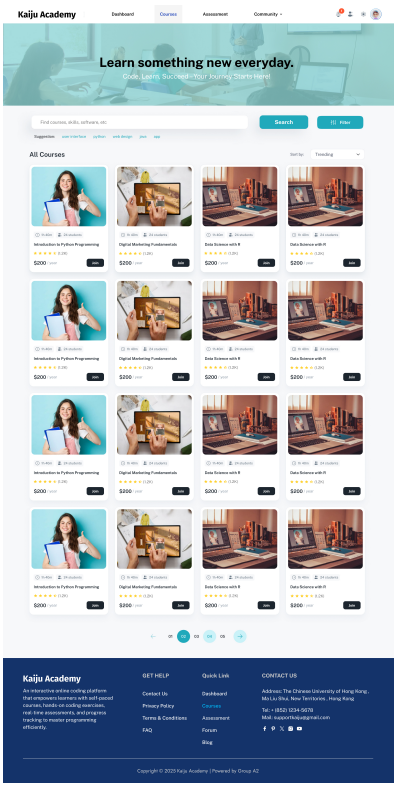
Figure 6.4: Authentication Screens

6.3.2 Student Learning Journey

After login, the student will be redirected to the dashboard.



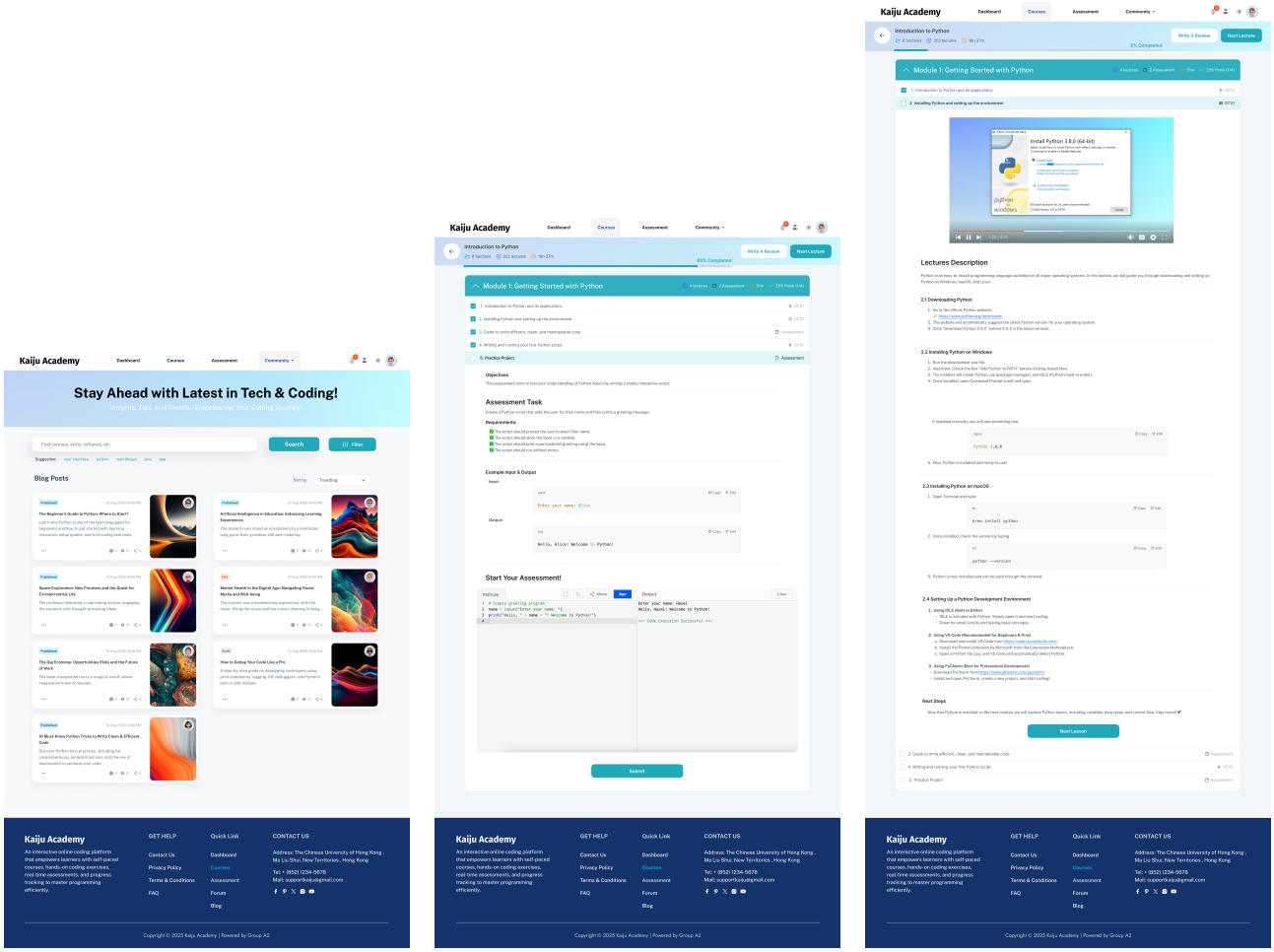
(a) Student Dashboard



(b) Course Catalog

Figure 6.5: Student courses view

6.3.3 Learning Environment



(a) Learning Blog **(b)** Assessment Interface **(c)** During Course Interface

Figure 6.6: Learning and Assessment Interface

6.4 Accessibility Considerations

- **Screen Reader Support:** Proper ARIA labels and semantic HTML
- **Text Resize:** Interface remains functional when text is enlarged 200%
- **Motion Control:** Animations can be disabled via prefers-reduced-motion
- **Alternative Text:** All images include descriptive alt text
- **Light/Dark Mode:** Option to switch between light and dark themes for user preference
- **Font Size Adjustment:** Users can easily enlarge or reduce font size for better readability
- **Color Blindness Support:** Special color sets available to accommodate color blindness

6.5 Responsive Design

- **Breakpoints:** Key breakpoints at 480px, 768px, 1024px, and 1440px
- **Layout Adaptations:**
  - Single column layout on mobile
  - Sidebar appears as overlay on small screens
  - Multi-column layout on tablets and larger
  - Expanded dashboard visualizations on desktop
- **Touch Targets:** Minimum 44×44px for all interactive elements

7. Assumptions

7.1 Technical Constraints

7.1.1 Hardware Constraints

- **Minimum Device Specifications:** Users require devices with:
  - At least 4GB RAM for smooth performance
  - Screen resolution of 320px width minimum
  - Internet connection with 5 Mbps minimum bandwidth
- **Server Environment:** Development and deployment assumes:
  - AWS cloud infrastructure
  - Serverless architecture with AWS Lambda
  - No dedicated hardware requirements
  - Horizontal scaling capabilities

7.1.2 Software Constraints

- **Browser Support:** Application designed for:
  - Modern browsers (Chrome, Firefox, Safari, Edge - latest 2 versions)
  - HTML5 and JavaScript required
- **Mobile Support:**
  - Responsive web design (no native app initially)
  - Mobile browser focused (iOS Safari, Android Chrome)

7.2 Operational Assumptions

7.2.1 Load and Performance

- Peak submission rate: 10,000 code submissions per minute
- 80/20 read/write ratio for database operations
- Code execution takes maximum 5 seconds per submission

7.2.2 Development Process

- CI/CD pipeline with automated testing
- Feature branching development workflow

7.2.3 Maintenance and Support

- 99.9% uptime requirement excluding planned maintenance
- Automated backups performed daily and retained for 30 days

7.3 Dependencies

7.3.1 Third-Party Services

- Authentication via OAuth providers (Google, GitHub, Microsoft)
- Payment processing through Stripe
- Email delivery through Amazon SNS
- CDN services through AWS CloudFront
- Media transcoding through AWS MediaConvert