Roads to Rome

Table of Contents

1. System Description

Overview

Roads to Rome is a comprehensive online learning platform built with modern web technologies. It enables students to browse and enroll in courses, instructors to create and manage educational content, and administrators to moderate the platform. The system features an embedded code execution environment, AI-powered quiz generation, and a robust course management workflow.

Technology Stack

Frontend

- **Framework**: React 18 with TypeScript
- **Build Tool**: Vite
- **UI Library**: shadcn/ui with Tailwind CSS
- **State Management**: React Context API
- **Routing**: React Router v6
- **HTTP Client**: Axios

Backend

- **Runtime**: Node.js with TypeScript
- **Framework**: Express.js
- **Database**: MongoDB with Mongoose ODM
- **Authentication**: JWT (access + refresh tokens) + GitHub OAuth
- **File Storage**: Supabase Storage
- **Code Execution**: Piston API
- **AI Integration**: Google Gemini API

Infrastructure

- **Frontend Hosting**: Vercel
- **Backend Hosting**: Render
- **CI/CD**: GitHub Actions
- **Containerization**: Docker + Docker Compose

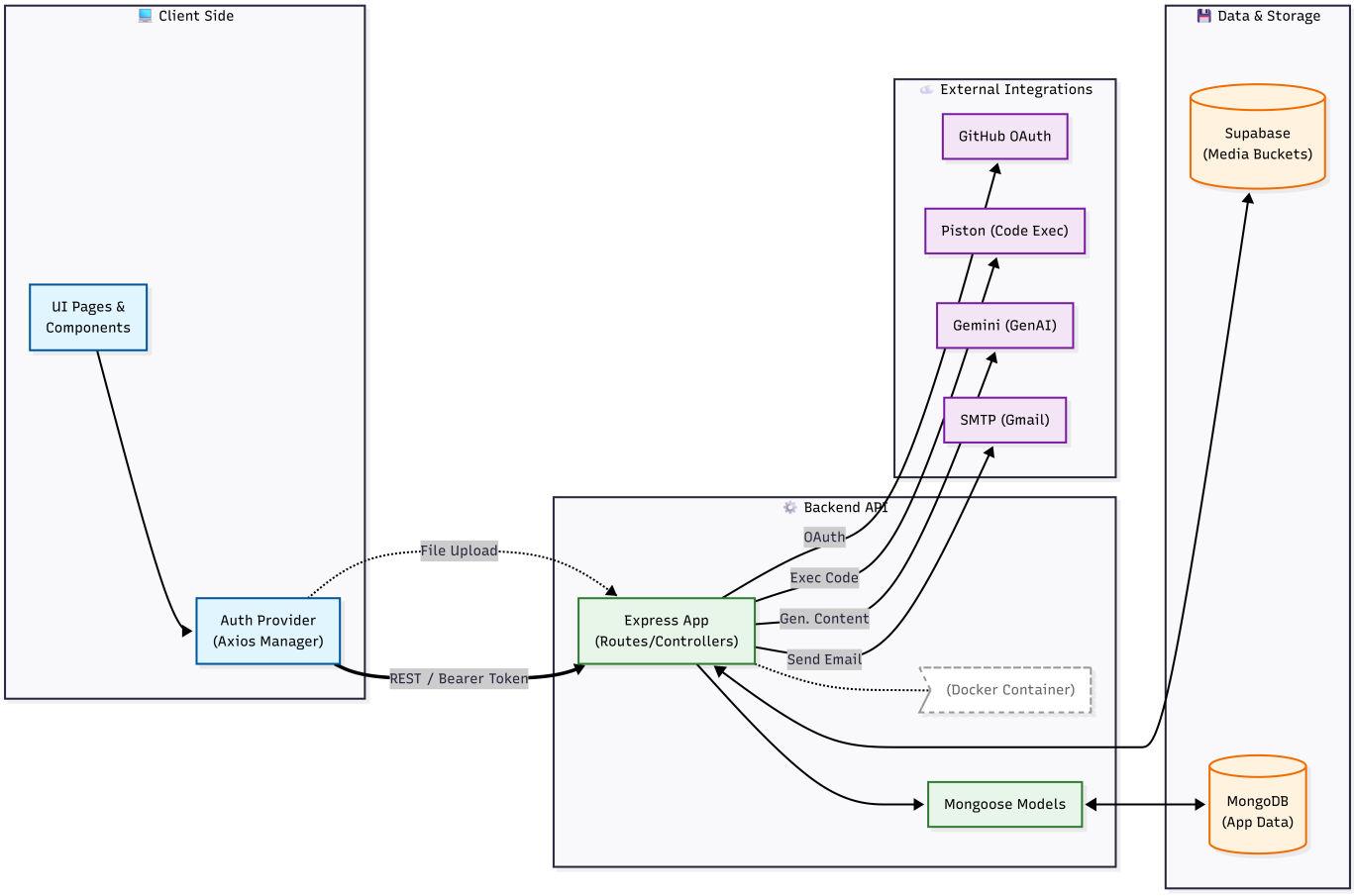
Key Features

- **Multi-Role System**: Guest, Student, Instructor, and Admin roles with distinct capabilities
- **Course Management**: Create, edit, publish, and moderate courses with rich content
- **Interactive Learning**: Video lessons, quizzes with multiple question types, and progress tracking
- **Embedded IDE**: In-browser code editor supporting Python and JavaScript
- **AI Quiz Generator**: Automatically generate quiz questions from lesson content
- **Budget System**: Coin-based payment system (1 coin = 1,000 VND)
- **Course Moderation**: Admin workflow for approving/rejecting instructor-submitted courses
- **Real-time Progress**: Track lesson completion and course progress
- **Responsive Design**: Mobile-first approach with full desktop support

Architecture

Roads to Rome follows a modern **client-server architecture** with clear separation of concerns:

- **Presentation Layer**: React SPA with component-based UI
- **API Layer**: RESTful Express.js backend
- **Data Layer**: MongoDB for persistent storage
- **External Services**: Supabase (file storage), Piston (code execution), Gemini (AI), GitHub (OAuth)



2. Team Information

Student ID	Full Name
22127095	Do Dinh Hai
22127123	Tran Nguyen Minh Hoang
22127131	Le Ho Phi Hoang

3. Project Plan Tracker

Development Workflow

Our team follows an agile approach with **weekly sprints**, leveraging modern collaboration tools and methodologies:

- **Sprint Structure**: 1-week sprints aligned with course schedules and project milestones
- **Task Management**: AI-assisted task splitting ensures fair and equal work distribution
- **Task Assignment**: Wheel of Fortune method for unbiased, random task allocation
- **Documentation**: Notion serves as our single source of truth for task tracking, environment variables, and project documentation
- **Communication**: Facebook Messenger for daily updates, quick coordination, and asynchronous communication

Weekly Milestones

Week 1 — Authentication, Authorization, Layout & Dashboard Foundation *(16-22 hours)*

Goal: Build the core technical foundation including authentication, authorization, design system, and dashboard prototype.

Key Deliverables:

- Repository setup with FE/BE architecture
- Database design (users, courses, lessons, enrollments, quizzes, questions, progress, submissions, reviews)
- Email/password authentication + GitHub OAuth
- Role-based authorization (guest, student, instructor, admin)
- AppShell layout with global theme system
- Dashboard prototype with summary cards, progress overview, and activity list

Week 2 — Guest Browsing + Student Learning MVP *(26-35 hours)*

Goal: Complete the main learning entry flow: browse → search → enroll → view lessons.

Key Deliverables:

- Homepage with latest courses, categories, and trending tags
- Course list page with filtering (category, tag) and pagination/infinite scroll
- Course detail page with locked content for guest users
- Full-text search across courses
- Public navigation (navbar)
- Enroll/unenroll flow with budget management
- Lesson viewer supporting text content and embedded video
- Student dashboard showing enrolled courses, completion status, and suggested courses

Week 3 — Instructor + Manual Quiz *(31-34 hours)*

Goal: Enable instructors to create courses/lessons/quizzes; allow students to take quizzes with instant scoring.

Key Deliverables:

- Course management (create, edit, delete, publish/unpublish)
- Lesson management with text content and video attachments
- Instructor dashboard showing created courses, student enrollment, and quiz analytics
- Manual quiz creation with three question types: single-choice, multiple-choice, and drag-and-drop
- Student quiz-taking functionality with auto-grading and score saving

Week 4 — Admin Module + AI Quiz Generator MVP *(30-34 hours)*

Goal: Build admin control panel and AI-powered quiz generation feature.

Key Deliverables:

- User management (view all users, assign roles, lock/unlock accounts)
- Course moderation workflow (draft → pending → approved/published → rejected/hidden)
- System statistics dashboard (total users, courses, enrollments)
- AI quiz generator: generate 5-10 MCQs from lesson text using AI
- Instructor review interface for AI-generated quizzes (preview, edit, approve)

Week 5 — Embedded IDE + Testing + Final Release *(~35 hours)*

Goal: Add in-browser coding capability, polish UI/UX, complete testing, and deploy.

Key Deliverables:

- Embedded IDE with code editor, Run button, and output console
- Multi-language runtime support (Python + JavaScript minimum)
- UI/UX refinement (responsive design, loading states, error handling)
- Testing infrastructure setup with test files for core functionality
- Production deployment with demo data
- Final documentation and demo video

Task Distribution Philosophy

- **Fairness**: AI analysis + random assignment ensures balanced workload across team members
- **Transparency**: All tasks documented in Notion with clear descriptions and weekly organization
- **Flexibility**: Messenger-based updates allow asynchronous coordination without rigid meeting schedules
- **Skill Development**: Task rotation exposes team members to diverse parts of the codebase
- **Continuous Improvement**: Weekly sprint cycles enable rapid adjustments and iterative development

---

4. Functionalities Analysis

User Guide

This section describes the functionalities available to different user roles and how to use the deployed application.

Access & URLs

- Frontend: deployed URL (or `http://localhost:5173` locally). Backend must be reachable at `http://localhost:3000` (or your deployed API base).
- All primary pages are routable: `/` home, `/courses` catalog, `/courses/:id` detail, `/dashboard`, `/enrolment`, `/courses/:courseId/lessons/:lessonId`, quiz paths, and role-specific pages listed below.

Accounts & Authentication

- Sign up: `/signup` (email/password). Login: `/login`. GitHub OAuth is also supported.
- Forgot/reset password: `/forgot-password` to request a link; `/reset-password/:token` to set a new password.
- Tokens: access token in memory; refresh token is an HttpOnly cookie. If you hit auth issues, log out and back in to refresh.

Navigation Basics (Header)

- Home and Courses are always available. When authenticated: `Dashboard` appears in the profile menu. Students see `My Enrollments`. Instructors see `AI Quiz`.
- Logout via the avatar dropdown; you will be redirected to `/login`.

Roles and Capabilities

- **Student**: browse and search courses, view course details, enroll, watch lessons, mark lessons complete, take quizzes, leave feedback/comments, track progress on `/dashboard` and `/enrolment`.

- **Instructor**: all student abilities plus create/edit courses and lessons, manage pricing/premium flags during creation, create quizzes (`/quizzes/new`), edit quizzes (`/quizzes/:id/edit`), and generate AI quizzes (`/ai-quiz`).
- **Admin**: full access. Review and moderate courses at `/course` (list) and `/course/:id` (review), manage users/roles/locks/budgets, adjust pricing, and view stats on `/admin`.

Course Catalog and Enrollment

- Browse catalog: `/courses` supports search/filter UI (front-end) backed by `/api/courses`.
- Course detail: `/courses/:id` shows description, lessons, instructor info, comments, and pricing/premium state.
- Enroll: click *Enroll* to create an enrollment. If a course has a price, the amount is deducted from your budget; if budget is insufficient, enrollment is blocked.
- Enrollments and progress are visible at `/enrolment` and `/dashboard`.

Lessons and Learning Flow

- Lesson viewer: `/courses/:courseId/lessons/:lessonId` (auth required) streams lesson content/video and attachments.
- Mark complete to update progress for the course; completion state feeds dashboards and recommendations.

Quizzes

- Take quizzes from course/lesson context: `/courses/:courseId/quiz/:quizId` (auth required). Attempt history is available via the backend `/api/quiz/:quizId/history`.
- Instructors create quizzes at `/quizzes/new`, edit at `/quizzes/:id/edit`, and can auto-generate quiz questions using `/ai-quiz`.

Instructor Course & Lesson Workflow

- Create course: `/courses/create` (INSTRUCTOR). Provide title, category, tags, short description, price/premium flag, thumbnail.
- Edit course: `/courses/:id/edit` (INSTRUCTOR). Update metadata or pricing.
- Manage lessons: `/courses/:courseId/lessons/create` to add lessons; `/courses/:courseId/lessons/:lessonId/edit` to update content, video, attachments, and ordering.

Admin Workflow

- Course review: `/course` lists pending/filtered courses; `/course/:id` opens review detail to approve/reject/hide and set price/premium.
- User management and stats: `/admin` gives access to admin dashboard data (users, roles, locks, budgets, stats).

Payments (Development vs Production)

- **Budget System**: Users have a coin-based budget (1 coin = 1,000 VND). Course prices are set in coins, and enrollment deducts the price from the user's balance.- Development: budget top-ups use the mock endpoint `/api/payments/mock` (surfaced via the top-up dialog in the student dashboard). No real charges occur.
- Enrollment: deducts from the user budget when price > 0; it fails with an insufficient-budget message if funds are low.
- Production: wire a real payment provider for top-ups and align the enrollment flow with that provider’s status and errors.

Troubleshooting

- Auth issues (401/403): log out/in to refresh tokens; confirm your role matches the required page (e.g., INSTRUCTOR for creation paths, ADMIN for review paths).
- Missing data: check that the backend is running and environment variables are set; `/api/health` should return OK.
- Upload/thumbnail/video problems: verify backend `POST /api/uploads` is reachable and that CORS settings permit your frontend origin.
- Quiz not loading: ensure you are logged in and the quiz URL includes both `courseId` and `quizId`.

API Reference

A concise list of the HTTP API endpoints exposed by the backend. For each endpoint we show the method, path, authentication/authorization requirements, a short description, and key request fields.

Method	Path	Auth	Description	Request
GET	`/api/health`	Public	Health check	—
POST	`/api/uploads`	Public	Upload a single file (returns public URL)	multipart/form-data: `file`

Authentication

Method	Path	Auth	Description	Request
POST	`/api/auth/login`	Public	Login with email & password; returns `accessToken` and sets `refreshToken` cookie	JSON: `{ email, password }`
GET	`/api/auth/github`	Public	Redirect to GitHub OAuth	—
GET	`/api/auth/github/callback`	Public	GitHub OAuth callback (query `code`)	query: `code`
POST	`/api/auth/logout`	Public	Clear `refreshToken` cookie	—
POST	`/api/auth/register`	Public	Register new user	JSON: `{ email, password, role, username, fullName }`
POST	`/api/auth/refresh-token`	Public (reads cookie)	Refresh access token using `refreshToken` cookie	—
POST	`/api/auth/forgot-password`	Public	Trigger password-reset email	JSON: `{ email }`
POST	`/api/auth/change-password`	Public	Reset password with token	JSON: `{ token, newPassword }`
GET	`/api/auth/me`	Auth required	Get current user's profile	—

Courses

Method	Path	Auth	Description	Request
GET	`/api/courses`	Public	List courses (supports pagination & filters via query params)	query: `page`, `limit`, `q`, `status`
GET	`/api/courses/suggestions`	Auth required	Get personalized course suggestions for the logged-in user	query: optional filters
GET	`/api/courses/:id`	Public	Get course by id	—
POST	`/api/courses`	Auth + Instructor/Admin	Create a course (supports thumbnail upload)	multipart/form-data: course fields + `thumbnail` file
PATCH	`/api/courses/:id`	Auth + Instructor/Admin	Update a course (supports thumbnail upload)	multipart/form-data: fields + optional `thumbnail`
DELETE	`/api/courses/:id`	Auth + Instructor/Admin	Delete a course	—

Method	Path	Auth	Description	Request
POST	`/api/courses/:courseId/comments`	Auth required	Post a comment on a course	JSON: `{ text }`

Lessons

Note: All lesson routes are nested under a course (`/api/courses/:courseId/lessons`) to maintain course context.

Method	Path	Auth	Description	Request
POST	`/api/courses/:courseId/lessons`	Auth + Instructor/Admin	Create a lesson (supports `video` and `attachments` file fields)	multipart/form-data: lesson fields, `video`, `attachments[]`
PATCH	`/api/courses/:courseId/lessons/:lessonId`	Auth + Instructor/Admin	Partially update a lesson	multipart/form-data: fields + files
PUT	`/api/courses/:courseId/lessons/:lessonId`	Auth + Instructor/Admin	Replace/update a lesson	multipart/form-data: fields + files
POST	`/api/courses/:courseId/lessons/:lessonId/complete`	Auth + Student/Admin	Mark lesson as complete for the current student	—
DELETE	`/api/courses/:courseId/lessons/:lessonId`	Auth + Instructor/Admin	Delete a lesson	—
GET	`/api/courses/:courseId/lessons/:lessonId`	Public	Get a single lesson by id	—
GET	`/api/courses/:courseId/lessons`	Public	List lessons for a course	query: pagination

Enrollments

Method	Path	Auth	Description	Request
GET	`/api/enrollments`	Auth required	List enrollments for the logged-in user	query: optional filters
POST	`/api/enrollments`	Auth required	Enroll current user in a course	JSON: `{ course_id, payment? }`
PATCH	`/api/enrollments/:id`	Auth required	Update enrollment (e.g., progress)	JSON: fields to update
DELETE	`/api/enrollments/:id`	Auth required	Delete/cancel enrollment	—

Instructor

Method	Path	Auth	Description	Request
GET	`/api/instructor/insights`	Auth + Instructor	Get instructor insights (analytics)	query: optional params

Method	Path	Auth	Description	Request
GET	`/api/instructors/:id/courses`	Public	List courses for a specific instructor (supports pagination)	query: `page`, `limit`
POST	`/api/instructor/ai-quiz`	Auth + Instructor	Generate an AI-generated quiz for content	JSON: quiz seed data

Quiz

Method	Path	Auth	Description	Request
GET	`/api/quiz/:quizId`	Auth + Admin/Instructor/Student	Get quiz by id	—
GET	`/api/quiz/:quizId/history`	Auth + Admin/Instructor/Student	Get quiz attempt history	query: pagination
POST	`/api/quiz/:quizId`	Auth + Admin/Instructor/Student	Submit answers for a quiz attempt	JSON: answers payload
GET	`/api/quiz`	Auth + Admin/Instructor	List all quizzes	query: `page`, `limit`
GET	`/api/quiz/instructor/:instructorId`	Auth + Admin/Instructor	List quizzes for an instructor	query params
POST	`/api/quiz`	Auth + Admin/Instructor	Create a quiz	JSON: quiz payload
PUT	`/api/quiz/:id`	Auth + Admin/Instructor	Update a quiz	JSON: updated quiz payload
DELETE	`/api/quiz/:id`	Auth + Admin/Instructor	Delete a quiz	—

Payments

Method	Path	Auth	Description	Request
POST	`/api/payments/mock`	Auth + Student/Instructor/Admin	Confirm a mock payment (development)	JSON: payment details

Code Execution

Method	Path	Auth	Description	Request
POST	`/api/code/runCodeSandbox`	Public	Run code in a sandboxed environment and return output	JSON: `{ code, language, stdin? }`

Admin

All admin endpoints require authentication and the `ADMIN` role.

Method	Path	Auth	Description	Request
GET	`/api/admin/admin-data`	Auth + Admin	Admin dashboard data	query params
GET	`/api/admin/current-user`	Auth + Admin	Get current user info (admin context)	—
GET	`/api/admin/users`	Auth + Admin	List all users	query: pagination, filters

Method	Path	Auth	Description	Request
GET	`/api/admin/users/role/:role`	Auth + Admin	List users filtered by role	path param `role`
GET	`/api/admin/users/search`	Auth + Admin	Search users (query)	query: `q`
PATCH	`/api/admin/users/:userId/role`	Auth + Admin	Update a user's role	JSON: `{ role }`
PATCH	`/api/admin/users/:userId/budget`	Auth + Admin	Update user budget	JSON: `{ budget }`
PATCH	`/api/admin/users/:userId/lock`	Auth + Admin	Toggle user locked state	JSON: `{ locked: boolean }`
GET	`/api/admin/courses`	Auth + Admin	Get courses by status (default: pending)	query: `status`
PATCH	`/api/admin/courses/:id/status`	Auth + Admin	Update course status (approve/reject/hide)	JSON: `{ status }`
PATCH	`/api/admin/courses/:id/price`	Auth + Admin	Update course price and premium flag	JSON: `{ price, premium }`
GET	`/api/admin/stats`	Auth + Admin	System statistics and metrics	—

5. Database Design

This project uses MongoDB (via Mongoose). Below is a concise summary of the main collections, key fields, relationships, indexes, and cascade behaviors found in `server/src/models`.

Collections & Schemas

- **Users (`users`)**
  - Key fields: `username` (string, unique, sparse), `fullName`, `email` (unique), `password` (hashed), `role` (enum), `interests` (string[]), `budget` (number), `locked` (boolean)
  - Indexes: **text** index on `username`, `email`, `fullName` for search; unique indexes on `email` and sparse `username`.
  - Notes: Uses `mongoose-paginate-v2` for paging.
- **Courses (`courses`)**
  - Key fields: `courseId` (string, unique), `title`, `thumbnail`, `category`, `tags` (string[]), `instructor` (ObjectId → `User`), `shortDescription`, `is_premium`, `price`, `status`, `reviewNote`, `reviewedBy` (ObjectId), `reviewedAt`.
  - Indexes: **text** index on `title`, `shortDescription`, `tags`.
  - Cascade: Deleting a course cascades to related `Lesson`, `Comment`, `Enrollment`, and `Quiz` documents.
  - Notes: Uses `mongoose-paginate-v2` for paging.
- **Lessons (`lessons`)**
  - Key fields: `id` (string, unique), `course_id` (string — references `Course.courseId`), `title`, `content` (HTML), `video` (URL), `duration`, `order`, `attachments` (string[]), timestamps.
  - Notes: Routes available both nested under `/api/courses/:courseId/lessons` and `/api/lessons` where applicable.
- **Enrollments (`enrollments`)**
  - Key fields: `studentId` (ObjectId → `User`), `courseId` (string → `Course.courseId`), `status`, `progress`, `lastLessonId`, `completed`, `completedLessons`.

    - Notes: Stores student reference as ObjectId and course reference as string (matching `Course.courseId`).

- **Comments (`comments`)**
    - Key fields: `courseId` (string → `Course.courseId`), `userId` (ObjectId → `User`), `userName`, `rating`, `content`, timestamps.

- **Quizzes (`quizzes`)**
    - Key fields: `id` (string, unique), `lesson_id` (string, optional), `course_id` (string → `Course.courseId`), `title`, `description`, `timelimit`, `questions` (array of question objects), `order`, timestamps.
    - Cascade: Deleting a quiz cascades to `SubmitQuiz` records.

- **SubmitQuizzes (`submitquizzes`)**
    - Key fields: `quizId` (string), `userId` (string), `answers` (array), `score`, `duration`, `submittedAt`.

Relationships

- `Course.instructor` → `User._id` (ObjectId)
- `Lesson.course_id` → `Course.courseId` (string)
- `Enrollment.studentId` → `User._id`; `Enrollment.courseId` → `Course.courseId` (string)
- `Comment.userId` → `User._id`; `Comment.courseId` → `Course.courseId`
- `Quiz.course_id` → `Course.courseId`; `Quiz.lesson_id` → `Lesson.id` (optional)

Indexes & Performance

- Text search on `User` and `Course` as noted above.
- Pagination is implemented with `mongoose-paginate-v2` on `User` and `Course`.
- Frequently-filtered fields like `status`, `courseId`, and role fields should be considered for additional indexes as usage grows.

Operational notes

- Deleting a `Course` triggers cascade deletes for related `Lesson`, `Comment`, `Enrollment`, and `Quiz` documents (handled in model hooks).
- Deleting a `Quiz` removes related `SubmitQuiz` documents.
- `courseId`, `id`, and `quiz.id` are used as human-friendly string identifiers (instead of ObjectIds) to simplify client-side lookups.
- For production readiness, consider adding explicit migration tooling (e.g., `migrate-mongo`) and monitoring for large delete operations.

---

6. UI/UX Design

Design System & Framework

Roads to Rome leverages **shadcn/ui** as its core component library, significantly accelerating the development cycle while maintaining high-quality, accessible, and consistent user interfaces. This approach provides several key advantages:

- **Rapid Prototyping**: Pre-built, customizable components reduce initial setup time from weeks to days
- **Design Consistency**: Unified component library ensures visual coherence across all pages
- **Accessibility First**: All shadcn/ui components follow WCAG guidelines out of the box
- **Type Safety**: Full TypeScript support eliminates runtime UI errors
- **Customization Flexibility**: Tailwind CSS integration allows easy theme modifications without fighting framework constraints

Component Architecture

Core UI Components:

- **Form Elements**: Input, Select, Checkbox, Radio with built-in validation states
- **Navigation**: Command palette, dropdown menus, tabs, and breadcrumbs

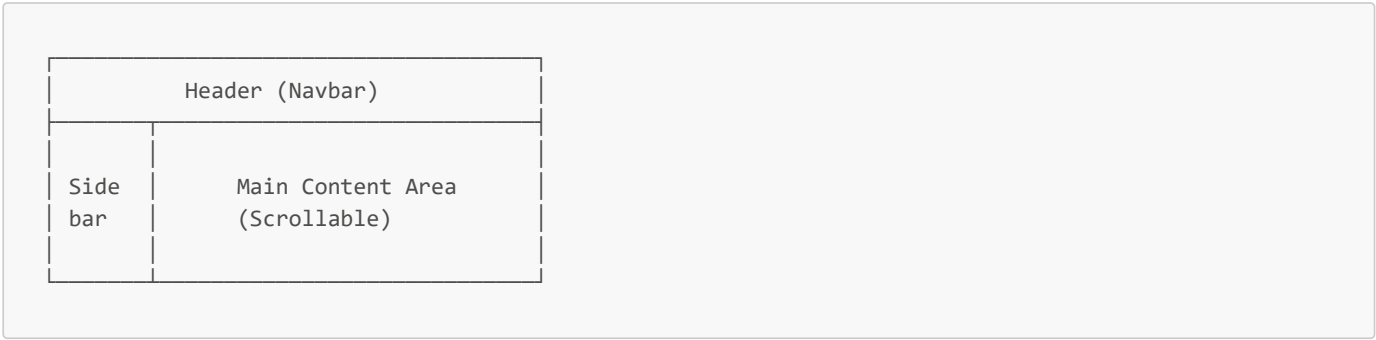
- **Feedback**: Toast notifications, dialogs, alerts, and loading skeletons
- **Data Display**: Cards, tables with sorting/filtering, badges, and avatars
- **Layout**: Responsive grid system, sidebars, and collapsible sections

Custom Components Built on shadcn/ui:

- `CourseCard`: Displays course thumbnail, title, instructor, pricing, and enrollment status
- `LessonViewer`: Rich content renderer supporting markdown, video embeds, and code blocks
- `QuizCreator`: Drag-and-drop interface for building MCQ questions
- `DashboardWidget`: Modular cards for stats, progress bars, and activity feeds
- `AdminTable`: Advanced data tables with inline editing and bulk actions

Layout Design

AppShell Structure:



- **Header**: Logo, navigation links, search bar, user profile dropdown
- **Sidebar**: Role-based navigation (collapses on mobile)
- **Content Area**: Dynamic page content with consistent padding and max-width constraints
- **Footer**: Optional contextual actions or additional navigation

Responsive Design Strategy

Breakpoints (Tailwind CSS):

- `sm`: 640px - Mobile landscape
- `md`: 768px - Tablet portrait
- `lg`: 1024px - Tablet landscape / Small desktop
- `xl`: 1280px - Desktop
- `2xl`: 1536px - Large desktop

Mobile-First Approach:

- Base styles target mobile devices
- Progressive enhancement for larger screens
- Touch-friendly hit targets (minimum 44x44px)
- Simplified navigation with hamburger menu on mobile
- Course cards stack vertically on small screens, grid layout on desktop

Color Scheme & Theming

Primary Palette:

- **Primary**: Indigo shades for CTAs, links, and focus states
- **Secondary**: Neutral grays for text and borders
- **Accent**: Purple for premium features and highlights
- **Semantic Colors**: Green (success), Red (error), Yellow (warning), Blue (info)

Dark Mode Support:

- Automatic theme detection based on system preferences
- Manual toggle in user settings
- CSS variables enable seamless theme switching without page reload

Design Tokens:

```
--primary: 239 84% 67%;
--secondary: 215 14% 34%;
--background: 0 0% 100%;
--foreground: 222 47% 11%;
--muted: 210 40% 96%;
--border: 214 32% 91%;
```

Typography

- **Headings**: Inter font family for clarity and modern aesthetic
- **Body**: System font stack for optimal readability
- **Code**: JetBrains Mono for code snippets and embedded IDE
- **Scale**: Modular scale (1.25 ratio) ensures visual hierarchy

User Experience Patterns

Loading States:

- Skeleton screens for content-heavy pages (course lists, dashboards)
- Spinner overlays for quick actions (enroll, submit quiz)
- Progress bars for file uploads and video streaming

Empty States:

- Friendly illustrations and clear CTAs when no data exists
- "Create your first course" prompts for new instructors
- "Explore courses" suggestions for students with no enrollments

Error Handling:

- Inline validation messages on form fields
- Toast notifications for async operation results
- Fallback UI for failed API requests with retry buttons

Micro-interactions:

- Button hover effects and active states
- Smooth page transitions
- Animated progress indicators
- Confetti animations on quiz completion

Accessibility Features

- **Keyboard Navigation**: Full support for tab navigation and shortcuts
- **Screen Reader Support**: ARIA labels and semantic HTML
- **Focus Management**: Visible focus indicators and logical tab order
- **Color Contrast**: WCAG AA compliant contrast ratios (minimum 4.5:1)
- **Alternative Text**: All images include descriptive alt attributes

Development Benefits

By adopting shadcn/ui, the team achieved:

- **60% reduction** in UI development time compared to building from scratch
- **Zero runtime bundle overhead** (components copied to codebase, not imported as library)
- **Consistent code patterns** across team members with varying frontend experience
- **Easy testing** with predictable component APIs and clear separation of concerns
- **Maintainability** through well-documented, single-responsibility components

This design approach allowed the team to focus on business logic and user flows rather than reinventing common UI patterns, directly contributing to the successful delivery of all planned features within the 5-week timeline.

---

7. Guideline

This section provides comprehensive instructions for setting up, running, and deploying the Roads to Rome platform.

Local Development Setup

Prerequisites

- Node.js 20 or higher
- MongoDB (local or cloud instance)
- Git

Running the Application Locally

Backend (Node.js)

Recommended for active development with hot reload:

```
cd server
npm install
npm run dev
```

Production build and start:

```
cd server
npm install --production
npm run build
npm start
```

Frontend (Vite + React)

Development mode:

```
cd client
npm install
npm run dev
```

Production build and preview:

```
cd client
npm run build
npm run preview
```

Default URLs

- Frontend → `http://localhost:5173`
- Backend → `http://localhost:3000`

Docker Deployment

Docker provides a consistent environment for development and testing. Before running, ensure `server/.env` exists with all required variables.

Production-like Container

```
cd server
docker compose -f docker-compose.yml up --build
```

Development Container (with Hot Reload + Piston)

The override file (`docker-compose.override.yml`) includes the Piston service for code execution features, supporting multiple programming languages (Python, JavaScript, etc.).

```
cd server
docker compose -f docker-compose.yml -f docker-compose.override.yml up --build
```

Managing Containers

Stop containers:

```
docker compose down
```

View logs:

```
docker compose logs -f server
```

Production Deployment

Frontend Deployment (Vercel)

The frontend is automatically deployed to Vercel via GitHub Actions when changes are pushed to the `main` or `dev` branches.

Manual Deployment:

```
cd client
npm run build
npx vercel --prod --token "$VERCEL_TOKEN" --confirm
```

Requirements:

- Vercel account and project
- `VERCEL_TOKEN` GitHub secret configured

- Environment variables set in Vercel dashboard

Backend Deployment (Render)

The backend is automatically deployed to Render via webhook when CI/CD pipeline completes successfully.

Configuration Steps:

1. Create a deploy hook under Render → *Manual Deploy → Deploy Hooks*
2. Store the URL in the `RENDER_DEPLOY_HOOK` GitHub secret
3. Configure all environment variables in Render dashboard (must match `server/.env` names)
4. Monitor deployments via Render service logs

CI/CD Pipeline

The GitHub Actions workflow (`.github/workflows/ci-cd.yaml`) runs on every `push` or `pull_request` targeting `main` or `dev`:

Jobs:

1. **`server-lint`**
   - Checks out repository
   - Sets up Node.js 20
   - Runs `npm ci` in `server/`
   - Installs ESLint + TypeScript plugins
   - Runs `npx eslint src --ext .ts || true` (lints but never fails CI)
2. **`server-deploy`**
   - Depends on: `server-lint`
   - Calls the Render deploy hook stored in `RENDER_DEPLOY_HOOK`
   - Exits with error if secret is missing
   - Allows `curl` to fail without failing CI
3. **`client-deploy`**
   - Depends on: `server-lint`
   - Installs dependencies in `client/`
   - Builds Vite with `npm run build`
   - Deploys to Vercel via `npx vercel --prod --token "$VERCEL_TOKEN" --confirm`
   - Requires `VERCEL_TOKEN` GitHub secret

Optional (commented out): Client lint/testing, server unit tests — enable these when stable.

Environment Configuration

Required Environment Variables

Create `server/.env` with the following variables:

Core Configuration

- `NODE_ENV` - Environment mode (development or production)
- `PORT` - Server port (default: 3000)
- `CLIENT_URL` - Frontend URL for CORS configuration

Database

- `MONGODB_URI` - MongoDB connection string

Authentication & Security

- `ACCESS_TOKEN_SECRET` - Secret for signing access tokens
- `REFRESH_TOKEN_SECRET` - Secret for signing refresh tokens
- `RESET_PASSWORD_SECRET` - Secret for password reset tokens
- `SESSION_SECRET` - Secret for session management

OAuth (GitHub)

- `GITHUB_CLIENT_ID` - GitHub OAuth application client ID
- `GITHUB_CLIENT_SECRET` - GitHub OAuth application secret

File Storage (Supabase)

- `SUPABASE_URL` - Supabase project URL
- `SUPABASE_SERVICE_KEY` - Supabase service role key for file uploads

Code Execution (Piston)

- `PISTON_URL` - Piston API endpoint for code execution

AI Integration (Google Gemini)

- `GEMINI_API_KEY` - Google Gemini API key for AI quiz generation
- `GEMINI_MODEL_NAME` - (Optional) Gemini model version

Email Service

- `GMAIL_USER` - Gmail account for sending emails
- `GMAIL_APP_PASSWORD` - Gmail app-specific password

Email Configuration Notes:

- **Development**: Uses Ethereal automatically (no setup required)
- **Production**: Requires a real SMTP provider (SendGrid, Mailgun, SES, or Gmail SMTP)

External Service Setup

GitHub OAuth Application

1. Register a new OAuth app at GitHub Settings → Developer settings → OAuth Apps
2. Set callback URLs:
    - Local: `http://localhost:3000/api/auth/github/callback`
    - Production: `https://your-backend-domain.com/api/auth/github/callback`
3. Copy Client ID and Client Secret to environment variables

Supabase Storage

1. Create a Supabase project
2. Enable Storage and create a public bucket for course thumbnails and videos
3. Copy project URL and service role key to environment variables

Piston API

- Use public Piston instance: `https://emkc.org/api/v2/piston`
- Or self-host Piston using Docker (included in `docker-compose.override.yml`)

Google Gemini API

1. Visit Google AI Studio
2. Create an API key
3. Add to environment variables

Authentication & Authorization

Access Token (JWT)

- **Lifetime**: ~1 hour
- **Signed with**: `ACCESS_TOKEN_SECRET`
- **Delivered**: JSON response
- **Storage**: Client memory (not localStorage for security)

Refresh Token (JWT)

- **Lifetime**: ~7 days
- **Signed with**: `REFRESH_TOKEN_SECRET`
- **Delivered**: `HttpOnly`, `SameSite=strict`, `secure` cookie
- **Security**: Not accessible to JavaScript

Token Refresh Flow

1. Client calls `/api/auth/refresh-token`
2. Server reads refresh token from cookie
3. Server validates and returns new access token

Role-Based Authorization

Roles:

- `GUEST` - Unauthenticated users (browse only)
- `STUDENT` - Enrolled learners
- `INSTRUCTOR` - Course creators
- `ADMIN` - Platform moderators

Implementation:

- Middleware: `authenticateToken`, `authorizeRoles([...])`
- Protected routes enforce role requirements
- UI elements conditionally rendered based on user role

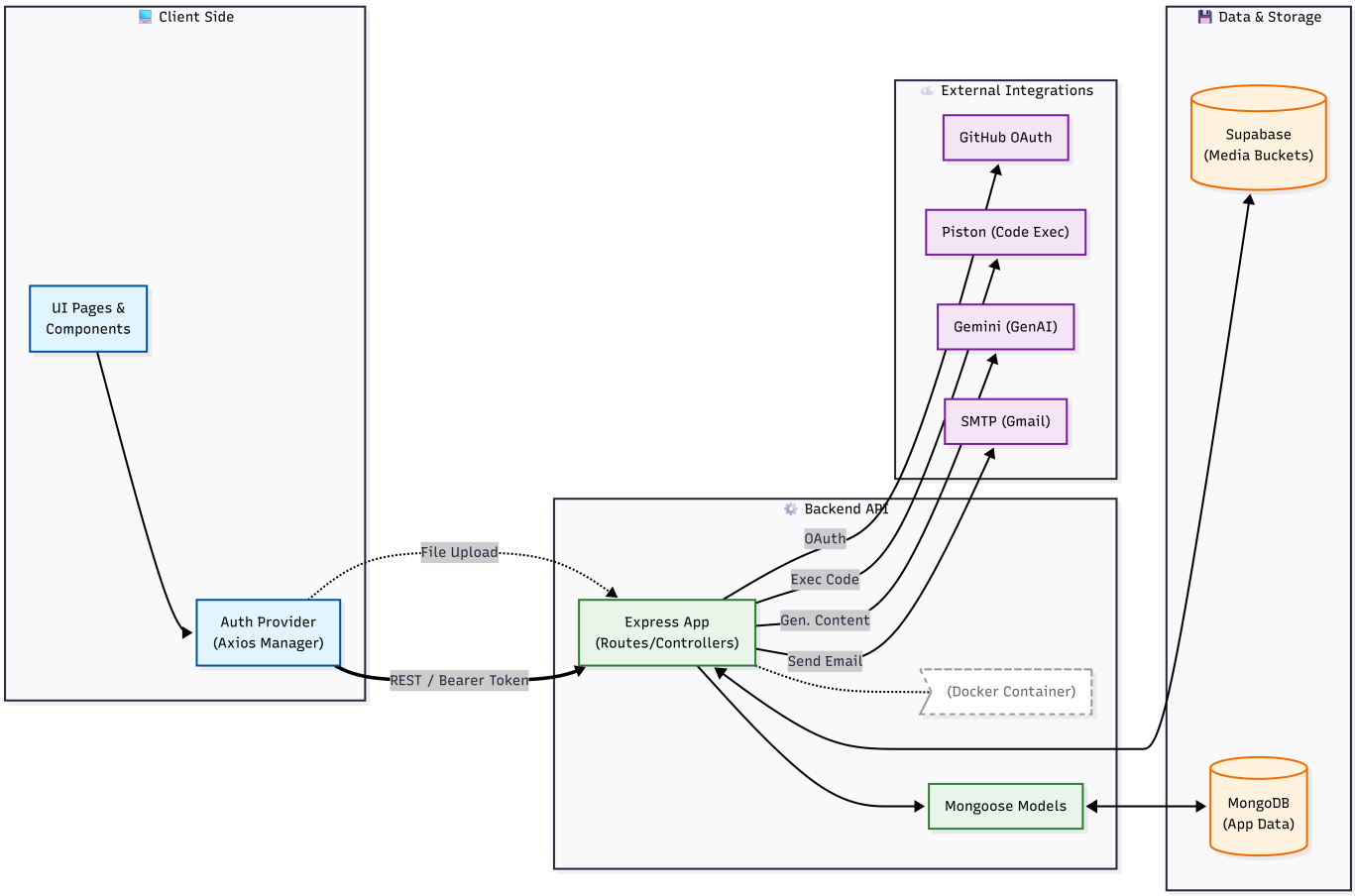
OAuth (GitHub)

- Automatically creates new users with `STUDENT` role
- Generates fallback password and sends via email
- Maps GitHub profile to user account

Key Files:

- `server/src/middlewares/auth.middleware.ts`
- `server/src/services/auth.service.ts`
- `server/src/enums/user.enum.ts`

Architecture Design



Decisions & Tradeoffs

- **HttpOnly refresh cookie + short-lived access token**
  - Better XSS protection
  - Simplifies client-side token handling
  - Requires CSRF considerations (mitigated with `SameSite=strict`)
- **Stateless JWTs**
  - No session store required
  - Easy to scale horizontally
  - Harder to force immediate revocation
- **No refresh token rotation**
  - Simpler
  - Less secure if refresh token is leaked
- **OAuth user creation**
  - Auto-creating accounts makes onboarding straightforward
  - Could be surprising; alternative is requiring explicit setup
- **Email handling**
  - Ethereal for dev to prevent real email sends
  - Real SMTP required for production