INTRODUCTION

Live - https://prolearn-lms.vercel.app/

Github - https://github.com/md-danishraza?tab=repositories&q=LMS

PROBLEM STATEMENT

The current e-learning landscape has low completion rates. Students often abandon courses out of frustration when they encounter problems they cannot solve alone. The lack of direct access to instructors creates a disconnect, making it difficult to apply theoretical knowledge to real-world scenarios.

SOLUTION : PROLEARN

ProLearn aims to solve this by building a system where:

- Students can learn at their own pace with high-quality courses.
- Students can directly book and attend live video sessions with their instructors for personalized help, code reviews, or career advice.
- Instructors can offer tangible, premium support, creating a new value proposition.

Online learning has revolutionized education, with platforms like Udemy and Coursera offering vast libraries of video courses. However, this model often suffers from a significant drawback: it is a passive, one-way experience. Students are left with forums and Q&A sections when they get stuck, lacking the personalized, real-time guidance that is crucial for mastering complex skills.

ProLearn is a full-stack web application that addresses this gap. It's an E-Learning platform that not only provides a rich library of on-demand video courses but also seamlessly integrates a **1-to-1 video mentorship feature**. This "hybrid" model combines the scalability of an LMS with the effectiveness of personal tutoring, creating a more supportive and effective learning environment.

OBJECTIVES & SCOPE

The primary objectives of the ProLearn project are:

- To develop a secure, high-performance, and scalable full-stack web application.
- To implement a robust user authentication and management system using **Clerk**.
- To create a system for browsing, purchasing, and streaming video courses.
- To integrate a payment gateway (**CashFree**) to simulate course purchases.
- To develop a scheduling system for 1-to-1 mentorship sessions.
- To integrate a real-time, low-latency video call feature using the **Agora SDK**.
- To deploy on current ongoing platforms like vercel serverless and render nodejs runtime.

Scope :

- A functional MVP (Minimum Viable Product).
- Users can sign up, log in, browse, and "purchase" (in test mode) courses.
- Enrolled users can stream video content.
- Users can book a mentorship session.
- Two users (student and mentor) can join a unique, 1-to-1 video call with chat.

PROCESS DESCRIPTION

System Design

Frontend (Vercel):

- **Next.js** serves the UI.
- **Clerk** handles the authentication. The frontend receives a JWT which it attaches to headers for every request to backend.
- **Redux** caches API responses (User profile, Courses) to reduce network load.

Backend (Render):

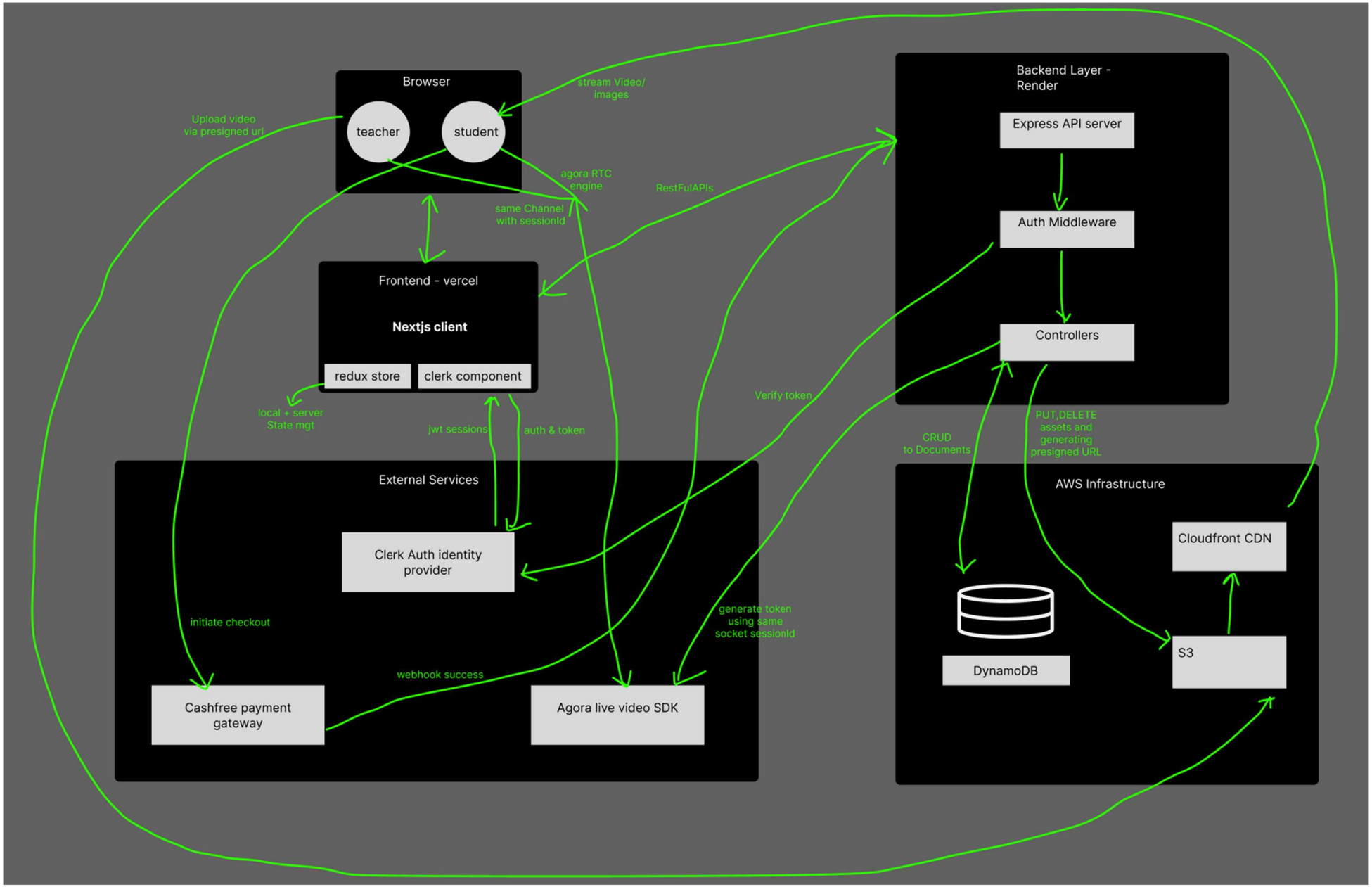
- **Express API** acts as the orchestrator. It doesn't store files; it only stores *references* (URLs) and metadata.
- **Middleware:** Intercepts requests to validate the Clerk JWT before allowing access to protected routes (like /teacher/courses).

Data Layer (AWS):

- **DynamoDB:** Stores structured data (Users, Courses, Transactions,UserCourseProgress). It uses single-table design concepts (PK/SK) for speed.
- **S3:** Acts as the "Hard Drive." It stores the raw video files and thumbnail images.
- **CloudFront:** Stream media from CloudFront edge locations for low latency.

Payments (Cashfree):

- The frontend initiates the payment.
- Crucially, the **Backend** verifies the success via a **Webhook**. This ensures a user can't just "fake" a frontend success message to get a course for free.


Browser
teacher
student
stream Video/
images
Upload video
via presigned url
agora RTC
engine
same Channel
with sessionId
RestFulAPIs
Backend Layer -
Render
Express API server
Auth Middleware
Controllers
Frontend - vercel
Nextjs client
redux store
clerk component
local + server
State mgt
jwt sessions
auth & token
Verify token
CRUD
to Documents
PUT,DELETE
assets and
generating
presigned URL
External Services
Clerk Auth identity
provider
AWS Infrastructure
Cloudfront CDN
DynamoDB
S3
initiate checkout
generate token
using same
socket sessionId
webhook success
Cashfree payment
gateway
Agora live video SDK

CLERK AUTHENTICATION

User Authentication & Role-Based Workflow

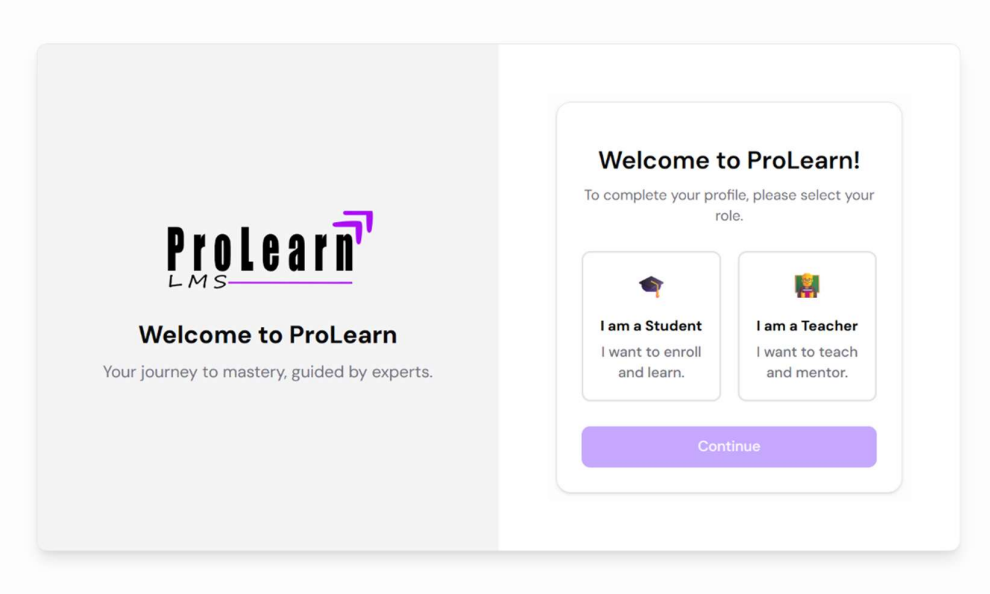
The primary challenge is to securely manage two distinct user types—**Students** and **Teachers**—and provide a seamless onboarding experience that directs them to the correct dashboard, all while handling special cases like in-context checkout.

This system is built using **Clerk** for core identity management, **Next.js Middleware** for route protection, and **Next.js Server Actions** for secure, server-side data mutation.

Core Technical Components

1. **Clerk (Identity Provider):** Handles all core authentication (sign-in, sign-up, session management).
2. **Clerk JWT Templates:** The Clerk JWT (session token) is customized to include a **userType** claim. This is the "source of truth" for a user's role and is available instantly to the middleware on every request, which is critical for performance and security.
3. **Next.js Middleware (middleware.ts):** Acts as the primary "gatekeeper" for the entire application. It intercepts all requests to protect routes and enforce role-based access.
4. **Onboarding Page (/onboarding):** A dedicated, mandatory page for all new users (except those in the checkout flow) to select their role.
5. **Server Action (updateUserRole):** A secure, server-side function that a user calls from the onboarding page. This is the **only** mechanism that can write to a user's publicMetadata, preventing users from changing their own roles from the browser console.
6. **Custom Components (SignInComp, SignUpComp):** These components adapt the authentication flow for different contexts, specifically handling redirects for the checkout page versus standard site navigation.

User Workflow 1: New User (Standard Onboarding)



This is the flow for a new user who signs up from the main "Sign Up" button.

1. **Sign Up:** The user signs up via SignUpComp. isCheckoutPage is false, so the component's getRedirectUrl() function returns '/onboarding'.
2. **Redirect to Onboarding:** After successful sign-up, Clerk redirects the user to /onboarding.
3. **Middleware (Check 1):** The middleware intercepts this. It sees:
    - userId exists.
    - userRole is undefined (the JWT template is empty as metadata was never set).
    - The user *is* on the onboarding page (isOnboardingRoute(req) is true).
    - The request is allowed to proceed, and the user sees the role-selection page.
4. **Role Selection:** The user clicks "I am a Student" and "Submit."
5. **Server Action:** The handleRoleSubmit function calls the updateUserRole server action. This action securely updates the user's publicMetadata in Clerk's database.
6. **Session & Redirect:**
    - The action returns { success: true }.
    - The client calls await session.reload(). This is **critical**. It forces Clerk to issue a *new* JWT, which now includes the userType: 'student' claim from the updated JWT template.
    - router.push('/user/courses') is called.
7. **Middleware (Check 2):** The middleware intercepts the *new* request to /user/courses. It sees:
    - userId exists.
    - userRole is now 'student'.
    - The route is a student route (isStudentRoute(req) is true).

- The user is granted access, and the student dashboard loads successfully.

User Workflow 2: New User (Checkout Flow)

This flow is designed to minimize friction during a purchase.

1. **Sign Up:** The user is on /checkout and uses the SignUpComp modal.
2. **Redirect to Checkout:** isCheckoutPage is true, so getRedirectUrl() returns /checkout?step=2&id=.... The user completes their purchase.
3. **Post-Purchase Navigation:** The user (who is logged in but has no role) clicks a link to their dashboard (e.g., /user/courses).
4. **Middleware (Check 3):** The middleware intercepts this. It sees:
   - userId exists.
   - userRole is undefined.
   - The route is *not* the onboarding page (!isOnboardingRoute(req) is true).
5. **Forced Onboarding:** The middleware's primary rule fires, redirecting the user to /onboarding to complete their profile setup. From here, the flow continues identically to **Workflow 1 (Step 5)**.

User Workflow 3: Returning User (Sign In)

This flow demonstrates the power of the middleware and JWT claims.

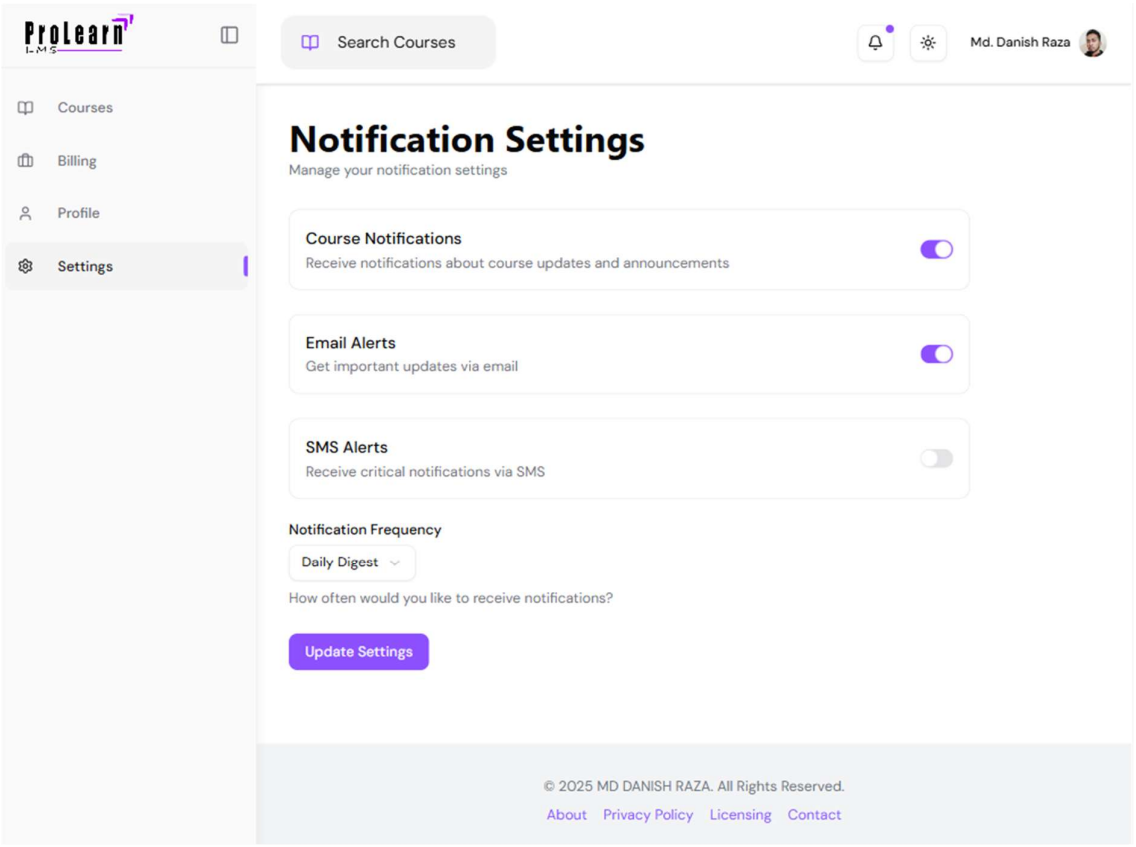
1. **Sign In:** A returning user logs in via SignInComp.
2. **Redirect to Home:** isCheckoutPage is false, so getRedirectUrl() returns /.
3. **Middleware (Check 4):** The middleware intercepts the request to /. It sees:
   - userId exists.
   - userRole is 'teacher' (read instantly from the JWT).
4. **Role-Based Redirect:** The middleware's logic (if (userId && userRole)) fires. It sees the user is on the homepage (/) but has a role, and it redirects them to their correct dashboard at /teacher/courses.

UI & Profile Integration

This role-based logic extends to the UI:

- **UserButton:** The UserButton in the main navbar dynamically sets its userProfileUrl prop by reading the same userRole variable, ensuring teachers are sent to /teacher/profile and students to /user/profile.

User settings Preferences



I am saving user(teacher/student) setting preferences in clerk user metadata also
JWT auth token will sent on each request to server and server will use useAuth() clerk middleware to authenticate and allow user
So only authorized user can update the settings

1. Frontend (Client-Side)
    1. Form Rendering (Shadcn/ui): The user navigates to the settings page, which renders a form built with shadcn/ui components (<Form>, <Input>, <Switch>, etc.).
    2. Schema Definition (Zod): A zod schema defines the shape, data types, and validation rules for the settings (e.g., bio: z.string().max(200)).
    3. State Management (react-hook-form): The useForm hook is initialized with zodResolver, linking the Zod schema to the form. It manages the form's state, user input, and validation errors.
    4. Submission: On clicking "Save," react-hook-form validates the data. If successful, its onSubmit handler is called.
    5. API Call (RTK Query): The onSubmit handler calls the updateUser mutation hook (from api.ts). RTK Query constructs a PUT request to your backend API endpoint (e.g., /api/user/clerk/:userId) and sends the validated form data as the request body.

---

2. Backend (Server-Side)

6. Route Handling (Node.js/Express): Node.js server receives the PUT request. The Express router matches the endpoint and passes the request to the updateUser controller.
7. Data Persistence (Clerk Controller): The controller extracts the userId from req.params and the userData from req.body. It then calls the clerkClient.users.updateUserMetadata() function, passing the userId and the userData object to be saved in Clerk.
8. API Response: Clerk's API updates the user's metadata. Your controller then sends a 200 OK JSON response (e.g., { message: "User updated successfully!" }) back to the frontend.

Checkout Flow

Step 1: Details & Auth (Client-Side)

1. A user (either signed-in or out) lands on the /checkout?step=1 page.
2. The page displays a 2-column grid:
    - **Left Column:** Shows the course details and order summary (using **useCurrentCourse**).
    - **Right Column (Conditional):**
        - If !isSignedIn, it shows the <SignInComp /> or <SignUpComp />.
        - If isSignedIn, it shows the signed-in user's details and a "Continue to Payment" button.
3. A signed-out user signs in. The component re-renders, and the right column now shows the "Continue to Payment" button.
4. The user clicks "Continue to Payment," which calls **MapsToStep(2).**

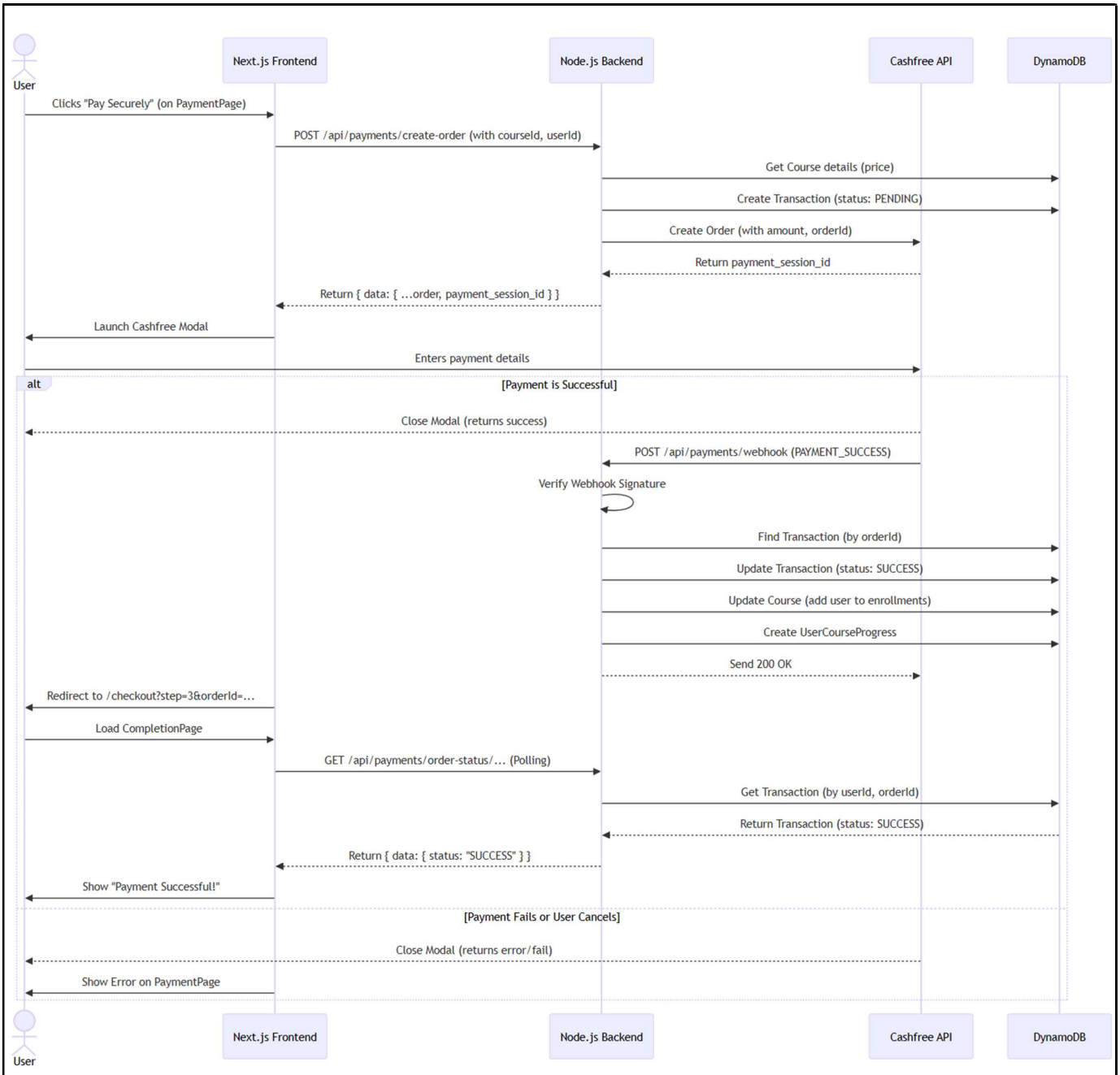
Step 2: Payment (Full-Stack)

5. The user is redirected to /checkout?step=2 (PaymentPage).
6. The PaymentPage component loads the Cashfree SDK using the load() function.
7. The user clicks "Pay Securely." The handlePayment function fires.
8. **[Backend Call 1]** The useCreateOrderMutation hook is triggered, sending a POST request to backend (/api/payments/create-order) with the **courseId** and **userId**.
9. **[Backend Logic 1]** Node.js controller receives the request:
    - It validates the user and course data.
    - It creates a new Transaction in DynamoDB with a status: 'PENDING'.
    - It calls the Cashfree API to generate an order and receives a **payment_session_id**.
    - It returns a JSON response: { message: "...", data: { ...cashfree_order_data... } }.

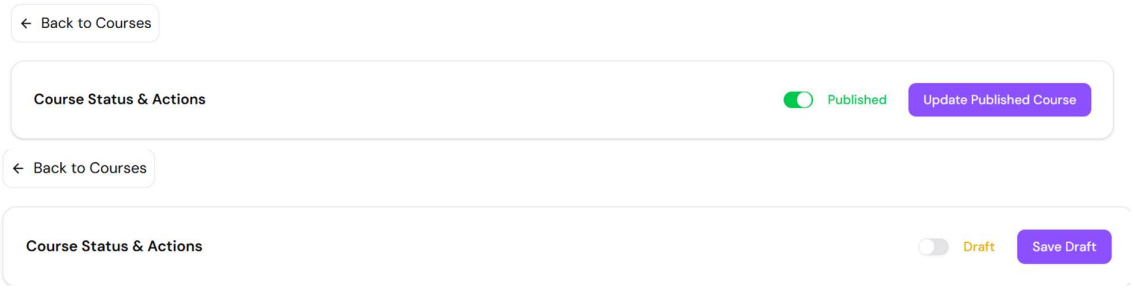
10. **[Frontend Action]** The handlePayment function receives the payment_session_id. It then launches the Cashfree payment modal using cashfree.checkout().
11. The user completes the payment in the modal.

Step 3: Completion & Fulfillment (Full-Stack)

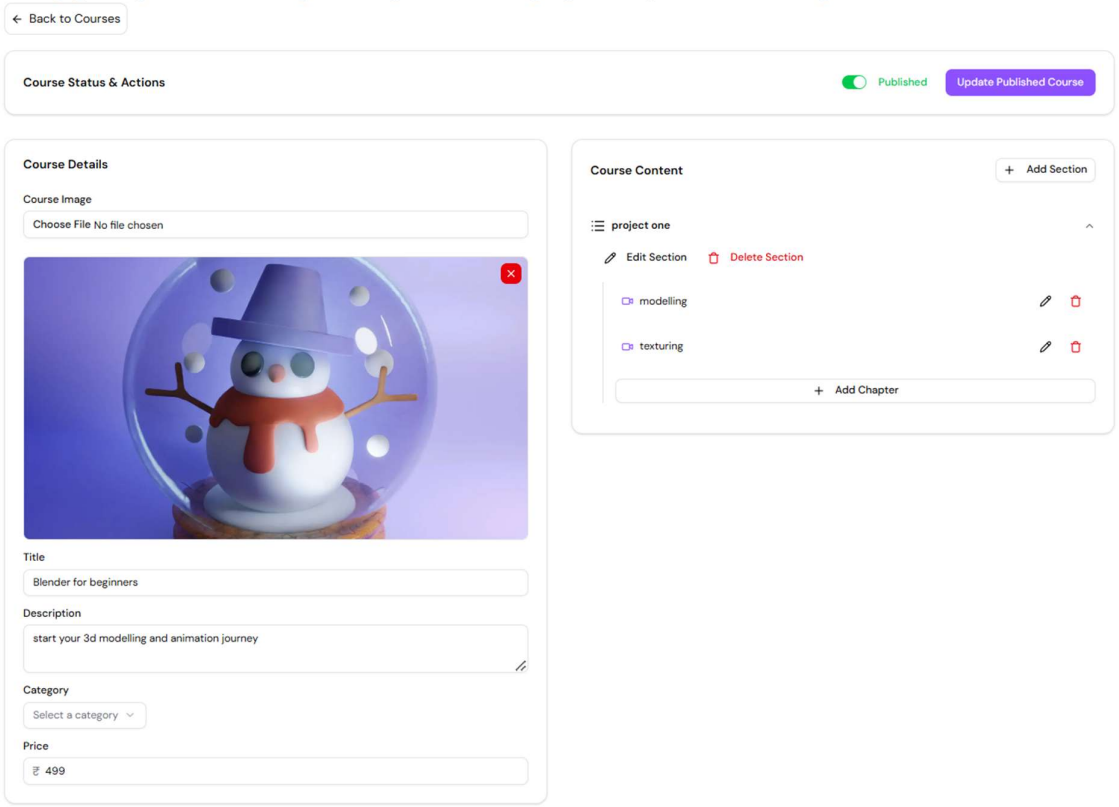
12. **[Webhook]** *Simultaneously*, the Cashfree server sends a PAYMENT_SUCCESS webhook (a POST request) to my backend (/api/payments/webhook).
13. **[Backend Logic 2] handleWebhook** controller:
    - Verifies the webhook's signature.
    - Finds the PENDING transaction in DynamoDB.
    - Updates the transaction status to "SUCCESS".
    - **Grants Access:**
        1. Adds the userId to the Course's enrollments list.
        2. Creates a new UserCourseProgress record for that user and course.
    - Sends a 200 OK response to Cashfree.
14. **[Frontend Redirect]** After the user completes the modal, the cashfree.checkout() promise resolves. Your handlePayment function then calls MapsToStep(3), redirecting the user to /checkout?step=3&orderId=....
15. **[Frontend Polling]** The CompletionPage loads and uses the orderId to call your **useGetOrderStatusQuery** hook.
16. **[Backend Call 2]** This hook hits /api/payments/order-status/:userId/:orderId endpoint.
17. **[Backend Logic 3]** Your getOrderStatus controller performs a super-fast Transaction.get() using **the userId and orderId** primary key. It returns the (now "SUCCESS") status from the database.
18. **[Frontend Success]** The useGetOrderStatusQuery hook receives the "SUCCESS" status, and the UI updates from a "Verifying..." spinner to a "Payment Successful!" message.


User
Next.js Frontend
Node.js Backend
Cashfree API
DynamoDB
Clicks "Pay Securely" (on PaymentPage)
POST /api/payments/create-order (with courseId, userId)
Get Course details (price)
Create Transaction (status: PENDING)
Create Order (with amount, orderId)
Return payment_session_id
Return { data: { ...order, payment_session_id } }
Launch Cashfree Modal
Enters payment details
alt
[Payment is Successful]
Close Modal (returns success)
POST /api/payments/webhook (PAYMENT_SUCCESS)
Verify Webhook Signature
Find Transaction (by orderId)
Update Transaction (status: SUCCESS)
Update Course (add user to enrollments)
Create UserCourseProgress
Send 200 OK
Redirect to /checkout?step=3&orderId=...
Load CompletionPage
GET /api/payments/order-status/... (Polling)
Get Transaction (by userId, orderId)
Return Transaction (status: SUCCESS)
Return { data: { status: "SUCCESS" } }
Show "Payment Successful!"
[Payment Fails or User Cancels]
Close Modal (returns error/fail)
Show Error on PaymentPage
User
Next.js Frontend
Node.js Backend
Cashfree API
DynamoDB

TEACHER – COURSE EDITOR PAGE



Using redux store for local state of editor when clicked on draft or publish it uploads all videos sequentially to S3 first and then update the database result in synced Local state + Remote State
Using presigned url for uploading video as api gateway block the request if file size>10mb



The Course Editor is a comprehensive, single-page interface designed to give instructors full control over their course structure and content. It employs a split-view layout to manage metadata and curriculum simultaneously.

1. Course Details Management (Left Panel)

- **Metadata Input:** Teachers input essential course information including the Title, Description, Category, and Price.
- **Thumbnail Upload:** The interface includes a file uploader for the course thumbnail. When a file is selected, a local preview is generated immediately for visual confirmation before the final upload to AWS S3.

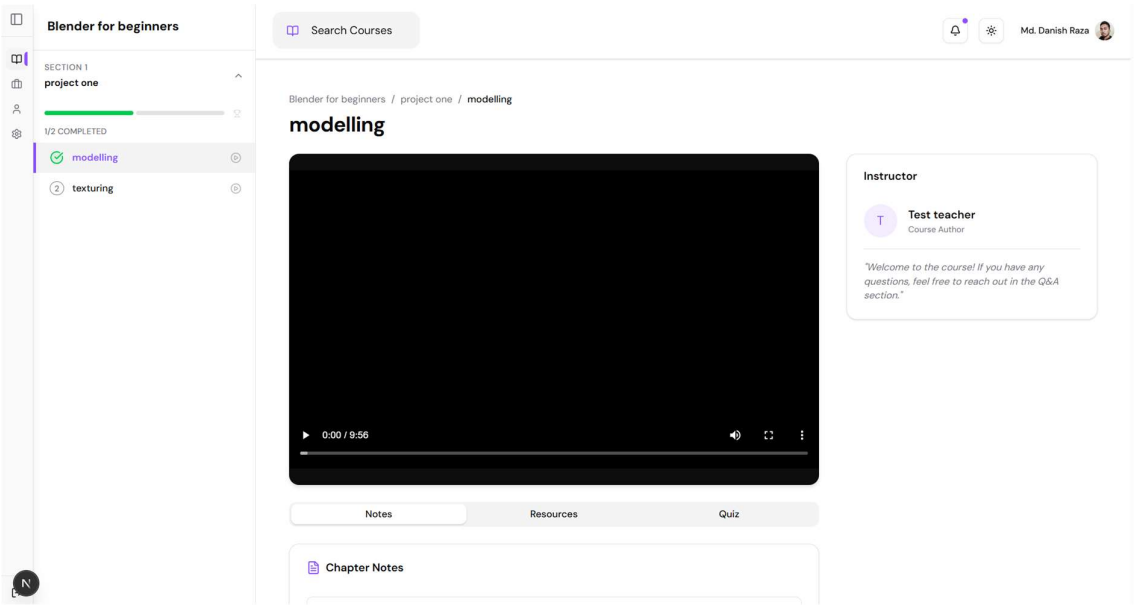
2. Curriculum Structuring (Right Panel)

- **Hierarchical Content:** The editor uses a nested list structure. Teachers can create multiple **Sections** (e.g., "Introduction", "Advanced Concepts"), and within each section, they can add multiple **Chapters** (lessons).
- **Modal-Based Editing:** Adding or editing a section or chapter opens a dedicated modal (dialog). This keeps the main interface clean while allowing for detailed input, such as uploading chapter-specific video lectures.
- **Local State Management:** The complex structure of sections and chapters is managed locally using **Redux**. This allows teachers to build out the entire course structure, reorder items, and make changes instantly without waiting for a server response after every single click.

3. Publishing & Persistence

- **Draft vs. Published:** A toggle switch in the header allows teachers to control the visibility of the course. Draft courses are saved but hidden from the student marketplace.
- **The Save Process:** When the "Update/Save" button is clicked, the application performs a robust multi-step operation:
    1. **Media Upload:** Any new video files or images are identified and uploaded directly to **AWS S3** using secure pre-signed URLs.
    2. **URL Resolution:** The S3 URLs returned from the upload are mapped back to their respective chapters.
    3. **Database Update:** The entire course object, including the new media URLs and structure, is sent to the backend to update the **DynamoDB** record.

USER – COURSE VIEW PAGE



The Student Course Interface is an immersive, distraction-free environment designed to facilitate consumption and progress tracking. It organizes complex course structures into an intuitive, navigable layout.

1. Content Consumption Area (Left Panel)

- **Video Player:** The centerpiece of the page is a responsive video player that streams high-quality educational content directly from **AWS CloudFront**. It supports standard playback controls (play/pause, seek, volume).
- **Intelligent Progress Tracking:** The player includes logic to automatically track user engagement. When a student watches **80%** of a video, the system automatically marks the chapter as "Completed," updating their progress in the background without requiring manual input.
- **Supplementary Materials (Showcase):** Below the video, a tabbed interface provides quick access to resources relevant to the specific chapter.

2. Curriculum Navigation & Progress (Right/Sidebar)

- **Structured Syllabus:** The **Chapter Sidebar** displays the full course hierarchy (Sections > Chapters). It allows students to visualize their journey through the material.
- **Visual Status Indicators:** Each chapter features dynamic icons to indicate status:
    - **Green Checkmark:** Completed chapters.
    - **Play Icon/Text Icon:** Pending chapters (differentiated by content type).
    - **Highlight:** The currently active chapter is visually highlighted for context.

Mentorship session with Live Video(agora) and Chat(socket.io)

1. Video Call Workflow (Agora)
The video system uses a **Token-Based Authentication** flow to ensure security.

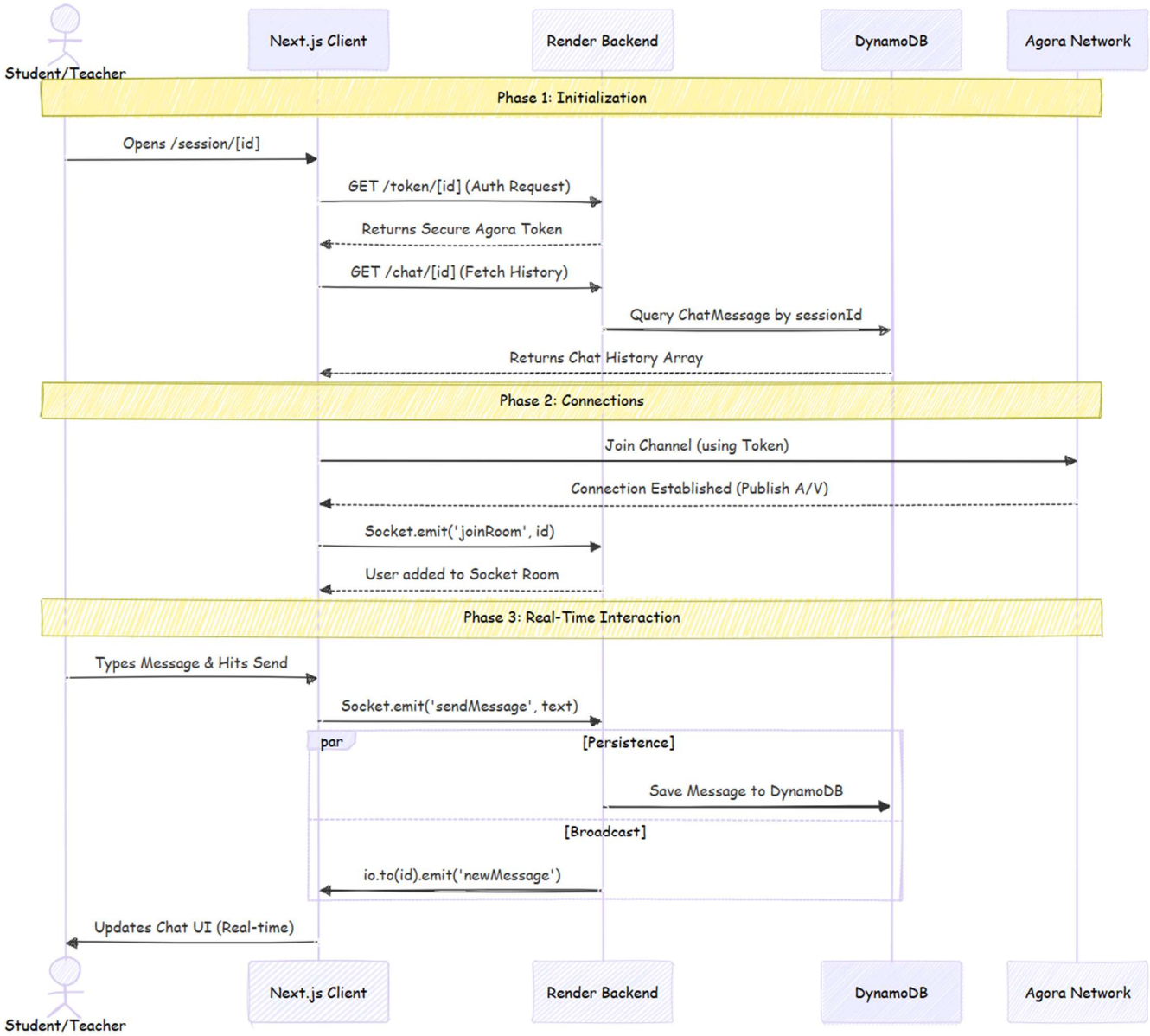
- **Initialization:** When either the Student or Teacher enters the session page, the frontend makes an API call to your Node.js backend (/api/mentorship/token/:sessionId).
- **Token Generation:** The backend uses your **Agora App Certificate** to generate a temporary, secure access token valid only for that specific sessionId.
- **Connection:** The frontend receives the token and initializes the **Agora RTC Engine**.
- **Channel Join:** Both users join the same "Channel" (named after the sessionId). Agora handles the peer-to-peer connection, NAT traversal, and bitrate adaptation.
- **Publish/Subscribe:**
    - **Local:** The browser captures the user's camera/mic and "Publishes" the stream to the channel.
    - **Remote:** The client "Subscribes" to the other user's stream and renders it in the video container.

2. Real-Time Chat Workflow (Socket.io)

The chat system runs parallel to the video but is handled entirely by **Render Backend**.

- **Room Assignment:** On load, the client connects to the WebSocket server and emits a joinRoom event with the sessionId. This isolates their conversation from other users.
- **Message History:** The frontend simultaneously calls an API endpoint to fetch previous chat logs from **DynamoDB**, ensuring context is never lost.
- **Sending Messages:** When a user types a message, it is emitted to the server via sendMessage.
- **Persistence & Broadcast:**
    1. The Server intercepts the message and **saves it to DynamoDB** (using the ChatMessage model).
    2. The Server **broadcasts** the message immediately to the specific room (io.to(sessionId).emit(...)).
- **Updates:** The receiving client's state updates instantly, scrolling the chat window to show the new message.

```
Directory structure
app/
├── user/
│   └── sessions/
│       ├── page.tsx          <-- List of upcoming/past sessions
│       └── [sessionId]/
│           └── page.tsx      <-- The Video/Chat Room
├── teacher/
│   └── sessions/
│       ├── page.tsx          <-- List of requested/upcoming sessions
│       └── [sessionId]/
│           └── page.tsx      <-- The Video/Chat Room
```



DATABASE & ENTITY RELATIONSHIPS

Logical data model for the **ProLearn** application, designed around a NoSQL (DynamoDB) architecture. The model is centered on four primary entities: **User, Course, Transaction, UserCourseProgress, MentorshipSession and ChatMessage**.

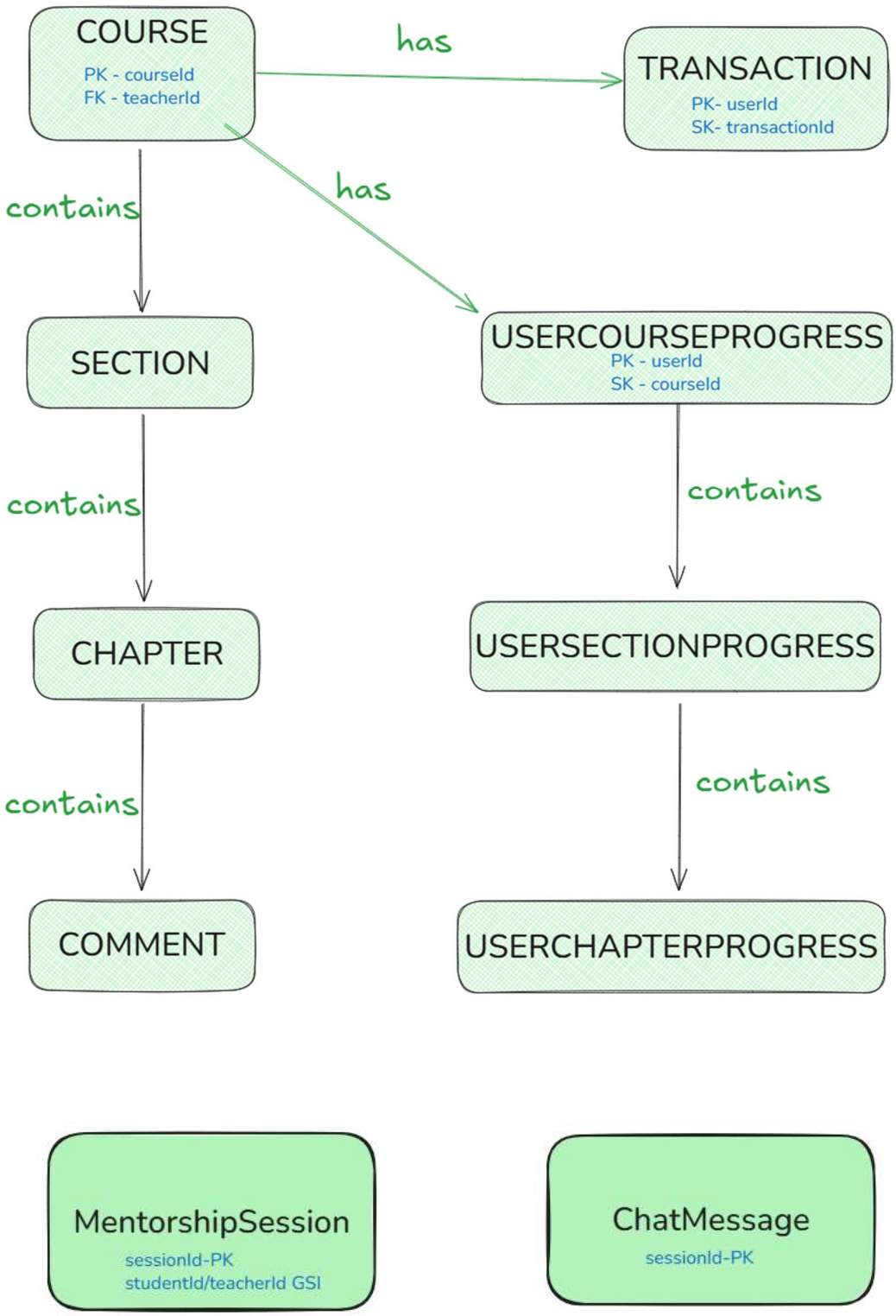
- **Core Relationship**: A User (identified by a Clerk ID) can have a "Teacher" role, allowing them to create multiple Courses. A User with a "Student" role initiates a Transaction to purchase a Course.
- **Enrollment Logic**: Upon a successful Transaction, a UserCourseProgress item is created. This entity is the cornerstone of the student experience, linking a specific User to a specific Course to track their progress.
- **NoSQL Design**: The diagram highlights DynamoDB's single-table design principles through composite primary keys. Both Transaction and UserCourseProgress use a [userId, courseId] composite key (PK), allowing for highly efficient, single-query lookups (e.g., "find all courses for a user" or "find all users for a course").
- **Data Composition**: The Course entity demonstrates **embedded data**, or composition. It directly contains its content, which is structured as a list of Sections, which in turn contain Chapters. Finally, Chapters can contain a list of Comments, which are linked back to the User who wrote them.

MentorshipSession Model This entity manages the lifecycle of a 1-on-1 video appointment.

- **Primary Key:** sessionId (Unique identifier for the meeting).
- **Indexing:** It utilizes **Global Secondary Indexes (GSIs)** on studentId and teacherId. This NoSQL pattern allows the application to efficiently query "All sessions for a specific Student" or "All sessions for a specific Teacher" without scanning the entire database.
- **Attributes:** Stores the status (scheduled/completed/cancelled), date, and the associated courseId.

ChatMessage Model This entity stores the persistent chat history for every session.

- **Composite Primary Key:** It uses a composite key design for optimized retrieval:
  - **Partition Key (PK):** sessionId (Groups all messages belonging to one specific meeting together).
  - **Sort Key (SK):** timestamp (Automatically orders messages chronologically).
- **Efficiency:** This design allows the frontend to fetch the entire chat history for a session in a single, low-latency database query.


COURSE
PK - courseId
FK - teacherId
has
TRANSACTION
PK- userId
SK- transactionId
contains
has
SECTION
USERCOURSEPROGRESS
PK - userId
SK - courseId
contains
contains
CHAPTER
USERSECTIONPROGRESS
contains
contains
COMMENT
USERCHAPTERPROGRESS
MentorshipSession
sessionId-PK
studentId/teacherId GSI
ChatMessage
sessionId-PK

TOOLS AND TECHNOLOGY USED

This project will be built using a modern, decoupled architecture:

- **Frontend:** Next.js (App Router), React, Tailwind CSS, Shadcn/ui
- **Backend:** Node.js (with TypeScript), Express.js
- **Database:** AWS DynamoDB (NoSQL)
- **Authentication:** Clerk
- **Payments:** CashFree (Test Mode)
- **Video Sessions:** Agora SDK
- **Deployment & Infrastructure:**
    - **Frontend:** Vercel
    - **Backend:** Render (Nodejs Runtime)
    - **Storage:** AWS S3 (for course videos and assets)
    - **CDN:** AWS CloudFront (for low-latency content delivery)

System architecture and modules:

1. **User & Auth Module (Clerk):** Handles user registration, login, profile management, and role-based access (Student vs. Mentor).
2. **Course Module:** Allows users to browse course listings, view details, and enroll.
3. **Payment Module (CashFree):** Manages the checkout and payment process, unlocking course access upon successful transaction.
4. **Streaming Module (S3 + CloudFront):** Securely stores video files in an S3 bucket and delivers them globally with low latency via CloudFront.
5. **Mentorship Module (Agora):** The core feature. This module includes:
    a. A booking interface for students to schedule sessions.
    b. A dashboard for mentors to manage their availability.
    c. A secure video call interface, powered by Agora, that generates temporary tokens for users to join a private, 1-to-1 video room.
6. **Backend API (Node.js/Express on Render):** A scalable RESTful API built with Node.js and Express.js, deployed and hosted on Render. It acts as the central orchestrator for the application, handling all business logic, interacting with the AWS DynamoDB database for persistence, managing secure video uploads to S3, and coordinating payment sessions via Cashfree.

Server Security Measures

The backend architecture incorporates a multi-layered security strategy to protect API endpoints and real-time communications:

- **Rate Limiting:** Implemented express-rate-limit to restrict the number of requests from a single IP address, mitigating the risk of Brute Force and DDoS attacks.
- **WebSocket Authentication:** Real-time connections via Socket.io are secured using a custom middleware that validates the **Clerk JWT (JSON Web Token)** during the handshake. Connections without a valid token are rejected immediately.
- **Identity Verification:** To prevent impersonation in chat, the sender's identity (userId) is derived server-side from the verified token rather than trusting client-provided data.
- **Header Hardening:** Utilizes helmet to automatically set secure HTTP headers (protecting against XSS and clickjacking) and enforces strict **CORS** (Cross-Origin Resource Sharing) policies to allow requests only from the trusted frontend domain.

Contact Page

The Contact page serves as the primary communication channel for users to reach support, report technical issues, or inquire about courses. It is built to be secure, user-friendly, and reliable.

- Frontend Interface: A responsive form built with React Hook Form and Shadcn/ui components. It features real-time client-side validation using Zod schemas to ensure data integrity (e.g., verifying email formats and enforcing minimum message lengths) before submission.
- Backend Processing: Upon submission, data is sent to a secure Node.js/Express API route. The backend performs a second layer of validation using express-validator to sanitize inputs.
- Email Delivery: The application integrates with the Resend API, a modern transactional email service. This ensures high deliverability rates, routing user inquiries directly to the administrator's inbox while maintaining logs of all communication attempts.

CONCLUSION AND LIMITATIONS

ProLearn is a modern solution to a persistent problem in online education. By integrating direct mentorship, it creates a more engaging and effective learning path.

Future Scope for this project includes:

- **Advanced Mentorship:** Implementing **Group Sessions** and **Live Webinars** to allow instructors to teach multiple students simultaneously.
- **AI Integration:** Utilizing AI to generate personalized learning paths, quiz questions, and automated code reviews based on course content.
- **Community Features:** Building a discussion forum and a peer-to-peer review system to foster a learning community beyond the student-teacher dynamic.
- **Analytics Dashboard:** Providing detailed analytics for teachers regarding student engagement, video drop-off rates, and course completion statistics.

Limitations:

- **Payment Processing:** The application currently operates in **Test Mode** using the Cashfree sandbox. Real-world financial transactions and handling of edge cases like refunds or chargebacks are not implemented.
- **Video Storage Optimization:** Video files are uploaded directly to S3 without server-side transcoding.
- **Scalability of WebSockets:** The current Socket.io implementation relies on a single backend instance. For high-scale usage, a Redis adapter would be needed to manage WebSocket connections across multiple server instances.
- **Content Moderation:** There is currently no automated moderation for chat messages or user-uploaded content, relying solely on the integrity of the users.

REFERENCES

Figma designs - link

Live - https://prolearn-lms.vercel.app/

Github - https://github.com/md-danishraza?tab=repositories&q=LMS

Official Documentation & Technologies:

1. **Next.js Documentation:** https://nextjs.org/docs - Used for Frontend Architecture and App Router implementation.
2. **React.js:** https://react.dev/ - Used for building the component-based user interface.
3. **Node.js & Express.js:** https://nodejs.org/en/docs/ - Used for developing the RESTful backend API.
4. **AWS Documentation:** https://docs.aws.amazon.com/ - Referenced for implementing DynamoDB (NoSQL), S3 (Storage), and CloudFront (CDN).
5. **TypeScript:** https://www.typescriptlang.org/docs/ - Used for static typing and code reliability.

Third-Party Services & SDKs:

6. **Clerk Authentication:** https://clerk.com/docs - Used for secure user management and session handling.

7. **Cashfree Payments:** https://docs.cashfree.com/ - Referenced for integrating the payment gateway and handling webhooks.

8. **Dynamoose ORM:** https://dynamoosejs.com/ - Used for modeling and querying data in DynamoDB.

9. **Shadcn/ui:** https://ui.shadcn.com/ - Used for pre-built, accessible UI components. 10. **Tailwind CSS:** https://tailwindcss.com/docs - Used for utility-first styling and responsive design.

Deployment Platforms:

11. **Vercel:** https://vercel.com/docs - Used for frontend deployment and CI/CD.
12. **Render:** https://render.com/docs - Used for hosting the Node.js backend service.