Technical Specification

System Requirements

Performance Requirements

- **Concurrent Users**: Support 50+ active rooms simultaneously with 2,000+ total users
- **Response Time**: < 200ms for API responses, < 50ms for WebSocket events
- **Availability**: 99.9% uptime during school hours (6 AM - 6 PM local time)
- **Real-time Latency**: < 100ms for submission approvals and roster updates
- **File Upload**: Support CSV files up to 1MB with 1,000+ student names

Scalability Requirements

- **Room Capacity**: 50 students per room maximum
- **Session Duration**: Support 8+ hour continuous sessions
- **Data Retention**: 2 years of participation history with archival capability
- **Export Capability**: Handle CSV exports up to 50,000 participation records
- **Authentication**: Support 10,000+ teacher accounts

Frontend Technical Specification

Technology Stack

- **Framework**: Next.js 14.2.28 with App Router
- **Language**: TypeScript 5.2.2 with strict mode enabled
- **Styling**: Tailwind CSS 3.3.3 with custom design system
- **UI Components**: Radix UI primitives with shadcn/ui component library
- **State Management**: React hooks with Context API and localStorage
- **Real-time**: Native WebSocket with custom event handling
- **Forms**: React Hook Form with Zod validation
- **File Handling**: Native File API with CSV parsing

Component Architecture

Authentication Components

Teacher Authentication System

```
interface TeacherAuthState {
  mode: 'login' | 'register'
  isLoading: boolean
  user: Teacher | null
  errors: Record<string, string>
}

interface TeacherAuthProps {
  onLogin: (credentials: LoginCredentials) => Promise<void>
  onRegister: (data: RegisterData) => Promise<void>
  onLogout: () => void
}

// Core authentication components:
// - TeacherLoginForm: Email/password authentication
// - TeacherRegisterForm: Account creation with validation
// - AuthModeToggle: Switch between login/register
// - PasswordStrengthIndicator: Real-time password validation
// - LogoutButton: Session termination with confirmation
```

Authentication Flow Features:
- Email format validation with regex pattern
- Password strength requirements (minimum 6 characters)
- Confirm password matching validation
- Loading states with spinner animations
- Error handling with specific error messages
- Success feedback with auto-redirect
- Remember me functionality via localStorage

Student Interface Components

Enhanced Student Landing (`/student`)

```
interface StudentLandingProps {
  onRoomJoin: (roomCode: string) => Promise<void>
  onStudentSelect: (studentId: string) => void
  roomData?: {
    code: string
    name: string
    students: Student[]
    isActive: boolean
  }
  validationError?: string
  isLoading: boolean
}

// Enhanced UI features:
// - Room code input with auto-uppercase formatting
// - Real-time room validation with visual feedback
// - Responsive student grid (2-4 columns based on screen size)
// - Radio button selection with hover states
// - Point selector dropdown positioned at interface top
// - Mobile-optimized touch targets (minimum 44px)
// - Loading skeletons during data fetching
// - Error states with retry functionality
```

Student Participation Interface

```
interface ParticipationInterfaceProps {
  student: Student
  room: Room
  currentPoints: number
  onSubmit: (points: 1 | 2 | 3) => Promise<void>
  submissionStatus: 'idle' | 'pending' | 'approved' | 'rejected'
  isSubmissionDisabled: boolean
}

// Participation features:
// - Point selection with visual indicators (1-3 points)
// - Submission status tracking with color-coded states
// - Real-time point updates via WebSocket
// - Rate limiting feedback (3 submissions per 5 minutes)
// - Success/failure animations
// - Accessibility support (ARIA labels, keyboard navigation)
```

Teacher Dashboard Components

Enhanced Teacher Dashboard (`/teacher`)

```
interface TeacherDashboardProps {
  teacher: Teacher
  rooms: EnhancedRoom[]
  stats: DashboardStats
  onRoomCreate: (data: RoomCreationData) => Promise<Room>
  onRoomDelete: (roomId: string) => Promise<void>
  onStudentUpload: (roomId: string, file: File) => Promise<UploadResult>
  onLogout: () => void
}

interface EnhancedRoom extends Room {
  _count: {
    students: number
    participations: number
    sessions: number
  }
  stats: {
    averageParticipation: number
    activeStudents: number
    pendingApprovals: number
  }
  lastActivity: Date
}
```

Dashboard Features:
- **Room Creation Dialog**: Multi-step form with CSV upload
- **Room Statistics Cards**: Student count, participation totals, activity indicators
- **Bulk Actions**: Multi-select for room operations
- **Search and Filter**: Find rooms by name, status, or activity
- **Activity Timeline**: Recent room activity feed
- **Quick Actions**: One-click room activation/deactivation

Room Management Features

```
interface RoomManagementFeatures {
  // CSV Upload System
  csvUpload: {
    validation: (file: File) => Promise<ValidationResult>
    preview: (file: File) => Promise<string[]>
    process: (file: File, roomId: string) => Promise<UploadResult>
  }

  // Room Deletion System
  deletion: {
    safeguards: ConfirmationDialog[]
    impact: DeletionImpactReport
    cascadeOptions: CascadeDeletionOptions
  }

  // Room Analytics
  analytics: {
    participationTrends: ChartData[]
    studentEngagement: EngagementMetrics
    sessionHistory: SessionSummary[]
  }
}
```

Presentation View Components

Enhanced Presentation Layout

```
interface PresentationViewProps {
  room: Room
  students: Student[]
  pendingSubmissions: Submission[]
  approvalQueue: ApprovalQueueItem[]
  onApproval: (submissionId: string, approved: boolean) => Promise<void>
  onBulkApproval: (submissionIds: string[], approved: boolean) => Promise<void>
  onReset: (type: ResetType, targetId?: string) => Promise<void>
  presentationSettings: PresentationSettings
}

interface PresentationSettings {
  layout: 'split' | 'overlay' | 'tabbed'
  fontSize: 'small' | 'medium' | 'large' | 'xlarge'
  theme: 'light' | 'dark' | 'high-contrast'
  autoScroll: boolean
  keyboardShortcuts: boolean
}
```

Presentation Features:
- **Responsive Layout System**: 70/30 split (large), tabbed (medium), overlay (small)
- **Keyboard Navigation**: Enter (approve), Escape (reject), Arrow keys (navigate)
- **Bulk Selection**: Checkbox selection for multiple approvals
- **Auto-scroll**: New submissions automatically scroll into view
- **Fullscreen Mode**: F11 support with escape handling
- **Presenter Notes**: Hidden notes visible only to teacher
- **QR Code Display**: Dynamic QR code for easy room joining

State Management Architecture

Global Application State

```
interface AppState {
  // Authentication state
  auth: {
    user: Teacher | null
    isAuthenticated: boolean
    token?: string
    expires?: Date
  }

  // Room management state
  rooms: {
    list: Room[]
    current?: Room
    loading: boolean
    error?: string
  }

  // Real-time connection state
  connection: {
    status: 'connected' | 'connecting' | 'disconnected'
    reconnectAttempts: number
    lastHeartbeat?: Date
  }

  // UI state
  ui: {
    theme: 'light' | 'dark'
    sidebarCollapsed: boolean
    notifications: NotificationItem[]
    modals: ModalState[]
  }
}
```

State Management Hooks

```
// Authentication hook
const useAuth = () => {
  const [auth, setAuth] = useLocalStorage<AuthState>('teacher-auth', null)

  const login = async (credentials: LoginCredentials) => {
    const response = await fetch('/api/auth/signin', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(credentials)
    })

    if (response.ok) {
      const { user } = await response.json()
      setAuth({ user, isAuthenticated: true, expires: new Date(Date.now() + 24 * 60 * 6
0 * 1000) })
      return user
    }
    throw new Error('Authentication failed')
  }

  const logout = () => {
    setAuth(null)
    window.location.href = '/teacher'
  }

  return { ...auth, login, logout }
}

// Real-time data hook
const useRealTimeRoom = (roomCode: string) => {
  const [roomData, setRoomData] = useState<RoomData | null>(null)
  const [socket, setSocket] = useState<WebSocket | null>(null)

  useEffect(() => {
    const ws = new WebSocket(`ws://localhost:3000/ws`)

    ws.onopen = () => {
      ws.send(JSON.stringify({ type: 'JOIN_ROOM', roomCode }))
    }

    ws.onmessage = (event) => {
      const data = JSON.parse(event.data)
      switch (data.type) {
        case 'ROOM_UPDATE':
          setRoomData(prev => ({ ...prev, ...data.payload }))
          break
        case 'STUDENT_UPDATE':
          setRoomData(prev => ({
            ...prev,
            students: prev?.students.map(s =>
              s.id === data.payload.id ? { ...s, ...data.payload } : s
            )
          }))
          break
      }
    }

    setSocket(ws)
    return () => ws.close()
  }, [roomCode])
```

```
  return { roomData, socket }
}
```

Responsive Design System

Breakpoint Configuration

```
/* Mobile First Responsive Design */
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  /* Mobile: 320px - 640px */
  .mobile-layout {
    @apply flex-col space-y-4 p-4;
  }

  /* Tablet: 641px - 1024px */
  .tablet-layout {
    @apply md:flex-row md:space-x-6 md:space-y-0 md:p-6;
  }

  /* Desktop: 1025px+ */
  .desktop-layout {
    @apply lg:grid lg:grid-cols-3 lg:gap-8 lg:p-8;
  }

  /* Presentation optimized layouts */
  .presentation-mobile {
    @apply block md:hidden;
  }

  .presentation-tablet {
    @apply hidden md:block lg:hidden;
  }

  .presentation-desktop {
    @apply hidden lg:flex lg:h-screen;
  }
}

/* Custom CSS Variables for Theme System */
:root {
  --color-primary: 219 234 254;
  --color-secondary: 241 245 249;
  --color-accent: 59 130 246;
  --color-success: 34 197 94;
  --color-warning: 245 158 11;
  --color-error: 239 68 68;

  --font-size-xs: 0.75rem;
  --font-size-sm: 0.875rem;
  --font-size-base: 1rem;
  --font-size-lg: 1.125rem;
  --font-size-xl: 1.25rem;
  --font-size-2xl: 1.5rem;

  --spacing-unit: 0.25rem;
  --border-radius: 0.375rem;
  --shadow-sm: 0 1px 2px 0 rgb(0 0 0 / 0.05);
  --shadow-md: 0 4px 6px -1px rgb(0 0 0 / 0.1);
}

[data-theme="dark"] {
  --color-primary: 30 41 59;
  --color-secondary: 51 65 85;
  --color-accent: 96 165 250;
}
```

Backend Technical Specification

Technology Stack

- **Runtime**: Node.js 18+ with TypeScript
- **Framework**: Next.js 14 API Routes
- **Database**: PostgreSQL 14+ with Prisma ORM 6.7.0
- **Authentication**: Custom bcrypt-based system
- **File Processing**: Native Node.js File System + CSV parsing
- **Real-time**: WebSocket (native or Socket.io)
- **Validation**: Zod for type-safe validation
- **Testing**: Jest + Supertest for API testing

Enhanced API Specification

Authentication Endpoints

Teacher Registration

```
POST /api/auth/signup
Content-Type: application/json

// Request
interface SignupRequest {
  name: string              // 1-100 characters
  email: string            // Valid email format
  password: string         // Minimum 6 characters
}

// Response (201 Created)
interface SignupResponse {
  success: true
  user: {
    id: string
    name: string
    email: string
    createdAt: string
  }
}

// Error Response (400 Bad Request)
interface SignupError {
  error: string
  details?: ValidationError[]
}

// Implementation with enhanced security
export async function POST(request: Request) {
  try {
    const { name, email, password } = await request.json()

    // Input validation
    const validation = signupSchema.safeParse({ name, email, password })
    if (!validation.success) {
      return NextResponse.json(
        { error: 'Validation failed', details: validation.error.issues },
        { status: 400 }
      )
    }

    // Check email uniqueness
    const existingTeacher = await prisma.teacher.findUnique({
      where: { email: email.toLowerCase() }
    })

    if (existingTeacher) {
      return NextResponse.json(
        { error: 'An account with this email already exists' },
        { status: 409 }
      )
    }

    // Hash password with salt
    const saltRounds = 12 // Increased for better security
    const hashedPassword = await bcrypt.hash(password, saltRounds)

    // Create teacher account
    const teacher = await prisma.teacher.create({
      data: {
        name: name.trim(),
        email: email.toLowerCase(),
```

```
        password: hashedPassword
      }
    })

    // Log successful registration
    logger.info('Teacher registered', {
      teacherId: teacher.id,
      email: teacher.email,
      timestamp: new Date().toISOString()
    })

    return NextResponse.json({
      success: true,
      user: {
        id: teacher.id,
        name: teacher.name,
        email: teacher.email,
        createdAt: teacher.createdAt.toISOString()
      }
    }, { status: 201 })

  } catch (error) {
    logger.error('Signup error', error)
    return NextResponse.json(
      { error: 'Failed to create account' },
      { status: 500 }
    )
  }
}
```

Teacher Login

```
POST /api/auth/signin
Content-Type: application/json

// Request
interface SigninRequest {
  email: string
  password: string
}

// Response (200 OK)
interface SigninResponse {
  success: true
  user: {
    id: string
    name: string
    email: string
  }
  session: {
    expires: string
  }
}

// Enhanced login implementation
export async function POST(request: Request) {
  try {
    const { email, password } = await request.json()

    // Rate limiting check
    const rateLimitKey = `login_attempts:${request.ip}`
    const attempts = await redis.incr(rateLimitKey)

    if (attempts === 1) {
      await redis.expire(rateLimitKey, 900) // 15 minutes
    }

    if (attempts > 5) {
      return NextResponse.json(
        { error: 'Too many login attempts. Please try again in 15 minutes.' },
        { status: 429 }
      )
    }

    // Find teacher
    const teacher = await prisma.teacher.findUnique({
      where: { email: email.toLowerCase() }
    })

    if (!teacher) {
      await new Promise(resolve => setTimeout(resolve, 1000)) // Timing attack preven-
tion
      return NextResponse.json(
        { error: 'Invalid email or password' },
        { status: 401 }
      )
    }

    // Verify password
    const isPasswordValid = await bcrypt.compare(password, teacher.password)

    if (!isPasswordValid) {
      await new Promise(resolve => setTimeout(resolve, 1000)) // Timing attack preven-
tion
```

```
      return NextResponse.json(
        { error: 'Invalid email or password' },
        { status: 401 }
      )
    }

    // Clear rate limit on successful login
    await redis.del(rateLimitKey)

    // Update last login
    await prisma.teacher.update({
      where: { id: teacher.id },
      data: { lastLoginAt: new Date() }
    })

    logger.info('Teacher signed in', {
      teacherId: teacher.id,
      email: teacher.email,
      ip: request.ip
    })

    return NextResponse.json({
      success: true,
      user: {
        id: teacher.id,
        name: teacher.name,
        email: teacher.email
      },
      session: {
        expires: new Date(Date.now() + 24 * 60 * 60 * 1000).toISOString()
      }
    })

  } catch (error) {
    logger.error('Signin error', error)
    return NextResponse.json(
      { error: 'Login failed' },
      { status: 500 }
    )
  }
}
```

Room Management Endpoints

Enhanced Room Creation

```
POST /api/rooms/create
Content-Type: multipart/form-data

// Request (FormData)
interface CreateRoomFormData {
  name: string              // Room name (1-100 characters)
  description?: string      // Optional description (max 500 characters)
  teacherId: string        // Teacher ID from authentication
  csvFile: File            // CSV file with student names
}

// Response (201 Created)
interface CreateRoomResponse {
  success: true
  room: {
    id: string
    code: string            // 6-character unique code
    name: string
    description?: string
    isActive: boolean
    createdAt: string
  }
  students: {
    created: number         // Number of students created
    total: number           // Total students in CSV
    names: string[]         // List of created student names
  }
}

// Implementation
export async function POST(request: Request) {
  try {
    const formData = await request.formData()
    const name = formData.get('name') as string
    const description = formData.get('description') as string
    const teacherId = formData.get('teacherId') as string
    const csvFile = formData.get('csvFile') as File

    // Validate inputs
    const validation = createRoomSchema.safeParse({
      name,
      description,
      teacherId,
      csvFile: csvFile ? {
        name: csvFile.name,
        size: csvFile.size,
        type: csvFile.type
      } : null
    })

    if (!validation.success) {
      return NextResponse.json(
        { error: 'Validation failed', details: validation.error.issues },
        { status: 400 }
      )
    }

    // Process CSV file
    const csvText = await csvFile.text()
    const studentNames = parseCsvStudentNames(csvText)

    if (studentNames.length === 0) {
```

```
      return NextResponse.json(
        { error: 'CSV file must contain at least one student name' },
        { status: 400 }
      )
    }

    // Generate unique room code
    let roomCode: string
    let isUnique = false
    let attempts = 0

    while (!isUnique && attempts < 10) {
      roomCode = generateRoomCode()
      const existing = await prisma.room.findUnique({
        where: { code: roomCode }
      })
      isUnique = !existing
      attempts++
    }

    if (!isUnique) {
      return NextResponse.json(
        { error: 'Failed to generate unique room code' },
        { status: 500 }
      )
    }

    // Create room and students in transaction
    const result = await prisma.$transaction(async (tx) => {
      // Create room
      const room = await tx.room.create({
        data: {
          code: roomCode!,
          name: name.trim(),
          description: description?.trim() || null,
          teacherId,
          isActive: true
        }
      })

      // Create students
      const studentsData = studentNames.map(studentName => ({
        name: studentName,
        roomId: room.id
      }))

      const students = await tx.student.createMany({
        data: studentsData,
        skipDuplicates: true
      })

      // Create initial session
      await tx.session.create({
        data: {
          name: `${room.name} - Initial Session`,
          roomId: room.id,
          isActive: true
        }
      })

      return { room, studentsCreated: students.count }
    })
```

```
    logger.info('Room created', {
      roomId: result.room.id,
      roomCode: result.room.code,
      teacherId,
      studentsCount: result.studentsCreated
    })

    return NextResponse.json({
      success: true,
      room: {
        id: result.room.id,
        code: result.room.code,
        name: result.room.name,
        description: result.room.description,
        isActive: result.room.isActive,
        createdAt: result.room.createdAt.toISOString()
      },
      students: {
        created: result.studentsCreated,
        total: studentNames.length,
        names: studentNames
      }
    }, { status: 201 })

  } catch (error) {
    logger.error('Room creation failed', error)
    return NextResponse.json(
      { error: 'Failed to create room' },
      { status: 500 }
    )
  }
}
```

Room Deletion with Safety Checks

```
DELETE /api/rooms/[id]/delete
Authorization: Bearer {teacherToken}

// Response (200 OK)
interface DeleteRoomResponse {
  success: true
  message: string
  deletedCounts: {
    students: number
    participations: number
    sessions: number
  }
}

// Implementation with enhanced safety
export async function DELETE(
  request: Request,
  { params }: { params: { id: string } }
) {
  try {
    const roomId = params.id

    // Verify teacher authorization
    const teacherId = await getTeacherIdFromRequest(request)
    if (!teacherId) {
      return NextResponse.json(
        { error: 'Authentication required' },
        { status: 401 }
      )
    }

    // Get room with ownership verification
    const room = await prisma.room.findFirst({
      where: {
        id: roomId,
        teacherId // Ensure teacher owns this room
      },
      include: {
        _count: {
          select: {
            students: true,
            participations: true,
            sessions: true
          }
        }
      }
    })

    if (!room) {
      return NextResponse.json(
        { error: 'Room not found or access denied' },
        { status: 404 }
      )
    }

    // Safety check for active sessions
    const activeSessions = await prisma.session.count({
      where: {
        roomId,
        isActive: true
      }
    })
```

```
    if (activeSessions > 0) {
      return NextResponse.json(
        { error: 'Cannot delete room with active sessions. Please end all sessions
first.' },
        { status: 409 }
      )
    }

    // Perform deletion in transaction
    const deletedCounts = await prisma.$transaction(async (tx) => {
      // Archive data before deletion (optional)
      await tx.roomArchive.create({
        data: {
          originalRoomId: room.id,
          roomCode: room.code,
          roomName: room.name,
          teacherId: room.teacherId,
          studentsCount: room._count.students,
          participationsCount: room._count.participations,
          sessionsCount: room._count.sessions,
          deletedAt: new Date()
        }
      })

      // Delete room (cascade will handle related data)
      await tx.room.delete({
        where: { id: roomId }
      })

      return room._count
    })

    logger.warn('Room deleted', {
      roomId,
      roomCode: room.code,
      teacherId,
      studentsDeleted: deletedCounts.students,
      participationsDeleted: deletedCounts.participations,
      sessionsDeleted: deletedCounts.sessions
    })

    return NextResponse.json({
      success: true,
      message: `Room "${room.name}" has been permanently deleted`,
      deletedCounts
    })

  } catch (error) {
    logger.error('Room deletion failed', error)
    return NextResponse.json(
      { error: 'Failed to delete room' },
      { status: 500 }
    )
  }
}
```

CSV Student Upload to Existing Room

```
POST /api/rooms/[id]/upload-students
Content-Type: multipart/form-data
Authorization: Bearer {teacherToken}

// Request
interface UploadStudentsRequest {
  csvFile: File            // CSV file with student names
}

// Response (200 OK)
interface UploadStudentsResponse {
  success: true
  studentsAdded: number
  duplicatesSkipped: number
  totalProcessed: number
  newStudents: string[]
  errors?: string[]
}

// Implementation
export async function POST(
  request: Request,
  { params }: { params: { id: string } }
) {
  try {
    const roomId = params.id
    const formData = await request.formData()
    const csvFile = formData.get('csvFile') as File

    // Verify teacher authorization and room ownership
    const teacherId = await getTeacherIdFromRequest(request)
    const room = await prisma.room.findFirst({
      where: { id: roomId, teacherId }
    })

    if (!room) {
      return NextResponse.json(
        { error: 'Room not found or access denied' },
        { status: 404 }
      )
    }

    // Validate CSV file
    if (!csvFile || csvFile.size === 0) {
      return NextResponse.json(
        { error: 'CSV file is required' },
        { status: 400 }
      )
    }

    if (csvFile.size > 1024 * 1024) { // 1MB limit
      return NextResponse.json(
        { error: 'CSV file too large. Maximum size is 1MB.' },
        { status: 413 }
      )
    }

    // Process CSV
    const csvText = await csvFile.text()
    const { studentNames, errors } = parseCsvStudentNames(csvText, {
      maxNames: 1000,
      validateNames: true
```

```
    })

    if (studentNames.length === 0) {
      return NextResponse.json(
        { error: 'CSV file must contain at least one valid student name' },
        { status: 400 }
      )
    }

    // Check for existing students
    const existingStudents = await prisma.student.findMany({
      where: { roomId },
      select: { name: true }
    })

    const existingNames = new Set(existingStudents.map(s => s.name.toLowerCase()))
    const newStudentNames = studentNames.filter(
      name => !existingNames.has(name.toLowerCase())
    )

    // Create new students
    let studentsAdded = 0
    if (newStudentNames.length > 0) {
      const studentsData = newStudentNames.map(name => ({
        name,
        roomId
      }))

      const result = await prisma.student.createMany({
        data: studentsData,
        skipDuplicates: true
      })

      studentsAdded = result.count
    }

    // Update room activity
    await prisma.room.update({
      where: { id: roomId },
      data: { lastActivityAt: new Date() }
    })

    logger.info('Students uploaded to room', {
      roomId,
      roomCode: room.code,
      studentsAdded,
      duplicatesSkipped: studentNames.length - newStudentNames.length,
      totalProcessed: studentNames.length
    })

    return NextResponse.json({
      success: true,
      studentsAdded,
      duplicatesSkipped: studentNames.length - newStudentNames.length,
      totalProcessed: studentNames.length,
      newStudents: newStudentNames,
      ...(errors.length > 0 && { errors })
    })

  } catch (error) {
    logger.error('CSV upload failed', error)
    return NextResponse.json(
      { error: 'Failed to upload students' },
```

```
      { status: 500 }
    )
  }
}
```

Database Implementation

Enhanced Prisma Schema

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Teacher {
  id              String   @id @default(cuid())
  name            String
  email           String   @unique
  password        String   // bcrypt hashed password
  isActive        Boolean  @default(true)
  lastLoginAt     DateTime?
  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt

  // Relations
  rooms           Room[]

  // Indexes
  @@index([email])
  @@index([isActive, lastLoginAt])
  @@map("teachers")
}

model Room {
  id              String    @id @default(cuid())
  code            String    @unique @db.VarChar(6)
  name            String
  description     String?
  teacherId       String
  isActive        Boolean   @default(true)
  maxStudents     Int       @default(50)
  createdAt       DateTime  @default(now())
  updatedAt       DateTime  @updatedAt
  lastActivityAt  DateTime  @default(now())

  // Relations
  teacher         Teacher         @relation(fields: [teacherId], references: [id], onD
elete: Cascade)
  students        Student[]
  sessions        Session[]
  participations  Participation[]

  // Indexes
  @@index([code])
  @@index([teacherId, isActive])
  @@index([lastActivityAt])
  @@map("rooms")
}

model Student {
  id              String   @id @default(cuid())
  name            String
  roomId          String
  totalPoints     Int      @default(0)
  isOnline        Boolean  @default(false)
  lastActive      DateTime @default(now())
  createdAt       DateTime @default(now())
```

```
  // Relations
  room            Room           @relation(fields: [roomId], references: [id], onDele
te: Cascade)
  participations  Participation[]

  // Constraints
  @@unique([name, roomId])

  // Indexes
  @@index([roomId, totalPoints])
  @@index([roomId, name])
  @@map("students")
}

model Session {
  id              String   @id @default(cuid())
  name            String
  roomId          String
  isActive        Boolean  @default(true)
  startedAt       DateTime @default(now())
  endedAt         DateTime?
  createdAt       DateTime @default(now())

  // Relations
  room            Room           @relation(fields: [roomId], references: [id], onDele
te: Cascade)
  participations  Participation[]

  // Indexes
  @@index([roomId, isActive])
  @@index([isActive, startedAt])
  @@map("sessions")
}

model Participation {
  id          String               PRIMARY KEY DEFAULT cuid()
  studentId   String               NOT NULL FOREIGN KEY → Student.id
  roomId      String               NOT NULL FOREIGN KEY → Room.id
  sessionId   String               NOT NULL FOREIGN KEY → Session.id
  points      Int                  NOT NULL CHECK (points >= 1 AND points <= 3)
  status      ParticipationStatus  DEFAULT 'PENDING'
  submittedAt DateTime             DEFAULT now()
  processedAt DateTime?
  approvedBy  String?              // Teacher ID who approved/rejected
  notes       String?              // Optional notes from teacher

  // Relations
  student     Student   @relation(fields: [studentId], references: [id], onDelete: Cas
cade)
  room        Room      @relation(fields: [roomId], references: [id], onDelete: Cas-
cade)
  session     Session   @relation(fields: [sessionId], references: [id], onDelete: Cas
cade)

  // Indexes
  @@index([status, roomId, submittedAt])
  @@index([studentId, status, submittedAt])
  @@index([sessionId, status])
  @@map("participations")
}

// Archive model for deleted rooms
```

```
model RoomArchive {
  id                    String   @id @default(cuid())
  originalRoomId        String
  roomCode              String
  roomName              String
  teacherId             String
  studentsCount         Int
  participationsCount   Int
  sessionsCount         Int
  deletedAt             DateTime
  deletedBy             String?

  @@map("room_archives")
}

enum ParticipationStatus {
  PENDING
  APPROVED
  REJECTED
}
```

Performance Optimizations

Database Query Optimization

Complex Query Examples

```
// Teacher dashboard with aggregated statistics
const getTeacherDashboardData = async (teacherId: string) => {
  const [teacher, rooms, stats] = await Promise.all([
    // Teacher basic info
    prisma.teacher.findUnique({
      where: { id: teacherId },
      select: { id: true, name: true, email: true }
    }),

    // Rooms with detailed statistics
    prisma.room.findMany({
      where: { teacherId },
      include: {
        _count: {
          select: {
            students: true,
            participations: { where: { status: 'APPROVED' } },
            sessions: true
          }
        },
        sessions: {
          where: { isActive: true },
          select: { id: true, name: true }
        }
      },
      orderBy: { lastActivityAt: 'desc' }
    }),

    // Aggregated teacher statistics
    prisma.$queryRaw`
      SELECT
        COUNT(DISTINCT r.id) as total_rooms,
        COUNT(DISTINCT s.id) as total_students,
        COUNT(DISTINCT p.id) FILTER (WHERE p.status = 'APPROVED') as
total_participations,
        AVG(student_stats.avg_points) as avg_points_per_student
      FROM rooms r
      LEFT JOIN students s ON r.id = s.room_id
      LEFT JOIN participations p ON s.id = p.student_id
      LEFT JOIN (
        SELECT
          s.room_id,
          AVG(s.total_points) as avg_points
        FROM students s
        GROUP BY s.room_id
      ) student_stats ON r.id = student_stats.room_id
      WHERE r.teacher_id = ${teacherId}
    `
  ])

  return { teacher, rooms, stats: stats[0] }
}

// Optimized real-time room data query
const getRoomRealtimeData = async (roomCode: string) => {
  return await prisma.room.findUnique({
    where: { code: roomCode },
    include: {
      students: {
        select: {
          id: true,
          name: true,
```

```
          totalPoints: true,
          isOnline: true
        },
        orderBy: [
          { totalPoints: 'desc' },
          { name: 'asc' }
        ]
      },
      participations: {
        where: { status: 'PENDING' },
        include: {
          student: {
            select: { name: true }
          }
        },
        orderBy: { submittedAt: 'asc' }
      },
      sessions: {
        where: { isActive: true },
        select: { id: true, name: true }
      }
    }
  })
}

// Bulk approval processing with optimistic updates
const processBulkApprovals = async (
  submissionIds: string[],
  approved: boolean,
  teacherId: string
) => {
  return await prisma.$transaction(async (tx) => {
    // Get submissions with student data
    const submissions = await tx.participation.findMany({
      where: {
        id: { in: submissionIds },
        status: 'PENDING' // Only process pending submissions
      },
      include: { student: true }
    })

    if (submissions.length === 0) {
      throw new Error('No pending submissions found')
    }

    // Update participation records
    await tx.participation.updateMany({
      where: { id: { in: submissions.map(s => s.id) } },
      data: {
        status: approved ? 'APPROVED' : 'REJECTED',
        processedAt: new Date(),
        approvedBy: teacherId
      }
    })

    // Update student points if approved
    if (approved) {
      for (const submission of submissions) {
        await tx.student.update({
          where: { id: submission.studentId },
          data: {
            totalPoints: {
              increment: submission.points
```

```
            }
          }
        })
      }
    }

    // Update room activity
    await tx.room.update({
      where: { id: submissions[0].roomId },
      data: { lastActivityAt: new Date() }
    })

    return {
      processedCount: submissions.length,
      pointsAwarded: approved ? submissions.reduce((sum, s) => sum + s.points, 0) : 0,
      updatedStudents: submissions.map(s => ({
        studentId: s.studentId,
        newTotal: approved ? s.student.totalPoints + s.points : s.student.totalPoints
      }))
    }
  })
}
```

Caching Strategy

Multi-layer Caching Implementation

```
import Redis from 'ioredis'
import { LRUCache } from 'lru-cache'

// Redis for distributed caching
const redis = new Redis({
  host: process.env.REDIS_HOST || 'localhost',
  port: parseInt(process.env.REDIS_PORT || '6379'),
  retryDelayOnFailover: 100,
  maxRetriesPerRequest: 3,
  lazyConnect: true
})

// In-memory LRU cache for frequently accessed data
const memoryCache = new LRUCache<string, any>({
  max: 1000,
  ttl: 5 * 60 * 1000 // 5 minutes
})

class CacheManager {
  // L1 Cache: Memory (fastest)
  async getFromMemory(key: string) {
    return memoryCache.get(key)
  }

  async setToMemory(key: string, data: any, ttlMs: number = 300000) {
    memoryCache.set(key, data, { ttl: ttlMs })
  }

  // L2 Cache: Redis (distributed)
  async getFromRedis(key: string) {
    try {
      const cached = await redis.get(key)
      return cached ? JSON.parse(cached) : null
    } catch (error) {
      logger.error('Redis get error', error)
      return null
    }
  }

  async setToRedis(key: string, data: any, ttlSeconds: number = 300) {
    try {
      await redis.setex(key, ttlSeconds, JSON.stringify(data))
    } catch (error) {
      logger.error('Redis set error', error)
    }
  }

  // L3 Cache: Database (source of truth)
  async getWithCache<T>(
    key: string,
    fetchFn: () => Promise<T>,
    ttlSeconds: number = 300
  ): Promise<T> {
    // Try memory cache first
    let data = this.getFromMemory(key)
    if (data) return data

    // Try Redis cache
    data = await this.getFromRedis(key)
    if (data) {
      this.setToMemory(key, data)
      return data
```

```
    }

    // Fetch from database
    data = await fetchFn()

    // Store in both caches
    this.setToMemory(key, data)
    this.setToRedis(key, data, ttlSeconds)

    return data
  }

  // Cache invalidation
  async invalidate(pattern: string) {
    memoryCache.clear()
    try {
      const keys = await redis.keys(pattern)
      if (keys.length > 0) {
        await redis.del(...keys)
      }
    } catch (error) {
      logger.error('Cache invalidation error', error)
    }
  }
}

const cache = new CacheManager()

// Usage examples
const getCachedRoomData = async (roomCode: string) => {
  return await cache.getWithCache(
    `room:${roomCode}`,
    () => getRoomRealtimeData(roomCode),
    120 // 2 minutes TTL
  )
}

const getCachedTeacherRooms = async (teacherId: string) => {
  return await cache.getWithCache(
    `teacher:${teacherId}:rooms`,
    () => prisma.room.findMany({
      where: { teacherId },
      include: { _count: { select: { students: true } } }
    }),
    300 // 5 minutes TTL
  )
}
```

Security Implementation

Enhanced Input Validation

Comprehensive Zod Schemas

```
import { z } from 'zod'

// Enhanced validation schemas
export const schemas = {
  // Authentication schemas
  teacherSignup: z.object({
    name: z.string()
      .min(1, 'Name is required')
      .max(100, 'Name must be less than 100 characters')
      .regex(/^[a-zA-Z\s\-\.]+$/, 'Name can only contain letters, spaces, hyphens, and
periods'),

    email: z.string()
      .email('Invalid email format')
      .max(255, 'Email too long')
      .transform(val => val.toLowerCase()),

    password: z.string()
      .min(6, 'Password must be at least 6 characters')
      .max(128, 'Password too long')
      .regex(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, 'Password must contain at least one
lowercase letter, one uppercase letter, and one number')
  }),

  teacherSignin: z.object({
    email: z.string().email().transform(val => val.toLowerCase()),
    password: z.string().min(1, 'Password is required')
  }),

  // Room management schemas
  createRoom: z.object({
    name: z.string()
      .min(1, 'Room name is required')
      .max(100, 'Room name too long')
      .regex(/^[a-zA-Z0-9\s\-_\.]+$/, 'Room name contains invalid characters'),

    description: z.string()
      .max(500, 'Description too long')
      .optional(),

    teacherId: z.string().cuid('Invalid teacher ID'),

    csvFile: z.object({
      name: z.string().endsWith('.csv', 'File must be a CSV'),
      size: z.number().max(1024 * 1024, 'File size must be less than 1MB'),
      type: z.string().includes('csv', 'Invalid file type')
    })
  }),

  // Student participation schemas
  submitParticipation: z.object({
    roomCode: z.string()
      .regex(/^[A-Z0-9]{6}$/, 'Invalid room code format')
      .transform(val => val.toUpperCase()),

    studentId: z.string().cuid('Invalid student ID'),

    points: z.number()
      .int('Points must be a whole number')
      .min(1, 'Minimum 1 point')
      .max(3, 'Maximum 3 points')
  }),
```

```
  // CSV upload validation
  csvUpload: z.object({
    file: z.instanceof(File)
      .refine(file => file.size > 0, 'File cannot be empty')
      .refine(file => file.size <= 1024 * 1024, 'File size must be less than 1MB')
      .refine(file => file.type === 'text/csv' || file.name.endsWith('.csv'), 'File
must be a CSV')
  })
}

// Validation middleware
export const validateRequest = (schema: z.ZodSchema) => {
  return async (req: Request) => {
    try {
      const body = await req.json()
      return schema.parse(body)
    } catch (error) {
      if (error instanceof z.ZodError) {
        throw new ValidationError('Invalid request data', error.issues)
      }
      throw error
    }
  }
}
```

Rate Limiting Implementation

Advanced Rate Limiting

```
interface RateLimitConfig {
  windowMs: number
  maxRequests: number
  message: string
  skipSuccessfulRequests?: boolean
  skipFailedRequests?: boolean
}

class RateLimiter {
  private redis: Redis
  private configs: Map<string, RateLimitConfig>

  constructor(redisInstance: Redis) {
    this.redis = redisInstance
    this.configs = new Map([
      ['auth', {
        windowMs: 15 * 60 * 1000, // 15 minutes
        maxRequests: 5,
        message: 'Too many authentication attempts'
      }],
      ['submission', {
        windowMs: 5 * 60 * 1000, // 5 minutes
        maxRequests: 20,
        message: 'Too many participation submissions'
      }],
      ['roomCreation', {
        windowMs: 60 * 60 * 1000, // 1 hour
        maxRequests: 10,
        message: 'Too many room creation attempts'
      }],
      ['csvUpload', {
        windowMs: 10 * 60 * 1000, // 10 minutes
        maxRequests: 50,
        message: 'Too many CSV upload attempts'
      }]
    ])
  }

  async checkLimit(
    type: string,
    identifier: string,
    customConfig?: Partial<RateLimitConfig>
  ): Promise<{ allowed: boolean; remaining: number; resetTime: Date }> {
    const config = { ...this.configs.get(type), ...customConfig }
    if (!config) {
      throw new Error(`Unknown rate limit type: ${type}`)
    }

    const key = `rate_limit:${type}:${identifier}`
    const now = Date.now()
    const windowStart = now - config.windowMs

    try {
      // Use Redis sorted sets for sliding window
      const pipeline = this.redis.pipeline()

      // Remove old entries
      pipeline.zremrangebyscore(key, 0, windowStart)

      // Count current requests
      pipeline.zcard(key)
```

```
      // Add current request
      pipeline.zadd(key, now, `${now}-${Math.random()}`)

      // Set expiration
      pipeline.expire(key, Math.ceil(config.windowMs / 1000))

      const results = await pipeline.exec()
      const count = results![1][1] as number

      const allowed = count < config.maxRequests
      const remaining = Math.max(0, config.maxRequests - count - 1)
      const resetTime = new Date(now + config.windowMs)

      return { allowed, remaining, resetTime }
    } catch (error) {
      logger.error('Rate limiting error', error)
      // Fail open - allow request if Redis is down
      return { allowed: true, remaining: config.maxRequests, resetTime: new Date() }
    }
  }

  middleware(type: string, getIdentifier: (req: Request) => string) {
    return async (req: Request) => {
      const identifier = getIdentifier(req)
      const result = await this.checkLimit(type, identifier)

      if (!result.allowed) {
        const config = this.configs.get(type)!
        throw new RateLimitError(config.message, result.resetTime)
      }

      return result
    }
  }
}

// Usage in API routes
const rateLimiter = new RateLimiter(redis)

export const authRateLimit = rateLimiter.middleware('auth', (req) => {
  const forwarded = req.headers.get('x-forwarded-for')
  const ip = forwarded ? forwarded.split(',')[0] : req.headers.get('x-real-ip') || 'un-
known'
  return ip
})

export const submissionRateLimit = rateLimiter.middleware('submission', (req) => {
  // Rate limit by student ID + IP for participation submissions
  const body = JSON.parse(req.body)
  const ip = req.headers.get('x-forwarded-for') || req.headers.get('x-real-ip') || 'un-
known'
  return `${body.studentId}:${ip}`
})
```

This comprehensive technical specification provides the complete implementation details for building and maintaining the Classroom Participation Tracker with all enhanced features including authentication, room management, CSV operations, and performance optimizations.