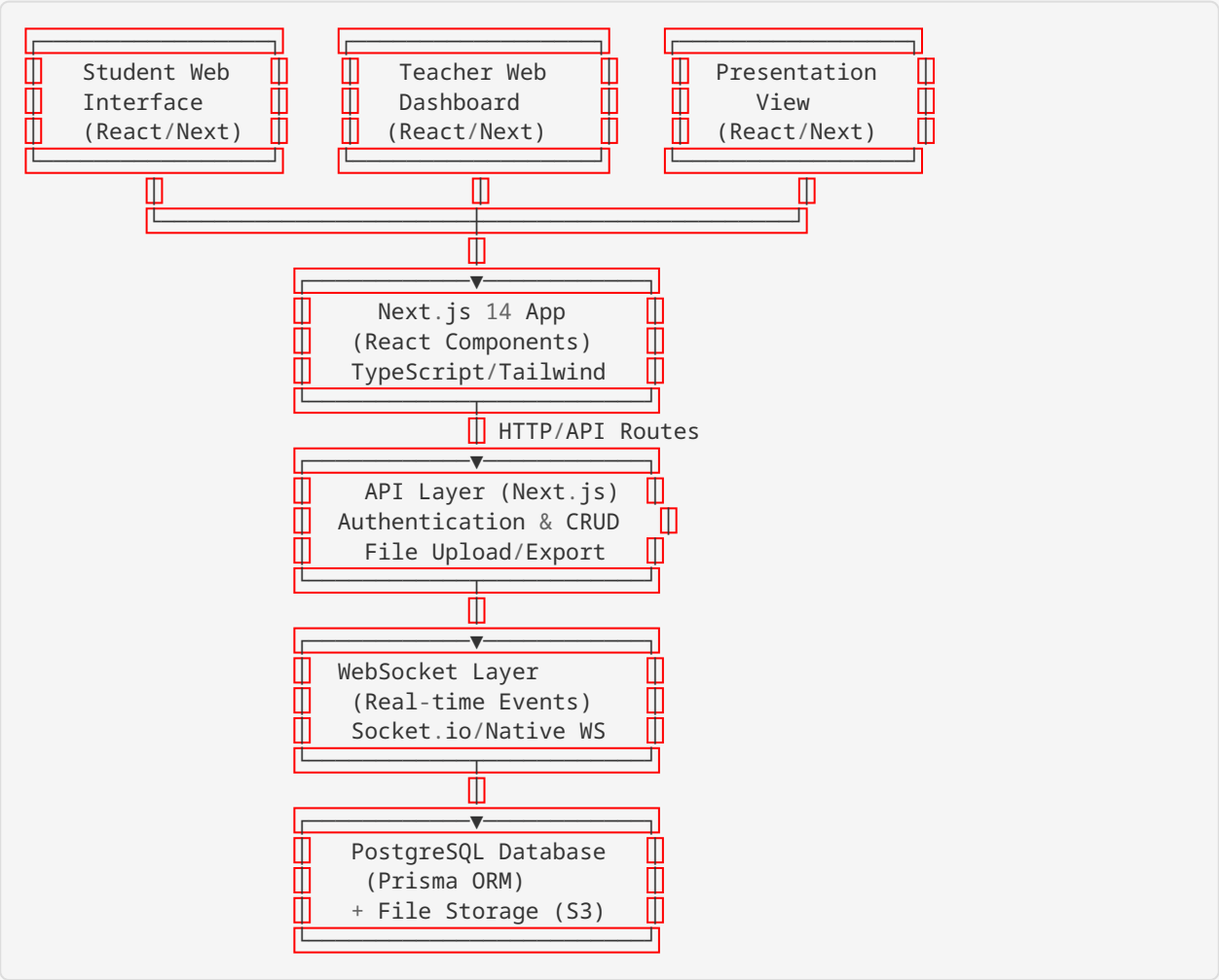
# System Architecture

## Overview

The Classroom Participation Tracker is built as a modern, secure web application with real-time capabilities. The system is designed to scale across multiple concurrent teacher sessions while maintaining simplicity, security, and performance. This document outlines the complete system architecture, including recent enhancements for authentication, file management, and room administration.

## System Architecture Diagram




## Component Architecture

### Frontend Layer

#### Authentication Components

##### Teacher Authentication Flow

```
// Authentication states and components
interface AuthState {
  mode: 'login' | 'register'
  isLoading: boolean
  user: Teacher | null
}

// Core auth components:
- TeacherLogin: Email/password authentication
- TeacherRegister: Account creation with validation
- AuthToggle: Switch between login/register modes
- LogoutButton: Secure session termination
```

**Features:**

- Password strength validation (minimum 6 characters)
- Email format validation
- Secure password confirmation
- Loading states with visual feedback
- Error handling with user-friendly messages

## Student Interface Components

### Student Landing ( `/student` )

```
interface StudentLandingProps {
  onRoomJoin: (roomCode: string) => void
  validationError?: string
  isLoading: boolean
}

// Enhanced features:
- Room code input with real-time validation
- Columned student list with radio button selection
- Point selection dropdown positioned at top
- Real-time room status checking
- Mobile-optimized responsive design
```

## Teacher Dashboard Components

### Teacher Dashboard ( `/teacher` )

```
interface TeacherDashboardProps {
  teacher: Teacher
  rooms: Room[]
  onRoomCreate: (roomData: RoomCreationData) => void
  onRoomDelete: (roomId: string) => void
  onStudentUpload: (roomId: string, file: File) => void
}

// New room management features:
- Room creation with CSV student roster upload
- Add students to existing rooms via CSV
- Safe room deletion with confirmation dialogs
- Room statistics and activity monitoring
- Logout functionality with session cleanup
```

**Room Management Features:**
- **Create Room**: Name, description, and initial CSV upload
- **Add Students**: Upload additional students to existing rooms
- **Delete Room**: Confirmation dialog showing impact (students/participations)
- **Room Statistics**: Student count, participation totals, activity status

### Presentation View Components

**Presentation Layout ( `/teacher/[roomCode]/presentation` )**

```
interface PresentationViewProps {
  room: Room
  students: Student[]
  pendingSubmissions: Submission[]
  onApproval: (submissionId: string, approved: boolean) => void
  onReset: (type: ResetType, targetId?: string) => void
}

// Enhanced layout features:
- 75% student roster with real-time point updates
- 25% compact approval queue (fixed position)
- Keyboard shortcuts for quick approvals (Enter/Escape)
- Responsive breakpoints for different screen sizes
- Auto-scroll to new submissions
```

### Shared UI Components

**Enhanced Component Library:**

```
// Core UI Components (shadcn/ui based)
- Button: Multiple variants with loading states
- Input: Validation states and file upload support
- Dialog: Confirmation and form dialogs
- AlertDialog: Destructive action confirmations
- Badge: Status indicators and counters
- Card: Content containers with hover effects
- Toast: Success/error notifications
- LoadingSpinner: Async operation feedback

// Custom Components
- CSVUploadDialog: File validation and preview
- ConfirmDeleteDialog: Safety confirmations for destructive actions
- RoomStatsCard: Statistics display with icons
- StudentList: Responsive columned layout with selection
```

## Backend Layer

### Authentication System

**Password-Based Authentication**

```
// Authentication endpoints
POST /api/auth/signup {
  name: string
  email: string
  password: string (min 6 chars)
}

POST /api/auth/signin {
  email: string
  password: string
}

// Security features:
- bcrypt password hashing (10 rounds)
- Email uniqueness validation
- Password strength requirements
- Secure session management
- Input sanitization and validation
```

### Enhanced API Routes Structure

```
/api/
├── auth/                    # Authentication system
│   ├── signup/             # Teacher registration
│   ├── signin/             # Teacher login
│   └── [...nextauth]/      # NextAuth integration (if needed)
├── rooms/                  # Room management
│   ├── create/             # Create room with CSV upload
│   ├── validate/           # Room code validation
│   ├── [id]/
│   │   ├── delete/         # Safe room deletion
│   │   ├── upload-students/ # Add students via CSV
│   │   ├── stats/          # Room statistics
│   │   ├── students/       # Student roster
│   │   └── sessions/       # Session management
├── students/               # Student operations
│   ├── join/               # Join room session
│   ├── submit/             # Submit participation points
│   └── status/             # Student status and points
├── participations/         # Participation management
│   ├── pending/            # Get pending approvals
│   ├── [id]/
│   │   ├── approve/        # Approve participation
│   │   └── reject/         # Reject participation
├── export/                 # Data export
│   └── csv/                # CSV export functionality
└── reset/                  # Reset operations
    ├── student/            # Individual student reset
    ├── class/              # Full class reset
    └── session/            # Session reset
```

### File Management System

#### CSV Upload Processing

```
// Enhanced CSV processing
interface CSVUploadResult {
  studentsAdded: number
  duplicatesSkipped: number
  totalProcessed: number
  errors: string[]
}

// Features:
- File type validation (.csv only)
- Content parsing with error handling
- Duplicate detection and reporting
- Preview functionality (first 10 names)
- Batch student creation with transaction safety
```

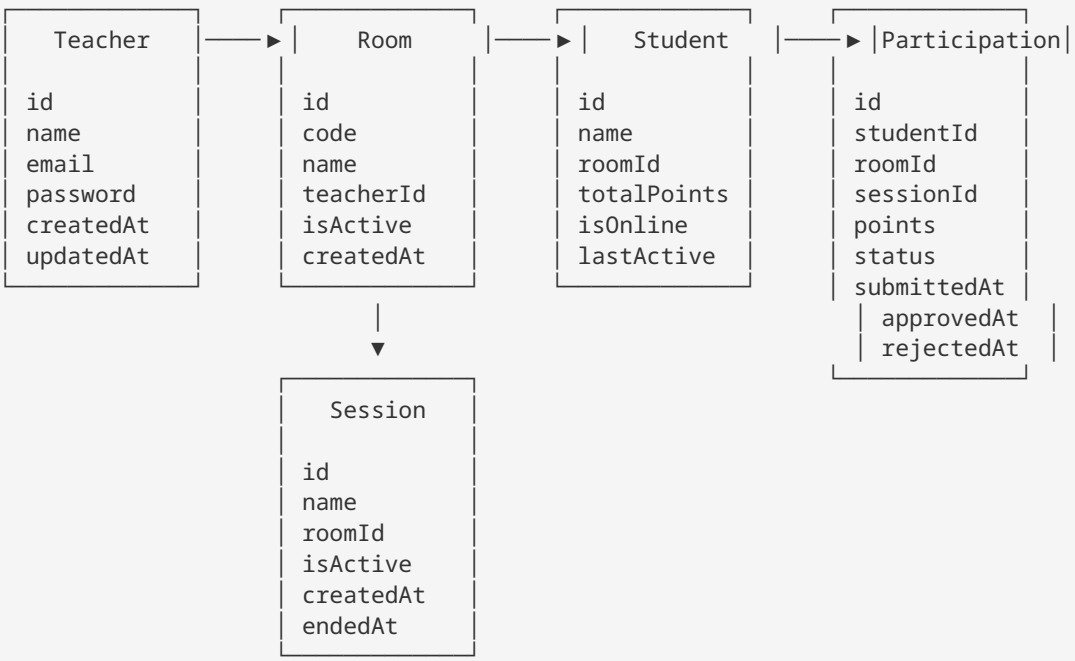
## WebSocket Events (Enhanced)

### Real-time Communication Events

```
// Client → Server Events
interface SocketEvents {
  'room:join': { roomCode: string, role: 'teacher' | 'student' }
  'room:leave': { roomCode: string }
  'submission:create': { studentId: string, points: number }
  'approval:process': { submissionIds: string[], approved: boolean }
  'room:update': { roomId: string, changes: Partial<Room> }
}

// Server → Client Events
interface ServerEvents {
  'room:status': { isActive: boolean, participantCount: number }
  'roster:update': { students: Student[], timestamp: number }
  'queue:update': { submissions: Submission[] }
  'points:update': { studentId: string, newTotal: number }
  'room:deleted': { roomId: string, message: string }
  'student:added': { students: Student[], count: number }
}
```

# Database Architecture

## Enhanced Entity Relationship Diagram




## Enhanced Schema Details

### Teacher Table (Updated)

```
CREATE TABLE Teacher (
  id              String    PRIMARY KEY DEFAULT cuid()
  name            String    NOT NULL
  email           String    UNIQUE NOT NULL
  password        String    NOT NULL  -- bcrypt hashed password
  createdAt       DateTime  DEFAULT now()
  updatedAt       DateTime  DEFAULT now()

  -- Relations
  rooms           Room[]

  -- Indexes
  INDEX idx_teacher_email (email)
)
```

### Room Table (Enhanced)

```
CREATE TABLE Room (
  id              String    PRIMARY KEY DEFAULT cuid()
  code            String    UNIQUE NOT NULL @db.VarChar(6)
  name            String    NOT NULL
  description     String?
  teacherId       String    NOT NULL FOREIGN KEY → Teacher.id
  isActive        Boolean   DEFAULT true
  createdAt       DateTime  DEFAULT now()
  updatedAt       DateTime  DEFAULT now()
  lastActivityAt  DateTime  DEFAULT now()

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

  -- Indexes
  INDEX idx_room_code (code)
  INDEX idx_room_teacher (teacherId, isActive)
  INDEX idx_room_activity (lastActivityAt DESC) WHERE isActive = true
)
```

### Student Table

```
CREATE TABLE Student (
  id          String    PRIMARY KEY DEFAULT cuid()
  name        String    NOT NULL
  roomId      String    NOT NULL FOREIGN KEY → Room.id
  totalPoints Int       DEFAULT 0
  isOnline    Boolean   DEFAULT false
  lastActive  DateTime  DEFAULT now()

  -- Relations
  room            Room            @relation(fields: [roomId], references: [id], onDelet
e: Cascade)
  participations  Participation[]

  -- Constraints
  UNIQUE(name, roomId)  -- Unique name per room

  -- Indexes
  INDEX idx_student_room (roomId, totalPoints DESC, name)
  INDEX idx_student_activity (lastActive DESC)
)
```

### Session Table

```
CREATE TABLE Session (
  id          String    PRIMARY KEY DEFAULT cuid()
  name        String    NOT NULL
  roomId      String    NOT NULL FOREIGN KEY → Room.id
  isActive    Boolean   DEFAULT true
  createdAt   DateTime  DEFAULT now()
  endedAt     DateTime?

  -- Relations
  room            Room            @relation(fields: [roomId], references: [id], onDelet
e: Cascade)
  participations  Participation[]

  -- Indexes
  INDEX idx_session_room (roomId, isActive)
  INDEX idx_session_active (isActive, createdAt DESC)
)
```

### Participation Table

```
CREATE TABLE 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()
  approvedAt  DateTime?
  rejectedAt  DateTime?

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

  -- Indexes
  INDEX idx_participation_pending (status, roomId, submittedAt) WHERE status = 'PENDING
'
  INDEX idx_participation_student (studentId, status, submittedAt DESC)
  INDEX idx_participation_session (sessionId, status)
)

ENUM ParticipationStatus {
  PENDING
  APPROVED
  REJECTED
}
```

## Database Optimizations

#### Enhanced Indexing Strategy

```sql
-- High-frequency authentication queries
CREATE INDEX CONCURRENTLY idx_teacher_email_password
ON teachers (email) WHERE password IS NOT NULL;

-- Room management queries
CREATE INDEX CONCURRENTLY idx_room_teacher_active
ON rooms (teacher_id, is_active, last_activity_at DESC);

-- Student roster queries with points
CREATE INDEX CONCURRENTLY idx_student_room_points
ON students (room_id, total_points DESC, name ASC);

-- Pending submissions queue
CREATE INDEX CONCURRENTLY idx_participation_queue
ON participations (room_id, status, submitted_at ASC)
WHERE status = 'PENDING';

-- Session activity tracking
CREATE INDEX CONCURRENTLY idx_session_activity
ON sessions (room_id, is_active, created_at DESC);
```

**Query Optimization Examples**

```
// Teacher dashboard with room statistics
const getTeacherDashboard = async (teacherId: string) => {
  return await prisma.teacher.findUnique({
    where: { id: teacherId },
    include: {
      rooms: {
        include: {
          _count: {
            select: {
              students: true,
              participations: { where: { status: 'APPROVED' } },
              sessions: true
            }
          }
        },
        orderBy: { lastActivityAt: 'desc' }
      }
    }
  })
}

// Room deletion with cascade information
const getRoomDeletionInfo = async (roomId: string) => {
  return await prisma.room.findUnique({
    where: { id: roomId },
    include: {
      _count: {
        select: {
          students: true,
          participations: true,
          sessions: true
        }
      }
    }
  })
}

// CSV student upload with duplicate checking
const uploadStudentsToRoom = async (roomId: string, studentNames: string[]) => {
  return await prisma.$transaction(async (tx) => {
    // Check existing students
    const existing = await tx.student.findMany({
      where: { roomId },
      select: { name: true }
    })

    const existingNames = new Set(existing.map(s => s.name))
    const newNames = studentNames.filter(name => !existingNames.has(name))

    // Create new students
    if (newNames.length > 0) {
      await tx.student.createMany({
        data: newNames.map(name => ({ name, roomId })),
        skipDuplicates: true
      })
    }

    return {
      studentsAdded: newNames.length,
      duplicatesSkipped: studentNames.length - newNames.length
    }
```

```
  })
}
```

# Security Architecture

## Enhanced Authentication Model

### Password Security

```
// Password hashing configuration
const BCRYPT_ROUNDS = 10;  // Industry standard

// Password validation rules
const passwordValidation = {
  minLength: 6,
  requireUppercase: false,  // Keep simple for educational use
  requireNumbers: false,
  requireSpecialChars: false,
  maxLength: 128
}

// Session management
const sessionConfig = {
  storage: 'localStorage',  // Client-side for simplicity
  timeout: 24 * 60 * 60 * 1000,  // 24 hours
  renewOnActivity: true
}
```

### Authorization Layers

1. **Public Access**: Landing pages, documentation
2. **Room Access**: Valid room code required for student entry
3. **Teacher Access**: Email/password authentication for room management
4. **Owner Access**: Teachers can only manage their own rooms

### Input Validation Pipeline

```
// Enhanced validation with Zod schemas
const schemas = {
  teacherAuth: z.object({
    email: z.string().email('Invalid email format'),
    password: z.string().min(6, 'Password must be at least 6 characters'),
    name: z.string().min(1).max(100).optional()
  }),

  roomCreation: z.object({
    name: z.string().min(1).max(100),
    description: z.string().max(500).optional(),
    csvFile: z.instanceof(File).refine(
      file => file.type === 'text/csv' || file.name.endsWith('.csv'),
      'Must be a CSV file'
    )
  }),

  studentSubmission: z.object({
    roomCode: z.string().regex(/^[A-Z0-9]{6}$/, 'Invalid room code'),
    studentId: z.string().cuid('Invalid student ID'),
    points: z.number().int().min(1).max(3)
  })
}
```

### Security Headers and Middleware

```
// Next.js security configuration
const securityHeaders = [
  {
    key: 'X-DNS-Prefetch-Control',
    value: 'on'
  },
  {
    key: 'X-XSS-Protection',
    value: '1; mode=block'
  },
  {
    key: 'X-Frame-Options',
    value: 'SAMEORIGIN'
  },
  {
    key: 'X-Content-Type-Options',
    value: 'nosniff'
  },
  {
    key: 'Content-Security-Policy',
    value: `
      default-src 'self';
      script-src 'self' 'unsafe-inline' 'unsafe-eval';
      style-src 'self' 'unsafe-inline';
      img-src 'self' data: https:;
      font-src 'self';
      connect-src 'self' ws: wss:;
    `.replace(/\s{2,}/g, ' ').trim()
  }
]
```

# Performance Architecture

## Frontend Performance Optimizations

### React/Next.js Optimizations

```
// Component memoization strategy
const StudentList = React.memo(({ students }: StudentListProps) => {
  return useMemo(() =>
    students.map(student => <StudentRow key={student.id} student={student} />),
    [students]
  )
})

// Virtual scrolling for large rosters (50+ students)
const VirtualizedStudentRoster = ({ students }: { students: Student[] }) => {
  return (
    <FixedSizeList
      height={400}
      itemCount={students.length}
      itemSize={60}
    >
      {({ index, style }) => (
        <div style={style}>
          <StudentRow student={students[index]} />
        </div>
      )}
    </FixedSizeList>
  )
}
```

### Bundle Optimization

```
// Dynamic imports for code splitting
const TeacherDashboard = dynamic(() => import('./components/TeacherDashboard'), {
  loading: () => <LoadingSpinner />,
  ssr: false
})

const PresentationView = dynamic(() => import('./components/PresentationView'), {
  loading: () => <LoadingSpinner />
})
```

## Backend Performance

### Database Connection Management

```
// Prisma configuration for production
const prisma = new PrismaClient({
  datasources: {
    db: {
      url: process.env.DATABASE_URL + '?connection_limit=20&pool_timeout=20'
    }
  },
  log: process.env.NODE_ENV === 'development' ? ['query', 'error', 'warn'] : ['error']
})
```

### Caching Strategy

```
// Memory cache for frequently accessed data
class MemoryCache {
  private cache = new Map<string, { data: any, expiry: number }>()

  set(key: string, data: any, ttlMs: number = 300000) { // 5 minutes default
    this.cache.set(key, {
      data,
      expiry: Date.now() + ttlMs
    })
  }

  get(key: string) {
    const cached = this.cache.get(key)
    if (cached && cached.expiry > Date.now()) {
      return cached.data
    }
    this.cache.delete(key)
    return null
  }
}

// Usage for room data
const roomCache = new MemoryCache()
const getCachedRoomData = async (roomCode: string) => {
  const cached = roomCache.get(`room:${roomCode}`)
  if (cached) return cached

  const roomData = await prisma.room.findUnique({
    where: { code: roomCode },
    include: { students: true, sessions: { where: { isActive: true } } }
  })

  roomCache.set(`room:${roomCode}`, roomData, 120000) // 2 minutes
  return roomData
}
```

## Scalability Considerations

### Load Balancing Strategy

```
// Stateless API design for horizontal scaling
interface ServerConfig {
  maxConcurrentRooms: 100
  maxStudentsPerRoom: 50
  maxConcurrentConnections: 2000
  requestRateLimits: {
    auth: '10/minute'
    submission: '20/minute'
    roomCreation: '5/hour'
  }
}
```

### WebSocket Scaling

```
// Connection management for multiple server instances
class ConnectionManager {
  private connections = new Map<string, Set<WebSocket>>()

  addToRoom(roomCode: string, ws: WebSocket) {
    if (!this.connections.has(roomCode)) {
      this.connections.set(roomCode, new Set())
    }
    this.connections.get(roomCode)?.add(ws)
  }

  broadcastToRoom(roomCode: string, data: any) {
    const roomConnections = this.connections.get(roomCode)
    if (roomConnections) {
      roomConnections.forEach(ws => {
        if (ws.readyState === WebSocket.OPEN) {
          ws.send(JSON.stringify(data))
        }
      })
    }
  }

  cleanup() {
    // Remove stale connections every 5 minutes
    setInterval(() => {
      this.connections.forEach((connections, roomCode) => {
        connections.forEach(ws => {
          if (ws.readyState !== WebSocket.OPEN) {
            connections.delete(ws)
          }
        })
        if (connections.size === 0) {
          this.connections.delete(roomCode)
        }
      })
    }, 300000)
  }
}
```

# Monitoring and Observability

## Application Metrics

### Key Performance Indicators

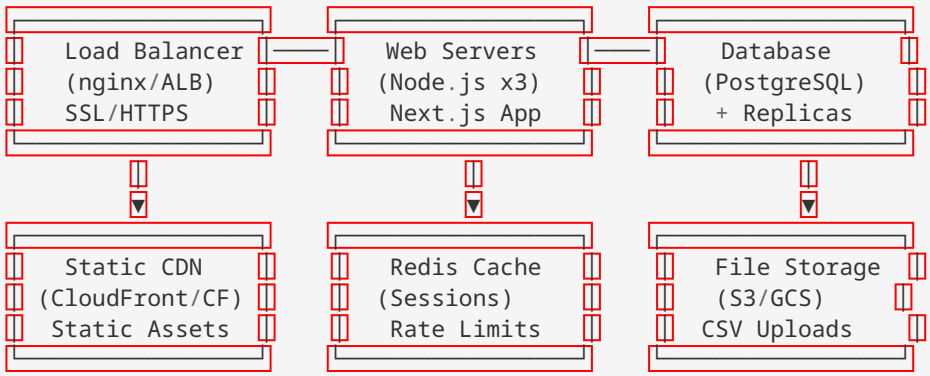
```
interface ApplicationMetrics {
  // User engagement
  activeRooms: number
  concurrentUsers: number
  dailyActiveTeachers: number

  // System performance
  averageResponseTime: number
  errorRate: number
  websocketConnections: number

  // Business metrics
  participationsPerHour: number
  approvalRate: number
  csvUploadSuccess: number

  // Security metrics
  failedLoginAttempts: number
  rateLimitHits: number
  suspiciousActivity: number
}
```

**Logging Strategy**

```
import winston from 'winston'

const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({
      filename: 'logs/error.log',
      level: 'error',
      maxsize: 5242880, // 5MB
      maxFiles: 5
    }),
    new winston.transports.File({
      filename: 'logs/combined.log',
      maxsize: 5242880,
      maxFiles: 5
    }),
    ...(process.env.NODE_ENV !== 'production' ? [
      new winston.transports.Console({
        format: winston.format.simple()
      })
    ] : [])
  ]
})

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

logger.warn('Room deletion attempted', {
  roomId,
  teacherId,
  studentsAffected: room._count.students,
  participationsAffected: room._count.participations
})

logger.error('CSV upload failed', error, {
  teacherId,
  roomId,
  fileName: file.name,
  fileSize: file.size
})
```

# Deployment Architecture

## Production Environment




## Infrastructure Requirements

**Minimum Production Setup**
- **Web Servers**: 2 CPU cores, 4GB RAM per instance
- **Database**: PostgreSQL 14+ with 4 CPU cores, 8GB RAM
- **Redis Cache**: 1GB RAM for session storage and rate limiting
- **Load Balancer**: SSL termination, health checks
- **File Storage**: S3-compatible storage for CSV uploads

**Recommended Production Setup**
- **Web Servers**: 3+ instances behind load balancer
- **Database**: Primary + read replica, automated backups
- **Redis Cluster**: High availability configuration
- **Monitoring**: Application and infrastructure monitoring
- **CDN**: Global content delivery for static assets

This architecture supports the enhanced application requirements while maintaining scalability, security, and performance for classroom environments with 50+ concurrent rooms and 2,000+ simultaneous users.