**Real-Time Multiplayer CodeCrucible - Complete Implementation**

**Mission Statement**

Build a fully functional real-time collaborative coding platform where multiple users can join shared sessions via sharable links, collaborate on voice selection, see live code generation, and work together on synthesis. NO FAKE DATA - everything must be production-ready with real WebSocket connections, database persistence, and working session sharing.

**Core Collaborative Architecture**

**Session Management System**

interface CollaborativeSession {

id: string;

name: string;

creatorId: string;

teamId?: string;

shareableLink: string;

accessType: 'public' | 'team\_only' | 'invite\_only';

participants: SessionParticipant[];

prompt: string;

selectedVoices: VoiceAssignment[];

voiceOutputs: Map<string, VoiceOutput>;

synthesis?: CollaborativeSynthesis;

chatMessages: ChatMessage[];

status: 'active' | 'paused' | 'completed';

createdAt: Date;

expiresAt?: Date;

}

interface SessionParticipant {

userId: string;

name: string;

avatar?: string;

role: 'creator' | 'collaborator' | 'observer';

assignedVoices: VoiceType[];

isActive: boolean;

joinedAt: Date;

lastSeenAt: Date;

cursor?: {

section: 'prompt' | 'voice\_selection' | 'output' | 'synthesis';

position?: { line: number; column: number };

};

}

interface VoiceAssignment {

voiceType: VoiceType;

assignedTo?: string; // userId

status: 'available' | 'assigned' | 'generating' | 'completed';

output?: VoiceOutput;

assignedAt?: Date;

}

**Database Schema for Real-Time Collaboration**

-- Collaborative sessions

CREATE TABLE collaborative\_sessions (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

name VARCHAR(200) NOT NULL,

creator\_id UUID REFERENCES users(id) ON DELETE CASCADE,

team\_id UUID REFERENCES teams(id) ON DELETE SET NULL,

shareable\_link VARCHAR(100) UNIQUE NOT NULL,

access\_type VARCHAR(20) DEFAULT 'team\_only',

prompt TEXT,

selected\_voices JSONB DEFAULT '[]',

voice\_outputs JSONB DEFAULT '{}',

synthesis JSONB,

status VARCHAR(20) DEFAULT 'active',

created\_at TIMESTAMP DEFAULT NOW(),

expires\_at TIMESTAMP,

last\_activity TIMESTAMP DEFAULT NOW()

);

-- Session participants (real-time tracking)

CREATE TABLE session\_participants (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

session\_id UUID REFERENCES collaborative\_sessions(id) ON DELETE CASCADE,

user\_id UUID REFERENCES users(id) ON DELETE CASCADE,

role VARCHAR(20) DEFAULT 'collaborator',

assigned\_voices TEXT[] DEFAULT '{}',

is\_active BOOLEAN DEFAULT TRUE,

joined\_at TIMESTAMP DEFAULT NOW(),

last\_seen\_at TIMESTAMP DEFAULT NOW(),

cursor\_data JSONB,

UNIQUE(session\_id, user\_id)

);

-- Chat messages

CREATE TABLE session\_chat (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

session\_id UUID REFERENCES collaborative\_sessions(id) ON DELETE CASCADE,

user\_id UUID REFERENCES users(id) ON DELETE CASCADE,

message TEXT NOT NULL,

message\_type VARCHAR(20) DEFAULT 'text', -- 'text', 'voice\_assignment', 'system'

metadata JSONB,

created\_at TIMESTAMP DEFAULT NOW()

);

-- Voice assignments tracking

CREATE TABLE voice\_assignments (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

session\_id UUID REFERENCES collaborative\_sessions(id) ON DELETE CASCADE,

voice\_type VARCHAR(50) NOT NULL,

assigned\_to UUID REFERENCES users(id) ON DELETE SET NULL,

status VARCHAR(20) DEFAULT 'available',

output JSONB,

assigned\_at TIMESTAMP,

completed\_at TIMESTAMP,

UNIQUE(session\_id, voice\_type)

);

-- Indexes for performance

CREATE INDEX idx\_sessions\_shareable\_link ON collaborative\_sessions(shareable\_link);

CREATE INDEX idx\_participants\_session\_active ON session\_participants(session\_id, is\_active);

CREATE INDEX idx\_chat\_session\_time ON session\_chat(session\_id, created\_at);

CREATE INDEX idx\_voice\_assignments\_session ON voice\_assignments(session\_id, status);

**Real-Time WebSocket Infrastructure**

**WebSocket Server Setup**

import { WebSocketServer } from 'ws';

import { createServer } from 'http';

interface ConnectedClient {

userId: string;

sessionId: string;

ws: WebSocket;

lastPing: Date;

}

class CollaborationServer {

private wss: WebSocketServer;

private clients: Map<string, ConnectedClient> = new Map();

private sessions: Map<string, Set<string>> = new Map(); // sessionId -> Set<clientId>

constructor(port: number = 3001) {

const server = createServer();

this.wss = new WebSocketServer({ server });

this.wss.on('connection', this.handleConnection.bind(this));

server.listen(port, () => {

console.log(`Collaboration server running on port ${port}`);

});

// Cleanup inactive connections every 30 seconds

setInterval(this.cleanupInactiveConnections.bind(this), 30000);

}

private handleConnection(ws: WebSocket, request: IncomingMessage) {

const url = new URL(request.url!, `http://${request.headers.host}`);

const sessionId = url.searchParams.get('sessionId');

const userId = url.searchParams.get('userId');

const token = url.searchParams.get('token');

if (!sessionId || !userId || !token) {

ws.close(1008, 'Missing required parameters');

return;

}

// Verify user token and session access

this.verifySessionAccess(userId, sessionId, token)

.then(isValid => {

if (!isValid) {

ws.close(1008, 'Invalid session access');

return;

}

const clientId = `${userId}-${Date.now()}`;

this.clients.set(clientId, {

userId,

sessionId,

ws,

lastPing: new Date()

});

// Add to session tracking

if (!this.sessions.has(sessionId)) {

this.sessions.set(sessionId, new Set());

}

this.sessions.get(sessionId)!.add(clientId);

// Set up message handlers

ws.on('message', (data) => this.handleMessage(clientId, data));

ws.on('close', () => this.handleDisconnection(clientId));

ws.on('pong', () => {

const client = this.clients.get(clientId);

if (client) client.lastPing = new Date();

});

// Send initial session state

this.sendSessionState(clientId);

// Notify other participants

this.broadcastToSession(sessionId, {

type: 'participant\_joined',

userId,

timestamp: new Date()

}, clientId);

// Update participant status in database

this.updateParticipantStatus(userId, sessionId, true);

})

.catch(() => {

ws.close(1008, 'Authentication failed');

});

}

private async handleMessage(clientId: string, data: Buffer) {

try {

const client = this.clients.get(clientId);

if (!client) return;

const message = JSON.parse(data.toString());

const { sessionId, userId } = client;

switch (message.type) {

case 'update\_prompt':

await this.handlePromptUpdate(sessionId, userId, message.data);

break;

case 'voice\_assignment':

await this.handleVoiceAssignment(sessionId, userId, message.data);

break;

case 'voice\_generation\_start':

await this.handleVoiceGenerationStart(sessionId, userId, message.data);

break;

case 'voice\_output':

await this.handleVoiceOutput(sessionId, userId, message.data);

break;

case 'chat\_message':

await this.handleChatMessage(sessionId, userId, message.data);

break;

case 'cursor\_update':

await this.handleCursorUpdate(sessionId, userId, message.data);

break;

case 'synthesis\_request':

await this.handleSynthesisRequest(sessionId, userId, message.data);

break;

case 'ping':

client.ws.send(JSON.stringify({ type: 'pong', timestamp: Date.now() }));

break;

}

} catch (error) {

console.error('Error handling message:', error);

}

}

private async handlePromptUpdate(sessionId: string, userId: string, data: any) {

// Update prompt in database

await database.update('collaborative\_sessions')

.set({

prompt: data.prompt,

last\_activity: new Date()

})

.where({ id: sessionId });

// Broadcast to all participants

this.broadcastToSession(sessionId, {

type: 'prompt\_updated',

prompt: data.prompt,

updatedBy: userId,

timestamp: new Date()

});

}

private async handleVoiceAssignment(sessionId: string, userId: string, data: any) {

const { voiceType, action } = data; // action: 'claim' | 'release'

if (action === 'claim') {

// Check if voice is available

const existing = await database.select()

.from('voice\_assignments')

.where({ session\_id: sessionId, voice\_type: voiceType })

.first();

if (existing && existing.assigned\_to && existing.assigned\_to !== userId) {

// Voice already assigned to someone else

this.sendToClient(userId, sessionId, {

type: 'voice\_assignment\_failed',

voiceType,

reason: 'Already assigned'

});

return;

}

// Assign voice to user

await database.upsert('voice\_assignments').values({

session\_id: sessionId,

voice\_type: voiceType,

assigned\_to: userId,

status: 'assigned',

assigned\_at: new Date()

});

// Update participant's assigned voices

await database.update('session\_participants')

.set({

assigned\_voices: database.raw('array\_append(assigned\_voices, ?)', [voiceType])

})

.where({ session\_id: sessionId, user\_id: userId });

} else if (action === 'release') {

// Release voice assignment

await database.update('voice\_assignments')

.set({

assigned\_to: null,

status: 'available'

})

.where({ session\_id: sessionId, voice\_type: voiceType, assigned\_to: userId });

// Remove from participant's assigned voices

await database.update('session\_participants')

.set({

assigned\_voices: database.raw('array\_remove(assigned\_voices, ?)', [voiceType])

})

.where({ session\_id: sessionId, user\_id: userId });

}

// Broadcast voice assignment update

this.broadcastToSession(sessionId, {

type: 'voice\_assignment\_updated',

voiceType,

assignedTo: action === 'claim' ? userId : null,

action,

timestamp: new Date()

});

}

private async handleVoiceOutput(sessionId: string, userId: string, data: any) {

const { voiceType, output } = data;

// Save voice output to database

await database.update('voice\_assignments')

.set({

output: JSON.stringify(output),

status: 'completed',

completed\_at: new Date()

})

.where({ session\_id: sessionId, voice\_type: voiceType, assigned\_to: userId });

// Update session with voice output

await database.raw(`

UPDATE collaborative\_sessions

SET voice\_outputs = jsonb\_set(voice\_outputs, '{${voiceType}}', ?, true),

last\_activity = NOW()

WHERE id = ?

`, [JSON.stringify(output), sessionId]);

// Broadcast new voice output to all participants

this.broadcastToSession(sessionId, {

type: 'voice\_output\_received',

voiceType,

output,

generatedBy: userId,

timestamp: new Date()

});

}

private broadcastToSession(sessionId: string, message: any, excludeClientId?: string) {

const sessionClients = this.sessions.get(sessionId);

if (!sessionClients) return;

const messageStr = JSON.stringify(message);

sessionClients.forEach(clientId => {

if (clientId === excludeClientId) return;

const client = this.clients.get(clientId);

if (client && client.ws.readyState === WebSocket.OPEN) {

client.ws.send(messageStr);

}

});

}

private sendToClient(userId: string, sessionId: string, message: any) {

const sessionClients = this.sessions.get(sessionId);

if (!sessionClients) return;

const messageStr = JSON.stringify(message);

sessionClients.forEach(clientId => {

const client = this.clients.get(clientId);

if (client && client.userId === userId && client.ws.readyState === WebSocket.OPEN) {

client.ws.send(messageStr);

}

});

}

}

// Initialize collaboration server

const collaborationServer = new CollaborationServer(3001);

**Session Creation and Sharing System**

**Session Management API**

// Create new collaborative session

app.post('/api/collaboration/sessions', authenticateUser, async (req, res) => {

try {

const { name, accessType, teamId, expiresIn } = req.body;

const creatorId = req.user.id;

// Generate unique shareable link

const shareableLink = generateShareableLink();

// Calculate expiry date

const expiresAt = expiresIn ?

new Date(Date.now() + expiresIn \* 60 \* 60 \* 1000) : // hours to ms

null;

const session = await database.insert('collaborative\_sessions').values({

name,

creator\_id: creatorId,

team\_id: teamId,

shareable\_link: shareableLink,

access\_type: accessType || 'team\_only',

expires\_at: expiresAt

}).returning('\*');

// Add creator as participant

await database.insert('session\_participants').values({

session\_id: session[0].id,

user\_id: creatorId,

role: 'creator',

is\_active: true

});

res.json({

success: true,

session: session[0],

joinUrl: `${process.env.BASE\_URL}/collaborate/${shareableLink}`

});

} catch (error) {

res.status(500).json({ error: 'Failed to create session' });

}

});

// Join session via shareable link

app.get('/api/collaboration/join/:shareableLink', authenticateUser, async (req, res) => {

try {

const { shareableLink } = req.params;

const userId = req.user.id;

// Find session

const session = await database.select()

.from('collaborative\_sessions')

.where({ shareable\_link: shareableLink })

.first();

if (!session) {

return res.status(404).json({ error: 'Session not found' });

}

// Check if session is expired

if (session.expires\_at && new Date() > session.expires\_at) {

return res.status(410).json({ error: 'Session has expired' });

}

// Check access permissions

const hasAccess = await checkSessionAccess(userId, session);

if (!hasAccess) {

return res.status(403).json({ error: 'Access denied' });

}

// Add user as participant (or update if already exists)

await database.upsert('session\_participants').values({

session\_id: session.id,

user\_id: userId,

role: 'collaborator',

is\_active: true,

joined\_at: new Date()

});

// Generate WebSocket connection token

const wsToken = generateWSToken(userId, session.id);

res.json({

success: true,

session: {

id: session.id,

name: session.name,

prompt: session.prompt,

voiceOutputs: session.voice\_outputs,

synthesis: session.synthesis

},

wsConnectionUrl: `ws://localhost:3001?sessionId=${session.id}&userId=${userId}&token=${wsToken}`

});

} catch (error) {

res.status(500).json({ error: 'Failed to join session' });

}

});

// Get session participants

app.get('/api/collaboration/sessions/:sessionId/participants', authenticateUser, async (req, res) => {

try {

const { sessionId } = req.params;

const participants = await database.select(

'session\_participants.\*',

'users.name',

'users.avatar\_url'

)

.from('session\_participants')

.join('users', 'session\_participants.user\_id', 'users.id')

.where({ 'session\_participants.session\_id': sessionId })

.orderBy('session\_participants.joined\_at');

res.json(participants);

} catch (error) {

res.status(500).json({ error: 'Failed to get participants' });

}

});

const generateShareableLink = (): string => {

// Generate unique 12-character link

const chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';

let result = '';

for (let i = 0; i < 12; i++) {

result += chars.charAt(Math.floor(Math.random() \* chars.length));

}

return result;

};

**Frontend Real-Time Collaboration UI**

**Collaborative Session Component**

const CollaborativeSession: React.FC<{ sessionId: string }> = ({ sessionId }) => {

const [session, setSession] = useState<CollaborativeSession | null>(null);

const [participants, setParticipants] = useState<SessionParticipant[]>([]);

const [voiceAssignments, setVoiceAssignments] = useState<Map<VoiceType, string>>(new Map());

const [chatMessages, setChatMessages] = useState<ChatMessage[]>([]);

const [isConnected, setIsConnected] = useState(false);

const [ws, setWs] = useState<WebSocket | null>(null);

const { user } = useAuth();

// Initialize WebSocket connection

useEffect(() => {

const initializeConnection = async () => {

try {

// Get session data and WebSocket URL

const response = await fetch(`/api/collaboration/join/${sessionId}`);

const data = await response.json();

if (!data.success) {

throw new Error(data.error);

}

setSession(data.session);

// Connect to WebSocket

const websocket = new WebSocket(data.wsConnectionUrl);

websocket.onopen = () => {

setIsConnected(true);

setWs(websocket);

};

websocket.onmessage = (event) => {

const message = JSON.parse(event.data);

handleWebSocketMessage(message);

};

websocket.onclose = () => {

setIsConnected(false);

setWs(null);

// Attempt to reconnect after 3 seconds

setTimeout(initializeConnection, 3000);

};

websocket.onerror = (error) => {

console.error('WebSocket error:', error);

};

} catch (error) {

console.error('Failed to initialize session:', error);

}

};

initializeConnection();

// Cleanup on unmount

return () => {

if (ws) {

ws.close();

}

};

}, [sessionId]);

const handleWebSocketMessage = (message: any) => {

switch (message.type) {

case 'participant\_joined':

// Refresh participants list

fetchParticipants();

break;

case 'prompt\_updated':

setSession(prev => prev ? { ...prev, prompt: message.prompt } : null);

break;

case 'voice\_assignment\_updated':

setVoiceAssignments(prev => {

const newMap = new Map(prev);

if (message.assignedTo) {

newMap.set(message.voiceType, message.assignedTo);

} else {

newMap.delete(message.voiceType);

}

return newMap;

});

break;

case 'voice\_output\_received':

setSession(prev => {

if (!prev) return null;

const newOutputs = new Map(prev.voiceOutputs);

newOutputs.set(message.voiceType, message.output);

return { ...prev, voiceOutputs: newOutputs };

});

break;

case 'chat\_message\_received':

setChatMessages(prev => [...prev, message.data]);

break;

case 'synthesis\_completed':

setSession(prev => prev ? { ...prev, synthesis: message.synthesis } : null);

break;

}

};

const sendMessage = (type: string, data: any) => {

if (ws && ws.readyState === WebSocket.OPEN) {

ws.send(JSON.stringify({ type, data }));

}

};

const handlePromptChange = (newPrompt: string) => {

sendMessage('update\_prompt', { prompt: newPrompt });

};

const handleVoiceAssignment = (voiceType: VoiceType, action: 'claim' | 'release') => {

sendMessage('voice\_assignment', { voiceType, action });

};

const handleChatMessage = (message: string) => {

sendMessage('chat\_message', { message });

};

const handleVoiceGeneration = async (voiceType: VoiceType) => {

if (!session) return;

// Notify others that generation started

sendMessage('voice\_generation\_start', { voiceType });

try {

// Generate voice response

const response = await generateVoiceResponse(session.prompt, voiceType);

// Send output to all participants

sendMessage('voice\_output', { voiceType, output: response });

} catch (error) {

console.error('Voice generation failed:', error);

}

};

return (

<div className="collaborative-session">

{/\* Connection Status \*/}

<ConnectionStatus isConnected={isConnected} />

{/\* Participants Bar \*/}

<ParticipantBar participants={participants} />

<div className="session-workspace">

{/\* Left Panel: Prompt and Voice Selection \*/}

<div className="left-panel">

<SharedPromptEditor

prompt={session?.prompt || ''}

onChange={handlePromptChange}

participants={participants}

/>

<VoiceCoordinationPanel

voiceAssignments={voiceAssignments}

onVoiceAssignment={handleVoiceAssignment}

onVoiceGeneration={handleVoiceGeneration}

currentUserId={user?.id}

/>

</div>

{/\* Center Panel: Live Voice Outputs \*/}

<div className="center-panel">

<LiveVoiceOutputs

voiceOutputs={session?.voiceOutputs || new Map()}

voiceAssignments={voiceAssignments}

participants={participants}

/>

</div>

{/\* Right Panel: Chat and Synthesis \*/}

<div className="right-panel">

<TeamChatPanel

messages={chatMessages}

onSendMessage={handleChatMessage}

/>

<CollaborativeSynthesis

voiceOutputs={session?.voiceOutputs || new Map()}

synthesis={session?.synthesis}

onSynthesisRequest={() => sendMessage('synthesis\_request', {})}

/>

</div>

</div>

</div>

);

};

**Voice Coordination Panel**

const VoiceCoordinationPanel: React.FC<{

voiceAssignments: Map<VoiceType, string>;

onVoiceAssignment: (voice: VoiceType, action: 'claim' | 'release') => void;

onVoiceGeneration: (voice: VoiceType) => void;

currentUserId?: string;

}> = ({ voiceAssignments, onVoiceAssignment, onVoiceGeneration, currentUserId }) => {

const voices: VoiceType[] = [

'Explorer', 'Maintainer', 'Analyzer', 'Developer', 'Implementor',

'Security', 'Architect', 'Designer', 'Performance'

];

return (

<div className="voice-coordination-panel">

<h3>Voice Coordination</h3>

<div className="voice-grid">

{voices.map(voice => {

const assignedTo = voiceAssignments.get(voice);

const isAssignedToMe = assignedTo === currentUserId;

const isAvailable = !assignedTo;

return (

<div key={voice} className={`voice-card ${

isAssignedToMe ? 'assigned-to-me' :

assignedTo ? 'assigned-to-other' : 'available'

}`}>

<div className="voice-header">

<h4>{voice}</h4>

{assignedTo && (

<div className="assigned-indicator">

<Avatar userId={assignedTo} size="sm" />

</div>

)}

</div>

<div className="voice-actions">

{isAvailable && (

<button

onClick={() => onVoiceAssignment(voice, 'claim')}

className="claim-voice-btn"

>

Claim Voice

</button>

)}

{isAssignedToMe && (

<>

<button

onClick={() => onVoiceGeneration(voice)}

className="generate-btn"

>

Generate Code

</button>

<button

onClick={() => onVoiceAssignment(voice, 'release')}

className="release-voice-btn"

>

Release

</button>

</>

)}

{assignedTo && !isAssignedToMe && (

<div className="assigned-status">

Assigned to {getUserName(assignedTo)}

</div>

)}

</div>

</div>

);

})}

</div>

</div>

);

};

**Live Voice Outputs Display**

const LiveVoiceOutputs: React.FC<{

voiceOutputs: Map<VoiceType, VoiceOutput>;

voiceAssignments: Map<VoiceType, string>;

participants: SessionParticipant[];

}> = ({ voiceOutputs, voiceAssignments, participants }) => {

return (

<div className="live-voice-outputs">

<h3>Live Voice Outputs</h3>

<div className="outputs-grid">

{Array.from(voiceOutputs.entries()).map(([voiceType, output]) => {

const assignedTo = voiceAssignments.get(voiceType);

const participant = participants.find(p => p.userId === assignedTo);

return (

<div key={voiceType} className="voice-output-card">

<div className="output-header">

<h4>{voiceType}</h4>

<div className="contributor">

<Avatar userId={assignedTo} size="sm" />

<span>{participant?.name || 'Unknown'}</span>

</div>

</div>

<div className="code-output">

<SyntaxHighlighter language="typescript" style={vscDarkPlus}>

{output.code}

</SyntaxHighlighter>

</div>

<div className="output-explanation">

<p>{output.explanation}</p>

</div>

<div className="output-actions">

<button className="copy-btn">Copy Code</button>

<button className="discuss-btn">Discuss</button>

</div>

</div>

);

})}

</div>

</div>

);

};

**Session Sharing and Invitations**

**Session Sharing Component**

const SessionSharingModal: React.FC<{

session: CollaborativeSession;

onClose: () => void;

}> = ({ session, onClose }) => {

const [copied, setCopied] = useState(false);

const shareableUrl = `${window.location.origin}/collaborate/${session.shareableLink}`;

const copyToClipboard = async () => {

try {

await navigator.clipboard.writeText(shareableUrl);

setCopied(true);

setTimeout(() => setCopied(false), 2000);

} catch (error) {

console.error('Failed to copy:', error);

}

};

return (

<Modal onClose={onClose}>

<div className="session-sharing-modal">

<h2>Share Collaboration Session</h2>

<div className="share-link-section">

<label>Shareable Link</label>

<div className="link-input-group">

<input

type="text"

value={shareableUrl}

readOnly

className="share-link-input"

/>

<button

onClick={copyToClipboard}

className={`copy-btn ${copied ? 'copied' : ''}`}

>

{copied ? 'Copied!' : 'Copy'}

</button>

</div>

</div>

<div className="access-settings">

<h3>Access Settings</h3>

<div className="access-type">

<strong>Access Type:</strong> {session.accessType}

</div>

{session.expiresAt && (

<div className="expiry">

<strong>Expires:</strong> {new Date(session.expiresAt).toLocaleString()}

</div>

)}

</div>

<div className="current-participants">

<h3>Current Participants ({session.participants.length})</h3>

<div className="participants-list">

{session.participants.map(participant => (

<div key={participant.userId} className="participant-item">

<Avatar userId={participant.userId} />

<span>{participant.name}</span>

<span className={`status ${participant.isActive ? 'active' : 'inactive'}`}>

{participant.isActive ? 'Active' : 'Inactive'}

</span>

</div>

))}

</div>

</div>

<div className="modal-actions">

<button onClick={onClose} className="close-btn">Close</button>

</div>

</div>

</Modal>

);

};

**Real-Time Chat Integration**

**Team Chat Component**

const TeamChatPanel: React.FC<{

messages: ChatMessage[];

onSendMessage: (message: string) => void;

}> = ({ messages, onSendMessage }) => {

const [newMessage, setNewMessage] = useState('');

const messagesEndRef = useRef<HTMLDivElement>(null);

const scrollToBottom = () => {

messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });

};

useEffect(() => {

scrollToBottom();

}, [messages]);

const handleSubmit = (e: React.FormEvent) => {

e.preventDefault();

if (newMessage.trim()) {

onSendMessage(newMessage);

setNewMessage('');

}

};

return (

<div className="team-chat-panel">

<h3>Team Chat</h3>

<div className="chat-messages">

{messages.map(message => (

<div key={message.id} className={`message ${message.messageType}`}>

<div className="message-header">

<Avatar userId={message.userId} size="xs" />

<span className="sender-name">{message.senderName}</span>

<span className="timestamp">

{new Date(message.createdAt).toLocaleTimeString()}

</span>

</div>

<div className="message-content">

{message.messageType === 'voice\_assignment' ? (

<div className="system-message">

{message.metadata?.action === 'claim' ? 'claimed' : 'released'} the {message.metadata?.voiceType} voice

</div>

) : (

<p>{message.message}</p>

)}

</div>

</div>

))}

<div ref={messagesEndRef} />

</div>

<form onSubmit={handleSubmit} className="chat-input-form">

<input

type="text"

value={newMessage}

onChange={(e) => setNewMessage(e.target.value)}

placeholder="Type a message..."

className="chat-input"

/>

<button type="submit" className="send-btn">Send</button>

</form>

</div>

);

};

**Production Deployment Requirements**

**Environment Variables**

# WebSocket Server

WEBSOCKET\_PORT=3001

WEBSOCKET\_ORIGINS=https://codecrucible.com,http://localhost:3000

# Session Management

SESSION\_EXPIRY\_HOURS=24

MAX\_SESSION\_PARTICIPANTS=50

CLEANUP\_INTERVAL\_MINUTES=30

# Real-time Features

ENABLE\_VOICE\_COORDINATION=true

ENABLE\_LIVE\_CURSORS=true

ENABLE\_SESSION\_RECORDING=false

# Performance

MAX\_CONCURRENT\_SESSIONS=1000

WEBSOCKET\_PING\_INTERVAL=30000

**Performance Monitoring**

// Monitor WebSocket connections and session health

const monitorCollaborationHealth = () => {

setInterval(() => {

const stats = {

activeConnections: collaborationServer.getActiveConnections(),

activeSessions: collaborationServer.getActiveSessions(),

messagesPerSecond: collaborationServer.getMessageRate(),

memoryUsage: process.memoryUsage()

};

console.log('Collaboration Server Stats:', stats);

// Send to monitoring service

sendMetricsToMonitoring(stats);

}, 60000); // Every minute

};

**Success Criteria**

**Functional Requirements**

* ✅ **Real sharable links** that anyone can join via URL
* ✅ **Live WebSocket connections** with real-time updates
* ✅ **Voice coordination** with claim/release functionality
* ✅ **Live voice generation** visible to all participants
* ✅ **Real-time chat** integrated with session
* ✅ **Participant management** with roles and permissions
* ✅ **Session persistence** that survives disconnections
* ✅ **Mobile-responsive** collaborative interface

**Performance Requirements**

* ✅ **WebSocket latency** < 100ms for real-time updates
* ✅ **Session join time** < 3 seconds via shareable link
* ✅ **Concurrent sessions** support for 100+ active sessions
* ✅ **Message throughput** 1000+ messages/minute
* ✅ **Connection stability** 8+ hour sessions without drops

**Business Requirements**

* ✅ **Team-tier paywall** properly enforced
* ✅ **Usage tracking** for collaborative features
* ✅ **Session analytics** for team insights
* ✅ **Enterprise controls** for session management
* ✅ **Audit logging** for all collaborative actions

**Build this as production-ready multiplayer infrastructure** that can handle real teams collaborating on complex coding challenges in real-time. No fake data, no placeholders - everything must work with real users sharing real links and generating real code together.