Multi-Camera Synchronization App - Repository Analysis

Analysis Date: December 1, 2025
Analyzed Repositories:
1. `NXConner/ex-pose-positions` (Intimacy Companion Suite)
2. `NXConner/size-sync-studio` (PumpGuard Performance Suite)

Executive Summary

Both repositories contain production-ready web applications built with modern React/TypeScript stacks that demonstrate relevant patterns for building a multi-camera synchronization application. The **ex-pose-positions** repository contains a fully functional **Camera Sync Studio** feature that implements real-time multi-device camera coordination using Supabase Realtime and WebRTC, which is directly applicable to your requirements. The **size-sync-studio** repository provides excellent camera utilities, measurement workflows, and Express server patterns.

Repository 1: ex-pose-positions (Intimacy Companion Suite)

Overview

A privacy-first partner experience platform with a comprehensive **Camera Sync Studio** feature that enables multi-angle capture with synchronized recording across multiple devices.

Technology Stack

- **Frontend:** React 19.2, TypeScript, Vite 7.1
- **UI Framework:** Tailwind CSS 4.1
- **Real-time Communication:** Supabase Realtime + WebRTC (peer-to-peer)
- **State Management:** React hooks with refs for WebRTC state
- **Mobile:** Capacitor 7.4 for Android/iOS
- **Backend:** Supabase (PostgreSQL + Realtime + Edge Functions)
- **Testing:** Vitest, Playwright, Testing Library

Key Architecture Patterns

1. Camera Sync Studio Architecture

Located in:
- `src/components/camera-sync.tsx` - UI component
- `src/hooks/use-camera-session.ts` - Core synchronization logic

Core Features:
- Multi-device camera coordination over local/remote network
- Real-time presence tracking (who’s online)

- WebRTC peer-to-peer video streaming
- Synchronized recording with countdown
- Automatic video download after recording
- Session sharing via join links

2. Real-time Synchronization Pattern

Supabase Realtime Channel Setup:

```
const channel = supabase.channel(`camera_sync:${sessionId}`, {
  config: {
    presence: {
      key: participantIdRef.current,
    },
  },
});

channel
  .on("presence", { event: "sync" }, handlePresenceSync)
  .on("broadcast", { event: "signal" }, ({ payload }) => {
    processSignal(payload as SignalPayload);
  })
  .on("broadcast", { event: "command" }, ({ payload }) => {
    handleCommand(payload as CommandPayload);
  });
```

Key Concepts:
- **Presence API:** Tracks which devices are online in real-time
- **Broadcast Events:**
- `signal` - WebRTC signaling (offer/answer/ICE candidates)
- `command` - Synchronized actions (start/stop recording)
- **Session-based rooms:** Each session has unique ID for isolation

3. WebRTC Peer-to-Peer Video Streaming

Connection Management:

```
const ensurePeer = (remoteId: string) => {
  let peer = new RTCPeerConnection(ICE_SERVERS);

  peer.onicecandidate = (event) => {
    // Send ICE candidates via Supabase broadcast
  };

  peer.ontrack = (event) => {
    // Receive remote video streams
    const [stream] = event.streams;
    remoteStreamsRef.current.set(stream.id, {
      stream,
      participantId: remoteId,
      label,
    });
  };

  // Add local camera tracks to peer connection
  localStreamsRef.current.forEach(({ descriptor }) => {
    descriptor.stream.getTracks().forEach((track) =>
      peer.addTrack(track, descriptor.stream)
    );
  });

  return peer;
};
```

Signaling Flow:

1. Devices join Supabase channel with presence
2. Higher participant ID initiates WebRTC offer
3. Signaling messages (offer/answer/ICE) sent via Supabase broadcast
4. Direct peer-to-peer video streams established
5. Remote streams displayed in real-time

4. Synchronized Recording Pattern

Countdown-based Synchronization:

```
const startSyncedRecording = (leadTimeMs = 3_000) => {
  const startAt = Date.now() + leadTimeMs;

  // Broadcast start command to all devices
  channel.send({
    type: "broadcast",
    event: "command",
    payload: {
      action: "start",
      from: participantIdRef.current,
      startAt,
      issuedAt: Date.now(),
    },
  });

  // Schedule local recording
  scheduleRecording(startAt);
};
```

Recording Implementation:
- Uses `MediaRecorder` API with VP9/Opus codec
- Records both local and remote streams simultaneously
- Automatic download as WebM files on stop
- Countdown UI shows time until recording starts

5. Multi-Camera Management

Device Enumeration & Selection:

```
const refreshDevices = async () => {
  const all = await navigator.mediaDevices.enumerateDevices();
  const cams = all
    .filter((device) => device.kind === "videoinput")
    .map((device, index) => ({
      deviceId: device.deviceId,
      label: device.label || `Camera ${index + 1}`,
    }));
  setDevices(cams);
};
```

Multiple Local Cameras:
- Supports multiple cameras on same device
- Each camera gets unique ID and label
- First camera includes audio, others video-only
- Tracks managed with refs to prevent re-renders

6. Session Sharing Pattern

Join Link Generation:

```
function buildShareLink(sessionId: string | null): string | null {
  if (!sessionId) return null;
  const { origin, pathname } = window.location;
  return `${origin}${pathname}?cameraSession=${encodeURIComponent(sessionId)}`;
}
```

Session Persistence:
- Session ID stored in localStorage
- Can be shared via URL parameter
- Automatic reconnection on page reload

Relevant Code Patterns for Multi-Camera App

Pattern 1: Participant Management

```
interface ParticipantInfo {
  id: string;
  label: string;
  isSelf: boolean;
  connected: boolean;
}

const handlePresenceSync = () => {
  const state = channel.presenceState();
  const entries: ParticipantInfo[] = [];
  Object.entries(state).forEach(([key, sessions]) => {
    entries.push({
      id: key,
      label: sessions[0]?.displayName || `Participant ${key.slice(0, 6)}`,
      isSelf: key === participantIdRef.current,
      connected: true,
    });
  });
  setParticipants(entries);
};
```

Pattern 2: Stream Lifecycle Management

```
// Cleanup on unmount
useEffect(() => {
  return () => {
    // Stop all local streams
    localStreamsRef.current.forEach(({ descriptor }) => {
      descriptor.stream.getTracks().forEach(track => track.stop());
    });

    // Close all peer connections
    peersRef.current.forEach(peer => peer.close());

    // Unsubscribe from channel
    channel.unsubscribe();
  };
}, []);
```

Pattern 3: Error Handling & Fallbacks

```
const attachCamera = async (deviceId: string) => {
  try {
    const constraints: MediaStreamConstraints = {
      video: deviceId === "default"
        ? { width: { ideal: 1280 }, height: { ideal: 720 } }
        : { deviceId: { exact: deviceId } },
      audio: includeAudio,
    };

    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    // ... success handling
  } catch (error) {
    setErrorMessage(
      error instanceof Error
        ? error.message
        : "Unable to access the selected camera."
    );
  }
};
```

UI/UX Patterns

Real-time Status Indicators:
- Connection status (online/offline)
- Recording state with countdown
- Participant presence indicators
- Camera feed labels (local/remote, primary)

Responsive Grid Layout:
- 2-column grid for video tiles on desktop
- Stacked layout on mobile
- Empty state messaging
- Muted indicator for local feeds

Supabase Configuration

Client Setup:

```
browserClient = createClient(url, anonKey, {
  auth: {
    persistSession: false,
    autoRefreshToken: false,
  },
  realtime: {
    params: {
      eventsPerSecond: 15,
    },
  },
});
```

Environment Variables:
- `VITE_SUPABASE_URL` - Supabase project URL
- `VITE_SUPABASE_ANON_KEY` - Anonymous/public key

Repository 2: size-sync-studio (PumpGuard Performance Suite)

Overview

A safety-first coaching and analytics platform with sophisticated camera utilities, measurement workflows, and Express backend. While it doesn't have multi-camera sync, it provides excellent patterns for camera handling and server architecture.

Technology Stack

- **Frontend:** React 18.3, TypeScript, Vite 5.4
- **UI Framework:** Tailwind CSS 3.4, shadcn/ui, Radix UI
- **State Management:** TanStack Query 5.83 (React Query)
- **Backend:** Express 4.19 with Helmet, CORS, rate limiting
- **Database:** Supabase (PostgreSQL)
- **Mobile:** Capacitor 7.4 for Android
- **Testing:** Vitest, Playwright
- **Logging:** Pino (structured logging)

Key Architecture Patterns

1. Advanced Camera Utilities

Located in: `src/utils/camera.ts`

Features:
- Device enumeration with facing mode detection
- Constraint building with fallbacks
- Zoom, torch, frame rate control
- FPS measurement
- Mobile-optimized hints (continuous focus/white balance)

Smart Device Selection:

```
const selectDeviceForFacing = async (
  facing: CameraFacingMode,
): Promise<string | undefined> => {
  const devices = await enumerateVideoDevices();

  // Infer front/back from labels
  const backTokens = ["back", "rear", "environment"];
  const frontTokens = ["front", "user", "face"];

  const candidates = devices.filter((d) =>
    facing === "environment"
      ? labelIncludes(d.label, backTokens)
      : labelIncludes(d.label, frontTokens),
  );

  if (candidates.length > 0) return candidates[0].deviceId;

  // Fallback: pick first/last device
  return facing === "environment"
    ? devices[devices.length - 1].deviceId
    : devices[0].deviceId;
};
```

Constraint Building with Fallbacks:

```
const startCamera = async (opts: CameraStartOptions): Promise<CameraStartResult> => {
  const candidates: MediaStreamConstraints[] = [];

  // Try exact deviceId first
  if (opts.deviceId) candidates.push(buildConstraints(opts));

  // Try inferred device for facing mode
  if (!opts.deviceId && opts.facingMode) {
    const deviceId = await selectDeviceForFacing(opts.facingMode);
    if (deviceId) candidates.push(buildConstraints({ ...opts, deviceId }));
  }

  // Generic fallback
  candidates.push(buildConstraints({ ...opts, deviceId: undefined }));

  // Try each candidate until one succeeds
  for (const c of candidates) {
    try {
      const stream = await navigator.mediaDevices.getUserMedia(c);
      return { stream, track, capabilities };
    } catch (err) {
      lastError = err;
    }
  }

  throw lastError;
};
```

Camera Capabilities Detection:

```
const capabilities: CameraCapabilities = {
  canTorch: !!capsAny.torch,
  canZoom: !!capsAny.zoom,
  zoom: capsAny.zoom
    ? { min: Number(capsAny.zoom.min ?? 1), max: Number(capsAny.zoom.max ?? 1) }
    : undefined,
  frameRate: capsAny.frameRate
    ? { min: capsAny.frameRate.min, max: capsAny.frameRate.max }
    : undefined,
};
```

2. Express Server Architecture

Located in: `server/`

Server Setup (server/index.js):

```
import { app } from './app.js';
import { config } from './config.js';

const port = config.PORT;
app.listen(port, () => {
  console.log(`[server] listening on http://localhost:${port}`);
});
```

App Configuration (server/app.js):
- Helmet for security headers
- CORS configuration
- Rate limiting
- Compression
- Structured logging with Pino
- OpenAPI/Swagger documentation
- Health check endpoints

Configuration Pattern (server/config.js):

```
export const config = {
  NODE_ENV: process.env.NODE_ENV || 'development',
  PORT: parseInt(process.env.PORT || '3001', 10),
  API_PREFIX: process.env.API_PREFIX || '/api',
  WEB_ORIGIN: process.env.WEB_ORIGIN || 'http://localhost:8080',
  SUPABASE_URL: process.env.SUPABASE_URL,
  SUPABASE_SERVICE_ROLE_KEY: process.env.SUPABASE_SERVICE_ROLE_KEY,
};
```

3. Component Architecture

Modular Component Structure:

```
src/
├── components/
│   ├── ui/              # Reusable UI primitives (shadcn/ui)
│   ├── measure/         # Measurement-specific components
│   ├── common/          # Shared components (ErrorBoundary, LoadingSkeleton)
│   └── health/          # Health tracking components
├── pages/               # Route-level components
├── hooks/               # Custom React hooks
├── utils/               # Utility functions
├── lib/                 # Config, design tokens, clients
├── types/               # TypeScript type definitions
└── integrations/        # External service integrations
```

4. State Management with TanStack Query

Data Fetching Pattern:

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';

// Query for fetching data
const { data, isLoading, error } = useQuery({
  queryKey: ['sessions', userId],
  queryFn: () => fetchSessions(userId),
  staleTime: 5 * 60 * 1000, // 5 minutes
});

// Mutation for updates
const mutation = useMutation({
  mutationFn: createSession,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ['sessions'] });
  },
});
```

Persistence:

```
import { persistQueryClient } from '@tanstack/react-query-persist-client';
import { createSyncStoragePersister } from '@tanstack/query-sync-storage-persister';

const persister = createSyncStoragePersister({
  storage: window.localStorage,
});

persistQueryClient({
  queryClient,
  persister,
});
```

5. Mobile Integration with Capacitor

Capacitor Configuration (capacitor.config.ts):

```
import { CapacitorConfig } from '@capacitor/core';

const config: CapacitorConfig = {
  appId: 'com.pumpguard.app',
  appName: 'PumpGuard',
  webDir: 'dist',
  plugins: {
    Camera: {
      presentationStyle: 'fullscreen',
    },
    StatusBar: {
      style: 'dark',
      backgroundColor: '#000000',
    },
  },
};
```

Native Features:
- Camera plugin for native camera access
- Haptics for tactile feedback
- Status bar customization
- Android/iOS builds via Capacitor CLI

6. Testing Strategy

Unit Tests (Vitest):

```
import { describe, it, expect, vi } from 'vitest';
import { render, screen } from '@testing-library/react';

describe('CameraComponent', () => {
  it('should enumerate devices', async () => {
    const mockDevices = [
      { deviceId: '1', kind: 'videoinput', label: 'Front Camera' },
      { deviceId: '2', kind: 'videoinput', label: 'Back Camera' },
    ];

    vi.spyOn(navigator.mediaDevices, 'enumerateDevices')
      .mockResolvedValue(mockDevices);

    const devices = await enumerateVideoDevices();
    expect(devices).toHaveLength(2);
  });
});
```

E2E Tests (Playwright):

```
import { test, expect } from '@playwright/test';

test('measure smoke test', async ({ page }) => {
  await page.goto('/measure');
  await expect(page.locator('h1')).toContainText('Measure');
});
```

7. Build & Deployment

Vite Configuration:
- React SWC plugin for fast refresh
- Compression plugin for gzip/brotli
- Bundle analyzer for size optimization
- Sentry plugin for error tracking

Docker Support:

```
# Dockerfile.web
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
EXPOSE 8080
CMD ["npm", "run", "preview"]
```

Scripts:
- `npm run dev` - Vite dev server
- `npm run server` - Express API server
- `npm run dev:all` - Concurrent dev + API
- `npm run build` - Production build + service worker
- `npm run build:android` - Capacitor Android build

Relevant Patterns for Multi-Camera App

Pattern 1: Camera Preferences Persistence

```
// Save camera preferences
const prefs = {
  deviceId: selectedDevice,
  facingMode,
  zoom: currentZoom,
};
localStorage.setItem("camera.prefs", JSON.stringify(prefs));

// Load on mount
useEffect(() => {
  const raw = localStorage.getItem("camera.prefs");
  if (raw) {
    const prefs = JSON.parse(raw);
    setSelectedDevice(prefs.deviceId);
    setFacingMode(prefs.facingMode);
  }
}, []);
```

Pattern 2: Camera Stream Management

```
useEffect(() => {
  let stream: MediaStream | null = null;

  const initCamera = async () => {
    try {
      const result = await startCamera({
        deviceId: selectedDevice,
        facingMode,
        width: 1280,
        height: 720,
      });
      stream = result.stream;
      setStream(stream);
      setCapabilities(result.capabilities);
    } catch (error) {
      setError(error.message);
    }
  };

  if (mode === 'live') {
    initCamera();
  }

  return () => {
    if (stream) stopStream(stream);
  };
}, [mode, selectedDevice, facingMode]);
```

Pattern 3: Error Boundaries

```
class ErrorBoundary extends React.Component {
  state = { hasError: false, error: null };

  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }

  componentDidCatch(error, errorInfo) {
    console.error('Camera error:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <ErrorFallback error={this.state.error} />;
    }
    return this.props.children;
  }
}
```

Integrated Concepts for Multi-Camera Sync App

Architecture Recommendations

1. Hybrid Approach: Supabase Realtime + WebRTC

Why this combination:
- **Supabase Realtime** for signaling, presence, and commands (lightweight)
- **WebRTC** for peer-to-peer video streaming (low latency, no server bandwidth)
- Proven pattern from ex-pose-positions repository

Architecture Diagram:

```
┌─────────────┐        ┌─────────────┐        ┌─────────────┐
│  Camera 1   │        │  Camera 2   │        │  Camera 3   │
│  (Phone)    │        │  (Tablet)   │        │  (Laptop)   │
└──────┬──────┘        └──────┬──────┘        └──────┬──────┘
       │                      │                      │
       │    Supabase Realtime Channel (Signaling)    │
       └──────────────────────┼──────────────────────┘
                              │
                   ┌──────────▼──────────┐
                   │  Supabase Backend   │
                   │  - Presence API     │
                   │  - Broadcast Events │
                   │  - Session Storage  │
                   └─────────────────────┘

       WebRTC Peer Connections (Direct Video Streams)
       ┌───────────────────────────────────────────┐
       │  Camera 1 ←──→ Camera 2 ←──→ Camera 3     │
       │  (Mesh topology for 1-4 cameras)          │
       └───────────────────────────────────────────┘
```

2. Technology Stack Recommendation

Frontend:
- React 18+ with TypeScript
- Vite for build tooling
- Tailwind CSS + shadcn/ui for UI components
- TanStack Query for server state management
- Zustand or Jotai for client state (simpler than Redux)

Backend:
- Supabase for:
- Realtime channels (signaling)
- PostgreSQL database (session metadata, recordings metadata)
- Storage (optional: for recording uploads)
- Authentication (optional: for user accounts)

Real-time Communication:
- Supabase Realtime for signaling
- WebRTC for video streaming
- STUN servers (Google's public STUN)
- Optional TURN server for NAT traversal (if needed)

Mobile:
- Capacitor for iOS/Android builds
- Progressive Web App (PWA) for web access

3. Core Features Implementation

Feature 1: Camera Connection (1-4 devices)

Implementation:

```
// hooks/useCameraSync.ts
export function useCameraSync(sessionId: string) {
  const [cameras, setCameras] = useState<Camera[]>([]);
  const [localStream, setLocalStream] = useState<MediaStream | null>(null);
  const [remoteStreams, setRemoteStreams] = useState<Map<string, MediaStream>>(new
Map());

  // Initialize Supabase channel
  useEffect(() => {
    const channel = supabase.channel(`session:${sessionId}`);

    channel
      .on('presence', { event: 'sync' }, handlePresenceSync)
      .on('broadcast', { event: 'signal' }, handleSignal)
      .on('broadcast', { event: 'command' }, handleCommand)
      .subscribe();

    return () => channel.unsubscribe();
  }, [sessionId]);

  // WebRTC peer management
  const connectToPeer = useCallback(async (peerId: string) => {
    const peer = new RTCPeerConnection(ICE_CONFIG);

    // Add local tracks
    localStream?.getTracks().forEach(track => {
      peer.addTrack(track, localStream);
    });

    // Handle remote tracks
    peer.ontrack = (event) => {
      setRemoteStreams(prev => new Map(prev).set(peerId, event.streams[0]));
    };

    // Create offer and send via Supabase
    const offer = await peer.createOffer();
    await peer.setLocalDescription(offer);

    channel.send({
      type: 'broadcast',
      event: 'signal',
      payload: { type: 'offer', to: peerId, sdp: offer },
    });

    return peer;
  }, [localStream]);

  return {
    cameras,
    localStream,
    remoteStreams,
    connectToPeer,
  };
}
```

Feature 2: Synchronized Recording

Implementation:

```
// hooks/useSyncedRecording.ts
export function useSyncedRecording(channel: RealtimeChannel) {
  const [isRecording, setIsRecording] = useState(false);
  const [countdown, setCountdown] = useState(0);
  const recordersRef = useRef<Map<string, MediaRecorder>>(new Map());

  const startRecording = useCallback((leadTimeMs = 3000) => {
    const startAt = Date.now() + leadTimeMs;

    // Broadcast to all cameras
    channel.send({
      type: 'broadcast',
      event: 'command',
      payload: {
        action: 'start_recording',
        startAt,
        timestamp: Date.now(),
      },
    });

    // Schedule local recording
    scheduleRecording(startAt);
  }, [channel]);

  const scheduleRecording = useCallback((startAt: number) => {
    const delay = startAt - Date.now();

    // Update countdown
    const interval = setInterval(() => {
      const remaining = Math.ceil((startAt - Date.now()) / 1000);
      setCountdown(Math.max(0, remaining));
    }, 100);

    // Start recording at exact time
    setTimeout(() => {
      clearInterval(interval);
      setCountdown(0);
      startLocalRecording();
    }, delay);
  }, []);

  const startLocalRecording = useCallback(() => {
    // Record all streams (local + remote)
    const streams = [localStream, ...Array.from(remoteStreams.values())];

    streams.forEach((stream, index) => {
      const recorder = new MediaRecorder(stream, {
        mimeType: 'video/webm;codecs=vp9,opus',
      });

      const chunks: Blob[] = [];
      recorder.ondataavailable = (e) => chunks.push(e.data);
      recorder.onstop = () => saveRecording(chunks, `camera-${index}`);

      recorder.start();
      recordersRef.current.set(`stream-${index}`, recorder);
    });

    setIsRecording(true);
  }, [localStream, remoteStreams]);

  const stopRecording = useCallback(() => {
    // Broadcast stop command
    channel.send({
      type: 'broadcast',
      event: 'command',
      payload: { action: 'stop_recording', timestamp: Date.now() },
    });

    // Stop all local recorders
    recordersRef.current.forEach(recorder => recorder.stop());
    recordersRef.current.clear();
    setIsRecording(false);
  }, [channel]);

  return {
    isRecording,
    countdown,
    startRecording,
    stopRecording,
  };
}
```

Feature 3: Playback with Camera Switching

Implementation:

```tsx
// components/PlaybackViewer.tsx
export function PlaybackViewer({ recordings }: { recordings: Recording[] }) {
  const [activeCamera, setActiveCamera] = useState(0);
  const [isPlaying, setIsPlaying] = useState(false);
  const videoRefs = useRef<Map<number, HTMLVideoElement>>(new Map());

  // Sync playback across all videos
  const syncPlayback = useCallback(() => {
    const activeVideo = videoRefs.current.get(activeCamera);
    if (!activeVideo) return;

    const currentTime = activeVideo.currentTime;

    videoRefs.current.forEach((video, index) => {
      if (index !== activeCamera) {
        video.currentTime = currentTime;
        if (isPlaying) video.play();
        else video.pause();
      }
    });
  }, [activeCamera, isPlaying]);

  // Switch camera while maintaining sync
  const switchCamera = useCallback((cameraIndex: number) => {
    const currentVideo = videoRefs.current.get(activeCamera);
    const newVideo = videoRefs.current.get(cameraIndex);

    if (currentVideo && newVideo) {
      newVideo.currentTime = currentVideo.currentTime;
      if (isPlaying) newVideo.play();
    }

    setActiveCamera(cameraIndex);
  }, [activeCamera, isPlaying]);

  return (
    <div className="playback-viewer">
      {/* Main video display */}
      <div className="main-video">
        {recordings.map((recording, index) => (
          <video
            key={recording.id}
            ref={(el) => el && videoRefs.current.set(index, el)}
            src={recording.url}
            className={index === activeCamera ? 'visible' : 'hidden'}
            onTimeUpdate={syncPlayback}
          />
        ))}
      </div>

      {/* Camera switcher */}
      <div className="camera-grid">
        {recordings.map((recording, index) => (
          <button
            key={recording.id}
            onClick={() => switchCamera(index)}
            className={index === activeCamera ? 'active' : ''}
          >
            <video src={recording.url} muted />
            <span>Camera {index + 1}</span>
          </button>
        ))}
      </div>

      {/* Playback controls */}
      <div className="controls">
        <button onClick={() => setIsPlaying(!isPlaying)}>
          {isPlaying ? 'Pause' : 'Play'}
        </button>
        {/* Additional controls */}
      </div>
    </div>
  );
}
```

Feature 4: Side-by-Side View

Implementation:

```
// components/SideBySideView.tsx
export function SideBySideView({ recordings }: { recordings: Recording[] }) {
  const [layout, setLayout] = useState<'2x2' | '1x3' | '4x1'>('2x2');
  const videoRefs = useRef<HTMLVideoElement[]>([]);

  // Sync all videos
  const syncAllVideos = useCallback(() => {
    const primaryVideo = videoRefs.current[0];
    if (!primaryVideo) return;

    const currentTime = primaryVideo.currentTime;
    const isPlaying = !primaryVideo.paused;

    videoRefs.current.slice(1).forEach(video => {
      if (Math.abs(video.currentTime - currentTime) > 0.1) {
        video.currentTime = currentTime;
      }
      if (isPlaying && video.paused) video.play();
      if (!isPlaying && !video.paused) video.pause();
    });
  }, []);

  const layoutClasses = {
    '2x2': 'grid grid-cols-2 grid-rows-2',
    '1x3': 'grid grid-cols-1 grid-rows-3',
    '4x1': 'grid grid-cols-4 grid-rows-1',
  };

  return (
    <div className="side-by-side-view">
      {/* Layout selector */}
      <div className="layout-controls">
        <button onClick={() => setLayout('2x2')}>2×2 Grid</button>
        <button onClick={() => setLayout('1x3')}>1×3 Stack</button>
        <button onClick={() => setLayout('4x1')}>4×1 Row</button>
      </div>

      {/* Video grid */}
      <div className={layoutClasses[layout]}>
        {recordings.map((recording, index) => (
          <div key={recording.id} className="video-container">
            <video
              ref={(el) => el && (videoRefs.current[index] = el)}
              src={recording.url}
              onTimeUpdate={index === 0 ? syncAllVideos : undefined}
              controls={index === 0}
            />
            <div className="video-label">Camera {index + 1}</div>
          </div>
        ))}
      </div>
    </div>
  );
}
```

4. Local Network Optimization

mDNS/Bonjour Discovery (Optional Enhancement):

```
// For local network discovery without manual IP entry
// This would require a native module or service worker

// Alternative: Use QR code with local IP
const generateJoinQR = (sessionId: string) => {
  const localIP = getLocalIP(); // From WebRTC or server
  const url = `https://lh6.googleusercontent.com/Knnn-BnwWKsux5nhlPaO7kn-
fzHVKq4YN_cZJqL1xu9pfFBNTfgAnIN69TD31v5kRCNEctvwcPgxTPX1NsA8KKmPiDpo9urSk7WhKQIukrmGq6
NmAPAfqrMlv3sA1wdELMci84mJi`;
  return generateQRCode(url);
};
```

WebRTC Configuration for Local Network:

```
const ICE_CONFIG: RTCConfiguration = {
  iceServers: [
    { urls: 'stun:stun.l.google.com:19302' },
    { urls: 'stun:stun1.l.google.com:19302' },
  ],
  iceTransportPolicy: 'all', // Allow both local and remote candidates
  iceCandidatePoolSize: 10,
};
```

5. Data Flow Architecture

Session Creation Flow:

```
1. User creates session → Generate unique session ID
2. Display QR code / share link
3. Initialize Supabase channel
4. Track presence (who's online)
5. Enable local camera
6. Wait for other devices to join
```

Device Join Flow:

```
1. Scan QR / open link → Extract session ID
2. Join Supabase channel
3. Announce presence
4. Enable local camera
5. Establish WebRTC connections with existing peers
6. Display all camera feeds
```

Recording Flow:

```
1. Primary device initiates recording
2. Broadcast start command with timestamp
3. All devices schedule recording at exact time
4. Countdown displayed on all devices
5. Recording starts simultaneously
6. Each device records:
   - Own camera feed
   - All remote feeds (optional)
7. Stop command broadcasts to all
8. Each device saves recordings locally
9. Optional: Upload to cloud storage
```

Playback Flow:

```
1. Load all recordings from session
2. Display in grid or single view
3. Sync playback across all videos
4. Allow camera switching
5. Support side-by-side layouts
```

6. State Management Strategy

Recommended: Zustand for Simplicity

```
// store/cameraStore.ts
import create from 'zustand';

interface CameraStore {
  sessionId: string | null;
  localStream: MediaStream | null;
  remoteStreams: Map<string, MediaStream>;
  cameras: Camera[];
  isRecording: boolean;
  countdown: number;

  setSessionId: (id: string) => void;
  setLocalStream: (stream: MediaStream) => void;
  addRemoteStream: (peerId: string, stream: MediaStream) => void;
  removeRemoteStream: (peerId: string) => void;
  startRecording: () => void;
  stopRecording: () => void;
}

export const useCameraStore = create<CameraStore>((set) => ({
  sessionId: null,
  localStream: null,
  remoteStreams: new Map(),
  cameras: [],
  isRecording: false,
  countdown: 0,

  setSessionId: (id) => set({ sessionId: id }),
  setLocalStream: (stream) => set({ localStream: stream }),
  addRemoteStream: (peerId, stream) => set((state) => ({
    remoteStreams: new Map(state.remoteStreams).set(peerId, stream),
  })),
  removeRemoteStream: (peerId) => set((state) => {
    const newMap = new Map(state.remoteStreams);
    newMap.delete(peerId);
    return { remoteStreams: newMap };
  }),
  startRecording: () => set({ isRecording: true }),
  stopRecording: () => set({ isRecording: false }),
}));
```

7. Error Handling & Edge Cases

Network Issues:

```
// Reconnection logic
const handleDisconnect = useCallback(() => {
  console.warn('Connection lost, attempting reconnection...');

  const reconnect = async () => {
    try {
      await channel.subscribe();
      console.log('Reconnected successfully');
    } catch (error) {
      console.error('Reconnection failed, retrying in 5s...');
      setTimeout(reconnect, 5000);
    }
  };

  reconnect();
}, [channel]);
```

Camera Permission Denied:

```
const requestCameraPermission = async () => {
  try {
    const stream = await navigator.mediaDevices.getUserMedia({ video: true });
    return stream;
  } catch (error) {
    if (error.name === 'NotAllowedError') {
      showError('Camera permission denied. Please enable camera access in settings.');
    } else if (error.name === 'NotFoundError') {
      showError('No camera found on this device.');
    } else {
      showError('Failed to access camera: ' + error.message);
    }
    throw error;
  }
};
```

Sync Drift Correction:

```
// Periodically sync timestamps
const syncTimestamps = useCallback(() => {
  const localTime = Date.now();

  channel.send({
    type: 'broadcast',
    event: 'sync',
    payload: { timestamp: localTime },
  });
}, [channel]);

// Run every 10 seconds
useEffect(() => {
  const interval = setInterval(syncTimestamps, 10000);
  return () => clearInterval(interval);
}, [syncTimestamps]);
```

8. Performance Optimization

Video Quality Adaptation:

```
const adaptVideoQuality = (bandwidth: number) => {
  let constraints;

  if (bandwidth > 5000) {
    // High bandwidth: 1080p
    constraints = { width: 1920, height: 1080, frameRate: 30 };
  } else if (bandwidth > 2000) {
    // Medium bandwidth: 720p
    constraints = { width: 1280, height: 720, frameRate: 30 };
  } else {
    // Low bandwidth: 480p
    constraints = { width: 640, height: 480, frameRate: 24 };
  }

  return constraints;
};
```

Lazy Loading & Code Splitting:

```
// Lazy load playback components
const PlaybackViewer = lazy(() => import('./components/PlaybackViewer'));
const SideBySideView = lazy(() => import('./components/SideBySideView'));

// Route-based code splitting
const routes = [
  { path: '/', component: lazy(() => import('./pages/Home')) },
  { path: '/session/:id', component: lazy(() => import('./pages/Session')) },
  { path: '/playback/:id', component: lazy(() => import('./pages/Playback')) },
];
```

9. Security Considerations

Session Access Control:

```
// Generate secure session IDs
const createSession = () => {
  const sessionId = crypto.randomUUID();
  const accessToken = generateSecureToken(); // For additional security

  return {
    sessionId,
    accessToken,
    createdAt: Date.now(),
    expiresAt: Date.now() + 24 * 60 * 60 * 1000, // 24 hours
  };
};

// Validate session access
const validateSession = (sessionId: string, token?: string) => {
  const session = getSession(sessionId);

  if (!session) return false;
  if (session.expiresAt < Date.now()) return false;
  if (token && session.accessToken !== token) return false;

  return true;
};
```

HTTPS Requirement:
- WebRTC requires HTTPS in production
- Use Let’s Encrypt for free SSL certificates
- Development: Use `localhost` (allowed without HTTPS)

10. Testing Strategy

Unit Tests:

```
// Test camera utilities
describe('Camera Utils', () => {
  it('should enumerate video devices', async () => {
    const devices = await enumerateVideoDevices();
    expect(devices.every(d => d.kind === 'videoinput')).toBe(true);
  });

  it('should select back camera for environment facing', async () => {
    const deviceId = await selectDeviceForFacing('environment');
    expect(deviceId).toBeDefined();
  });
});

// Test WebRTC connection
describe('WebRTC Connection', () => {
  it('should create peer connection', () => {
    const peer = new RTCPeerConnection(ICE_CONFIG);
    expect(peer.connectionState).toBe('new');
  });

  it('should add tracks to peer', async () => {
    const stream = await navigator.mediaDevices.getUserMedia({ video: true });
    const peer = new RTCPeerConnection(ICE_CONFIG);

    stream.getTracks().forEach(track => peer.addTrack(track, stream));

    expect(peer.getSenders().length).toBeGreaterThan(0);
  });
});
```

Integration Tests:

```
// Test multi-device sync
describe('Multi-Camera Sync', () => {
  it('should sync recording start across devices', async () => {
    const device1 = createMockDevice();
    const device2 = createMockDevice();

    await device1.joinSession('test-session');
    await device2.joinSession('test-session');

    const startTime = Date.now() + 3000;
    device1.startRecording(startTime);

    await waitFor(() => {
      expect(device2.countdown).toBeGreaterThan(0);
    });

    await waitFor(() => {
      expect(device1.isRecording).toBe(true);
      expect(device2.isRecording).toBe(true);
    }, { timeout: 5000 });
  });
});
```

E2E Tests:

```
// Test full workflow
test('multi-camera recording workflow', async ({ page, context }) => {
  // Create session
  await page.goto('/');
  await page.click('button:has-text("Create Session")');

  const sessionLink = await page.locator('[data-testid="session-link"]').textCon-
tent();

  // Join from second device
  const page2 = await context.newPage();
  await page2.goto(sessionLink);

  // Enable cameras
  await page.click('button:has-text("Enable Camera")');
  await page2.click('button:has-text("Enable Camera")');

  // Start recording
  await page.click('button:has-text("Start Recording")');

  // Wait for countdown
  await expect(page.locator('[data-testid="countdown"]')).toBeVisible();
  await expect(page2.locator('[data-testid="countdown"]')).toBeVisible();

  // Verify recording started
  await expect(page.locator('[data-testid="recording-indicator"]')).toBeVisible();
  await expect(page2.locator('[data-testid="recording-indicator"]')).toBeVisible();

  // Stop recording
  await page.click('button:has-text("Stop Recording")');

  // Verify recordings saved
  await expect(page.locator('[data-testid="recording-saved"]')).toBeVisible();
  await expect(page2.locator('[data-testid="recording-saved"]')).toBeVisible();
});
```

Implementation Roadmap

Phase 1: Foundation (Week 1-2)

- [ ] Set up project structure (Vite + React + TypeScript)
- [ ] Configure Tailwind CSS + shadcn/ui
- [ ] Set up Supabase project and client
- [ ] Implement camera utilities (from size-sync-studio patterns)
- [ ] Create basic UI layout and routing

Phase 2: Core Sync (Week 3-4)

- [ ] Implement Supabase Realtime channel setup
- [ ] Build presence tracking system
- [ ] Create WebRTC peer connection manager
- [ ] Implement signaling flow (offer/answer/ICE)
- [ ] Test multi-device video streaming

Phase 3: Recording (Week 5-6)

- [ ] Implement synchronized recording with countdown
- [ ] Build MediaRecorder integration
- [ ] Create recording management (start/stop/save)
- [ ] Add recording metadata storage
- [ ] Test recording sync across devices

Phase 4: Playback (Week 7-8)

- [ ] Build playback viewer component
- [ ] Implement camera switching
- [ ] Create side-by-side view layouts
- [ ] Add playback controls (play/pause/seek)
- [ ] Implement video sync during playback

Phase 5: Polish & Testing (Week 9-10)

- [ ] Add error handling and fallbacks
- [ ] Implement reconnection logic
- [ ] Create onboarding/tutorial flow
- [ ] Write comprehensive tests
- [ ] Performance optimization
- [ ] Mobile responsiveness
- [ ] Accessibility improvements

Phase 6: Deployment (Week 11-12)

- [ ] Set up production Supabase project
- [ ] Configure HTTPS/SSL
- [ ] Deploy to hosting platform (Vercel/Netlify)
- [ ] Set up monitoring and analytics
- [ ] Create user documentation

- [ ] Beta testing with real users

Key Takeaways

From ex-pose-positions:

1. **Supabase Realtime + WebRTC is a proven pattern** for multi-device video sync
2. **Presence API** provides elegant online/offline tracking
3. **Broadcast events** enable synchronized commands across devices
4. **Refs over state** for WebRTC objects prevents unnecessary re-renders
5. **Countdown-based sync** ensures simultaneous recording start
6. **Session sharing via links** is user-friendly for joining

From size-sync-studio:

1. **Robust camera utilities** with fallback strategies
2. **Device selection heuristics** improve UX on mobile
3. **Capabilities detection** enables adaptive features
4. **Express server patterns** for optional backend
5. **TanStack Query** for efficient data fetching
6. **Comprehensive testing** ensures reliability

Best Practices:

1. **Use TypeScript** for type safety with WebRTC APIs
2. **Implement cleanup** for all streams and connections
3. **Handle errors gracefully** with user-friendly messages
4. **Test on real devices** (mobile + desktop)
5. **Optimize for local network** (prefer local candidates)
6. **Keep UI responsive** during connection setup
7. **Provide visual feedback** for all async operations
8. **Support offline mode** where possible
9. **Log extensively** for debugging WebRTC issues
10. **Document WebRTC quirks** for future maintenance

Resources & References

Documentation:

- Supabase Realtime Docs (https://supabase.com/docs/guides/realtime)
- WebRTC API (MDN) (https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API)
- MediaRecorder API (https://developer.mozilla.org/en-US/docs/Web/API/MediaRecorder)
- MediaDevices API (https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices)

Libraries:

- Supabase JS Client (https://github.com/supabase/supabase-js)
- Simple Peer (https://github.com/feross/simple-peer) (optional WebRTC wrapper)

- RecordRTC (https://github.com/muaz-khan/RecordRTC) (advanced recording)

Tools:

- WebRTC Samples (https://webrtc.github.io/samples/)
- Supabase Studio (https://supabase.com/dashboard)
- Chrome WebRTC Internals (chrome://webrtc-internals)

Conclusion

Both repositories provide excellent foundations for building a multi-camera synchronization app. The **ex-pose-positions** repository's Camera Sync Studio is particularly valuable as it demonstrates a complete, working implementation of the exact features you need. The **size-sync-studio** repository complements this with robust camera utilities and server patterns.

Recommended Approach:
1. Start with the Camera Sync Studio code from ex-pose-positions as your foundation
2. Integrate the advanced camera utilities from size-sync-studio
3. Build the playback features (camera switching, side-by-side) as new components
4. Follow the testing and deployment patterns from both repos
5. Iterate based on user feedback

The combination of Supabase Realtime for signaling and WebRTC for video streaming is proven, scalable, and cost-effective for 1-4 camera setups. The architecture supports both local network and remote connections, making it flexible for various use cases.

Next Steps:
1. Set up a Supabase project
2. Clone the relevant code patterns into your new project
3. Start with basic camera enumeration and streaming
4. Add Supabase Realtime integration
5. Implement WebRTC peer connections
6. Build recording and playback features
7. Test extensively on multiple devices
8. Deploy and iterate

Good luck with your multi-camera sync application! The patterns from these repositories provide a solid foundation for success.