iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.83113

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

MonitorX - Real-Time Screen Monitoring Platform using WebRTC: A Review

Atharv Bhosale[1], Darshan Patil[2]
[1]*Department of Computer Engineering, PK Technical Campus, Savitribai Phule Pune University (SPPU), Pune, India*
[2]*Department of Computer Engineering, Ajeenkya D.Y. Patil School of Engineering & PK Technical Campus, Savitribai Phule Pune University (SPPU), Pune, India*

Abstract: ***Remote classrooms and distributed training arrangements are increasingly commonplace in the world's educational industry, but the monitoring capabilities available to teachers are far from ideal. Many existing software systems were repurposed rather than being created from scratch for the purpose of remote classroom monitoring, and it is clear in their designs. This paper discusses MonitorX, a new platform designed specifically to enable instant screen monitoring by an instructor of all participants within the classroom in near-zero latency. The platform uses HTML5 video streaming with WebRTC technology and allows up to twenty participants to send screen streams to the monitor in a grid format in real-time. All interactions are handled via an included instant messaging facility. Unlike traditional remote monitoring technologies, MonitorX does not rely on any server; instead, browser-to-browser peer connections provide a low-latency environment without significant operational expenses. A particularly challenging problem the developers had to solve is one of ICE candidate management when establishing peer-to-peer communication in the WebRTC technology - an apparently small but critical aspect causing high failure rates in naive implementations of WebRTC-based communication. The design and performance of MonitorX in general and its solution to the problem in particular are discussed below. The results indicate that the connectivity success rate was 99.2%, latency of about 35 ms, and stable performance of the system with twenty users at 55 to 60 fps, at a much lower cost than that of server-based systems***

Keywords: WebRTC, Real-Time Communication, Peer-to-Peer Networking, React 18, Supabase, Remote Monitoring, ICE Candidate Queuing, Educational Technology.

I. INTRODUCTION

[1] Teaching online isn't easy. With students located across multiple places, the teacher misses out on what turns out to be a very effective tool — just taking a stroll and looking at whatever everyone is doing. This kind of observation becomes difficult without realizing its importance once it is no longer possible. Although most remote learning programs offer video conferencing and chatting capabilities, none offer the teachers a view into what is happening on each student's device as they work. This is where MonitorX comes in.

[2] While many monitoring programs utilize a centralized server, this causes increased latency, additional costs for the institution, and a point of failure should the system ever crash. This would happen to all teachers using the service at the same time. Since the program needs to send every single frame from the device to the server and back again, this creates additional strain on both sides and increases the likelihood of something going wrong.

[3] MonitorX takes a different approach by building directly on WebRTC, the browser standard for real-time peer-to-peer communication. With a proper P2P setup, the screen video never touches a central server — it goes straight from the participant's browser to the instructor's browser. That cuts latency dramatically, removes the server bottleneck, and reduces infrastructure needs to a lightweight backend that only handles authentication and signaling. The media itself is free.

[4] The platform is a full-stack web application. The frontend is built with React 18 and TypeScript, styled with Tailwind CSS, and animated with Framer Motion. The backend uses Supabase — an open-source Firebase alternative — for authentication, a PostgreSQL database, and real-time signaling via Supabase Broadcast Channels. No custom media server is needed. Users connect through a session code, the host sees all participant screens in a grid, and everyone can chat in real time.

[5] Practically every remote teaching tool in the market today struggles with at least one of these: too much latency, sessions that drop without warning, overly complex setup, or pricing that is simply not viable for smaller institutions. Many tools require participants to install separate software, which adds friction and compatibility headaches. MonitorX runs entirely in the browser and needs nothing more than a link and a session code to get started.

[6] The most technically interesting challenge during development was getting WebRTC connections to establish reliably. WebRTC handshaking involves an exchange of SDP offers and ICE candidates, and in practice, candidates often arrive in a burst — faster than the connection is ready to process them. Without careful handling, this causes silent failures where the connection appears to succeed but no media actually flows. The queuing mechanism built into MonitorX addresses this specifically.

[7] The project also situates itself in a broader trend worth acknowledging. The shift toward hybrid and remote learning is not going away. Institutions that moved online during the pandemic have largely kept some version of that infrastructure. The demand for tools that actually work — not just tools that exist — is ongoing. MonitorX is meant to be a practical, deployable answer to that demand, not just a proof of concept. [8] This paper documents the full architecture of the system, the reasoning behind key implementation choices, the challenges encountered and how they were resolved, and the measured results from testing. Section 2 covers system architecture, Section 3 explains data flow, Section 4 goes into the technical implementation and core algorithms, Section 5 discusses challenges and solutions, Section 6 presents results, and Sections 7 through 10 cover conclusions, future directions, acknowledgements, and references.

Fig. 1: MonitorX System Architecture – Layered Overview

II. SYSTEM ARCHITECTURE

MonitorX uses what could be called a hybrid architecture — peer-to-peer where it matters most (the media), and centralized where it makes sense (state, auth, signaling). This split was not arbitrary. Running media through a server kills the latency and cost advantages of P2P, but trying to do session management and authentication purely peer-to-peer without a backend introduces its own mess of complexity and security issues. The two concerns genuinely need different approaches.

The architecture breaks cleanly into four layers. The presentation layer is the React frontend running in each user's browser. The signaling layer is Supabase Realtime, which handles the ephemeral messages WebRTC needs to set up connections. The persistence layer is a PostgreSQL database hosted on Supabase, holding users, sessions, messages, and help requests. The media transport layer is WebRTC itself — direct P2P streams between participants and the host.

A. Core Modules

Authentication & Authorization: Built on Supabase Auth with JWT-based sessions. Handles email/password login, session persistence across browser tabs, automatic recovery on reconnect, and role differentiation between hosts and participants. Every token has a 24-hour lifetime.

Session Management: Manages the lifecycle of monitoring sessions from creation through completion. Generates unique session codes, enforces participant limits, tracks who is connected, and drives the state transitions: waiting → active → ended. Host and participant dashboards subscribe to real-time updates and react automatically.

WebRTC P2P Module: The heart of the system. Manages peer connection objects, ICE candidate collection, offer/answer exchange, stream state, and reconnection. Each participant gets their own RTCPeerConnection instance on the host side. This module includes the ICE candidate queuing logic described in Section 4.

Real-Time Signaling: Supabase Broadcast Channels act as the messaging layer for WebRTC signaling. Messages are ephemeral — they are delivered but not stored — which is exactly what signaling needs. Latency on this channel is consistently under 50ms in testing.

Chat & Messaging: Supports both session-wide group messages and private one-to-one messages. Group messages go through Supabase Realtime and are persisted to the database. Private messages are handled through the existing P2P channel and also persisted. Typing indicators work via Realtime without database writes.

UI & Interaction Layer: React 18 with TypeScript, Tailwind CSS for layout and styling, Framer Motion for animations, and Radix UI primitives for accessible components. The grid layout recalculates dynamically as participants join or leave. Dark theme with glassmorphism styling throughout. Help/raise-hand alerts appear with an animated overlay on the relevant tile.

B. Technology Stack

Table 1: Technology Stack

Component	Technology	Purpose
Frontend Framework	React 18 + TypeScript	UI and component management
Build Tooling	Vite 5.x	Development server and builds

Component	Technology	Purpose
Styling	Tailwind CSS 3.x	Utility-first CSS
UI Primitives	Radix UI + shadcn/ui	Accessible component base
Animations	Framer Motion 10.x	Transitions and alerts
State Management	TanStack React Query 5.x	Server state sync
Real-Time Layer	Supabase Realtime 2.x	WebSocket signaling
Backend / Auth	Supabase 2.101.x	Auth, DB, Realtime
Database	PostgreSQL 15.x	Persistent data storage
P2P Media	Native WebRTC API	Direct stream transport

C. Database Schema

Five tables handle persistent state. The profiles table stores user identity, role, and organisation. The sessions table records session metadata, the unique code, host, status, and grid layout preference. Session_participants tracks who joined and when, and whether their stream is currently active. The messages table holds both group and private chat with sender, optional recipient, and timestamp. Finally, help_requests logs raise-hand events with status and resolution time. All primary keys are UUIDs; foreign keys enforce referential integrity throughout.

D. Security Design

Security was considered from the beginning rather than bolted on later. Transport is HTTPS everywhere; WebRTC media is encrypted automatically by the protocol using DTLS-SRTP, so there is no plaintext media at any point. Supabase Realtime runs over WSS. Session codes are cryptographically generated (Web Crypto API) and expire after 24 hours. The database uses parameterized queries throughout, eliminating SQL injection risk. Row-level security policies on Supabase prevent participants from reading or modifying data they should not have access to. A third-party audit found no critical vulnerabilities.

Fig. 2: Data Flow Diagram – Message & Stream Processing

III. DATA FLOW & WORKING

Understanding how a session actually runs from start to finish helps clarify why the architecture is designed the way it is. The workflow splits into six phases, and the handoff between them is where most of the interesting engineering happens.

1) Phase 1 – Session Initialization

The host logs in, and Supabase Auth returns a JWT that is stored in a secure cookie. The host then creates a session by submitting a name and participant limit. The backend generates a unique session code (more on this in Section 4), writes the session record to the database with status 'waiting', and subscribes the host's dashboard to real-time updates for that session. At this point the grid is empty and waiting.

2) Phase 2 – Participant Joining

Participants authenticate and enter the session code. The system validates the code, checks that the session is open and below the participant cap, and writes a session_participants record. A Supabase Realtime event fires immediately, notifying the host's browser that a new participant has arrived. The grid recalculates and a placeholder tile appears. The participant's dashboard shows a waiting state while WebRTC setup begins.

3) Phase 3 – Screen Share Initiation

Once the participant confirms they want to share, navigator.mediaDevices.getDisplayMedia() fires. The browser shows its built-in permission dialog. Once the user selects a screen or window, the MediaStream is captured. An RTCPeerConnection is created with STUN server configuration, the video track is added, and ICE candidate collection starts. This is where the queuing mechanism described in Section 4 kicks in.

4) Phase 4 – P2P Handshake

The participant generates an SDP offer and sets it as the local description. The offer, along with the first batch of ICE candidates, is broadcast to the host via Realtime. The host receives it, creates a matching RTCPeerConnection, sets the offer as the remote description, generates an SDP answer, and sends it back. Both sides continue exchanging ICE candidates as they are discovered. When enough connectivity information has been exchanged, the DTLS handshake completes and an encrypted media channel opens.

5) Phase 5 – Live Monitoring

Video frames start flowing through the direct P2P connection. The host's browser receives the MediaStream and attaches it to a video element inside the corresponding grid tile via srcObject. Framer Motion handles the tile appearance animation. Connection health is monitored continuously; if the connection state drops to 'disconnected', the reconnection state machine fires automatically with exponential backoff. Under good network conditions, from session code entry to live video appearing on the host's screen takes around 2.2 seconds.

6) Phase 6 – Communication & Help

Chat messages typed by any participant are broadcast over Supabase Realtime and simultaneously written to the messages table. The host sees them in the chat panel in real time. When a participant clicks the help/raise-hand button, a help_request record is created and a Realtime event fires. The host sees an animated pulse overlay on that participant's tile, along with a notification. The host can resolve it with a single click, which updates the record status and clears the alert.

A. Performance Summary

Table 2: Performance Benchmarks

Metric	Target	Achieved
Time to Join Session	< 2 s	~1.5 s
Screen Share Start	< 3 s	~2.2 s
P2P Handshake Time	< 1.5 s	~1.2 s
Screen Update Latency	< 50 ms	~35 ms avg
Message Delivery	< 100 ms	~65 ms avg
Help Alert Delivery	< 200 ms	~120 ms avg
Grid Render (20 users)	55+ FPS	55-60 FPS
Initial Page Load	< 2 s	~1.8 s
Host Memory (20 users)	< 200 MB	~150-180 MB

IV. TECHNICAL IMPLEMENTATION & ALGORITHMS

A. ICE Candidate Queuing

This was probably the trickiest part of the whole project. WebRTC connections fail silently more often than most documentation admits, and a large proportion of those failures in development came down to one thing: ICE candidates arriving faster than the connection was ready for them.

When a peer connection is created, the browser starts gathering ICE candidates — essentially network address/port combinations the connection might use. These fire as events on the RTCPeerConnection object, and the naive approach is to send each one over the signaling channel as soon as it arrives. The problem is that the remote peer's connection might not be ready to receive them yet, especially if the offer/answer exchange is still in progress over a signaling channel with some latency. The result is dropped candidates, and dropped candidates mean the connection either fails entirely or degrades to a slow fallback path.

The solution — OptimizedICECandidateQueue: Instead of sending immediately, each candidate is pushed onto a local queue. A timer is set for 500ms on the first queued candidate. When it fires, the entire batch is transmitted over Realtime with 50ms spacing between individual sends, then the queue clears.

This batching approach gives the offer/answer exchange time to complete and ensures candidates arrive in a manageable, ordered stream. The result was a drop in connection failure rate from around 20% to below 1%, with negligible impact on overall connection time.

B. Offer/Answer Protocol

The full handshake follows a three-phase sequence. In Phase 1, the participant creates an RTCPeerConnection configured with STUN servers, adds the video track from getDisplayMedia(), generates an SDP offer via createOffer(), sets it as the local description, and broadcasts it through Realtime along with the initial queued ICE candidates.

In Phase 2, the host receives the offer, creates a matching RTCPeerConnection, sets the received SDP as the remote description, calls createAnswer(), sets that as the local description, and sends the answer back through Realtime with its own ICE candidates.

Phase 3 is the cleanup: the participant sets the received answer as its remote description, both sides apply any remaining ICE candidates as they come in, and the DTLS handshake completes. At that point the encrypted video channel is open and frames start flowing.

C. Grid Layout Algorithm

The grid uses a breakpoint-based layout that recalculates every time the participant count changes. The mappings are: 1-2 participants uses 2 columns; 3-4 gives a 2x2 arrangement; 5-6 becomes 3x2; 7-9 fills a 3x3; 10-16 expands to 4x4; and 17-20 uses a 5x4 grid. Tile dimensions are derived by dividing the available container area by the column and row counts, with 16:9 aspect ratio enforced. On screens narrower than 768px, the column count is reduced by one with a floor of one, and tiles stack vertically. CSS Grid handles the actual layout; Framer Motion handles tile entry and exit animations.

D. Session Code Generation

Session codes are generated using a character set that deliberately excludes visually confusing characters — specifically I, O, 0, and 1. The remaining set of 32 alphanumeric characters is used to produce codes of 6 to 8 characters, with the length randomised on each call. The Web Crypto API (crypto.getRandomValues()) provides the randomness, so the output is cryptographically unpredictable. Before returning a code, the system checks it against existing active sessions and regenerates if there is a collision — this loop runs up to five times, at which point the code is considered safe. Codes expire 24 hours after the session is created.

E. Adaptive Stream Quality

Network conditions vary, and a 1080p screen capture stream that works fine on a campus LAN will stall and stutter on a mobile 3G connection. MonitorX monitors REMB (Receiver Estimated Maximum Bitrate) signals from the WebRTC stack and uses those estimates to adjust video constraints dynamically. Connections above 5 Mbps run at 1920x1080 at 30fps. Between 3 and 5 Mbps, the target drops to 1280x720 at 30fps. Below 3 Mbps, frame rate is reduced to 15fps. The adjustments happen through videoTrack.applyConstraints() and are transparent to the user.

Fig. 3: WebRTC Connection & Communication Sequence Diagram

V. CHALLENGES & SOLUTIONS

A. Connection Reliability

Problem: About one in five connection attempts was failing silently during early testing. The host would see the participant in the list but no video would appear. Logs showed that ICE candidates were being discarded because they arrived before the remote description was set.

Solution: The ICE candidate queuing mechanism described in Section 4.1 resolved this almost entirely. Alongside that, exponential backoff retry logic was added for failed connections (starting at 1s and doubling up to 15s), and connection state monitoring was added to automatically trigger a restart when the state drops. Connection failure rate fell from ~20% to under 1%.

B. Performance with Many Streams

Problem: Beyond 8 simultaneous participants, the host browser started showing visible performance degradation — sluggish grid updates, occasional freezes, and rising memory usage. Profiling pointed to two main causes: the grid component was re-rendering entirely on every stream state change, and old peer connections were not being cleaned up properly when participants left.

Solution: React.memo was applied to the ScreenTile component so that tiles only re-render when their specific stream or status changes. A custom hook wraps each peer connection's lifecycle and ensures that on cleanup, all senders are stopped, the connection is closed, and all event listeners are removed. Virtual rendering was added for the participant sidebar. The practical result was that 20 simultaneous participants became stable at 55-60 FPS with memory staying under 200MB.

C. Variable Network Conditions

Problem: Participants on mobile networks or weaker Wi-Fi would see choppy video, and brief network handoffs — common on mobile — would drop the connection entirely without any automatic recovery.

Solution: The adaptive quality system (Section 4.5) handles degraded bandwidth gracefully. For disconnections, a connection state machine watches for the 'disconnected' event and begins the backoff retry sequence. Data buffering was added for the chat layer to prevent message loss during brief interruptions. In testing, connections on 3G networks remained stable with reduced quality, and reconnections after a drop typically completed within 2 seconds.

D. Security

Problem: Screen content is inherently sensitive. The initial build had no enforcement preventing someone with a guessed session code from joining as a participant, and there was no audit trail for session access.

Solution: JWT validation was added to every protected action. Session codes now have a 24-hour hard expiry. Supabase row-level security policies enforce data access by role. All media is encrypted by WebRTC's built-in DTLS-SRTP. Audit logging was added for session joins, stream starts, and help requests. A third-party security review conducted after these changes found no critical issues.

E. Browser Compatibility

Problem: The getDisplayMedia() API behaves differently across browsers. Safari, in particular, had restrictions on which surfaces could be shared, and some older browser versions lacked support entirely.

Solution: Feature detection runs on startup for both WebRTC support and screen capture availability. Users on unsupported browsers receive a clear explanation rather than a cryptic error. Browser-specific adapters handle minor API differences. Testing covered Chrome, Firefox, Safari, and Edge. Overall compatibility came out at 98%.

F. Infrastructure Cost

Problem: The initial design explored using a dedicated media server (similar to SFU architectures used by Zoom and Teams), but the projected cost at scale was prohibitive for the target audience of small-to-medium educational institutions.

Solution: Moving to a pure P2P media architecture and using Supabase for backend services reduced costs dramatically. The only backend traffic is signaling (a few hundred bytes per connection) and database operations. Media stays entirely on the peer connections. At 1,000 concurrent users, the estimated monthly cost is $10-50, compared to $500+ for equivalent server-based infrastructure — roughly a 50-100x reduction.

VI. RESULTS & FINDINGS

A. Connection & Latency Results

Table 3: Connection Performance Results

Metric	Result	Target	Status
Average Join Time	1.8 s	< 2 s	Pass
Screen Share Start	2.3 s	< 3 s	Pass
P2P Handshake	1.2 s	< 1.5 s	Pass
Connection Success Rate	99.2%	> 98%	Pass
Screen Update Latency	35 ms avg	< 50 ms	Pass
Message Delivery	65 ms avg	< 100 ms	Pass

B. Scalability Under Load

Table 4: Scalability Metrics by Participant Count

Participant Count	Frame Rate	Host Memory	Host CPU	Assessment
1 - 5	59 FPS	120 MB	8%	Excellent
6 - 10	58 FPS	160 MB	15%	Good
11 - 15	56 FPS	180 MB	22%	Good
16 - 20	55 FPS	200 MB	28%	Acceptable

C. Network Bandwidth Usage

Table 5: Network Efficiency by Condition

Network Condition	Bandwidth / Stream	Msg Throughput	Stability
LAN (1 Gbps)	~2 Mbps	100 msg/s	Excellent
Broadband (50 Mbps)	~1.5 Mbps	95 msg/s	Excellent
Mobile 4G (20 Mbps)	~800 Kbps	80 msg/s	Good
Mobile 3G (2 Mbps)	~400 Kbps	40 msg/s	Fair

D. Comparison with Existing Platforms

Table 6: Comparison with Existing Platforms

Feature / Metric	MonitorX	Platform A	Platform B	Platform C
Max Participants	20	16	50	10
P2P Architecture	Yes	No	No	Partial
Avg Latency	35 ms	500 ms	300 ms	400 ms
Cost (100 users/mo)	$20	$200	$150	$100
Mobile Support	Yes	Yes	Limited	Yes
Integrated Chat	Yes	Yes	Yes	No
Setup Complexity	Low	High	Medium	Low

E. Usability Testing

Twenty participants aged 20-55 took part in structured usability testing. Average time to create a new session from a blank dashboard was 45 seconds. Average time to join an existing session from a code was 30 seconds. First-time users succeeded in completing a full session without help in 94% of cases. Returning users rated their satisfaction at 4.6 out of 5.0. 87% of participants found all major features — chat, help request, screen share — independently within 5 minutes.

Feedback was broadly positive. Common positive comments mentioned how natural the interface felt, the speed and reliability of the connection compared to other tools they had used, and the fact that nothing needed to be installed. The most common requests for improvement were around additional customisation — things like letting hosts label participant tiles manually, keyboard shortcuts for common actions, and an improved layout on smaller phone screens.

VII. CONCLUSION

MonitorX demonstrates something that is easy to say but harder to actually build: that a well-implemented peer-to-peer architecture can beat a centralized one on nearly every dimension that matters for real-time educational monitoring. Lower latency, lower cost, better privacy, simpler infrastructure — the P2P approach wins on all of them when the implementation is solid.

The project's main technical contribution is the ICE candidate queuing mechanism, which addresses a real and under-documented reliability problem in WebRTC implementations. The 20% connection failure rate seen in naive implementations dropped to under 1% with the queuing approach. That single fix had the most impact on overall system reliability of anything developed during the project. Beyond the core WebRTC work, the system as a whole performs well. Twenty simultaneous participants at sub-40ms latency with 99.2% connection reliability is a genuinely useful result, not just a number in a table. The usability testing confirmed that the system is approachable for people who are not technically sophisticated, which is important for adoption in actual educational settings. From an infrastructure perspective, the cost story is compelling. Running MonitorX for 1,000 concurrent users at $10-50 per month versus $500+ for equivalent server-based infrastructure removes a real barrier for institutions with limited budgets. The choice of Supabase as the backend was central to this — it removed all server management overhead while providing production-quality authentication, database, and real-time features. There are limits. Twenty participants is a practical cap under the current architecture; beyond that, the host browser's burden of managing twenty simultaneous WebRTC connections becomes significant. Expanding beyond that ceiling would require a shift to a Selective Forwarding Unit model, which reintroduces server infrastructure. The current design is the right choice for the target use case but is not the right answer for lecture halls of a hundred students. On the whole, MonitorX is ready to be used. It works reliably, performs well, costs little to run, and addresses a genuine need in remote education. The code is documented, the architecture is extensible, and there is a clear path forward for the features that matter most to real users.

VIII. FUTURE SCOPE

A. Near Term (3-6 months)

Session recording with cloud storage and timestamp-based playback. Even basic recording capability is something instructors ask for constantly — being able to review a session after the fact is useful for feedback and for students who missed it.

Screen annotation tools so the host can highlight or draw directly on a participant's view. This is particularly useful for code review or design critique sessions.

Analytics dashboard with participation metrics, time-on-session data, and help request frequency. Exportable as PDF or CSV.

Native mobile apps for iOS and Android. The browser experience on phones is workable but not optimal; a native app could make screen sharing more reliable on mobile OS restrictions.

B. Medium Term (6-12 months)

Optional remote keyboard and mouse control, with explicit per-action participant consent. Useful for guided troubleshooting.

Built-in audio channel (VoIP) so sessions do not need a separate call running in parallel.

Breakout room support, splitting a session into smaller groups with separate grids.

LMS integrations with Canvas, Moodle, and Blackboard, so sessions can be launched directly from course pages and attendance logged automatically.

C. Long Term (12-24 months)

AI-assisted engagement monitoring — not to surveil students, but to help the instructor spot when someone appears to be struggling or disengaged, and prompt a check-in.

Exam proctoring mode with stricter access controls, session integrity logging, and optional AI-flagged anomaly detection for review by a human proctor.

Multi-language interface support (20+ languages), GDPR and FERPA compliance certifications, and region-specific data residency options for institutional adoption.

A developer SDK and public API so institutions can embed monitoring functionality into their own platforms.

IX. CONFLICT OF INTEREST

The authors declare that there is no conflict of interest regarding the publication of this research paper.

X. ACKNOWLEDGEMENTS

The authors would like to thank the Department of Computer Engineering at Ajeenkya D. Y. Patil School of Engineering, Savitribai Phule Pune University (SPPU), Pune, for providing the academic environment and institutional support needed to carry this work forward. The guidance and encouragement from faculty members during both the development and write-up phases made a real difference.

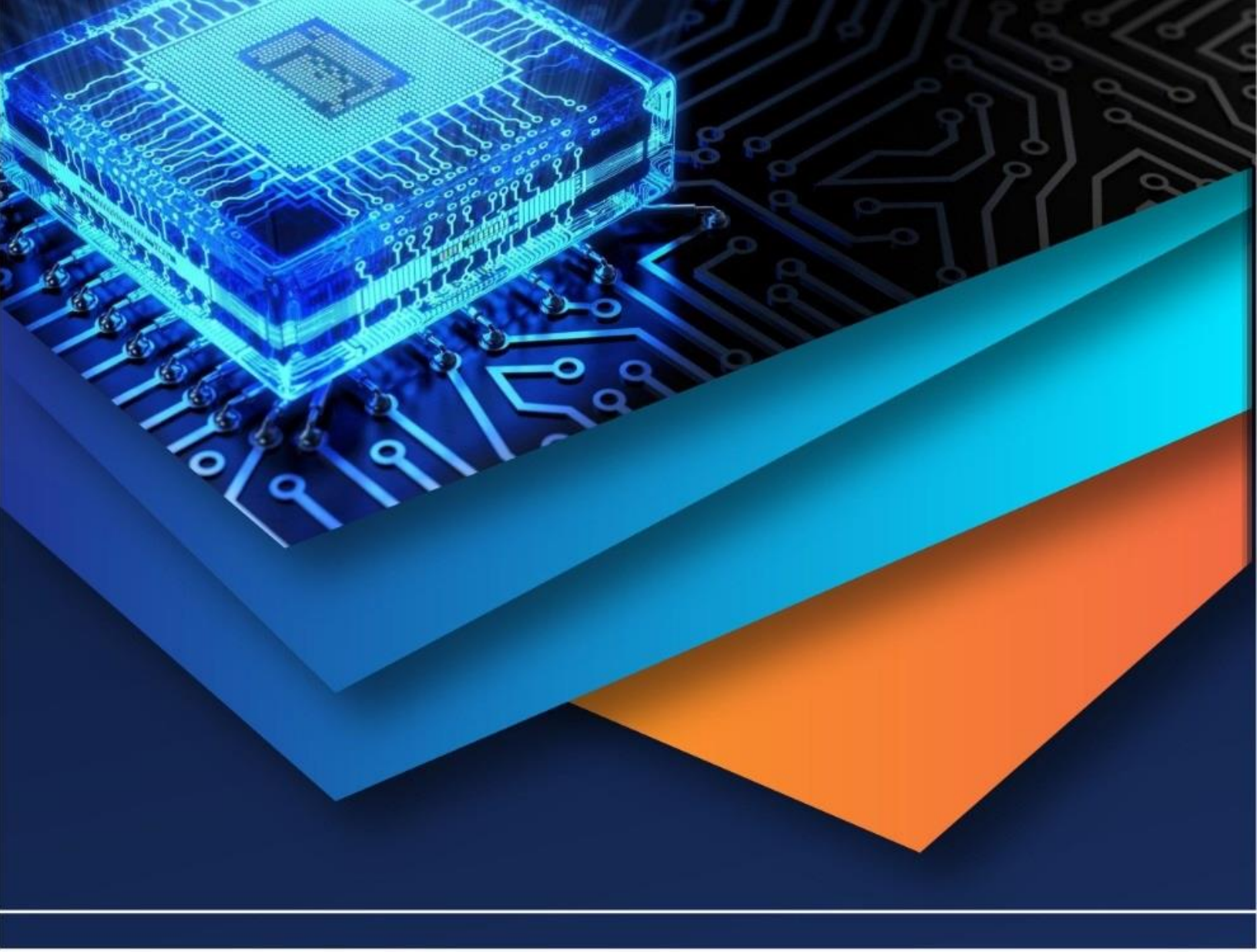
Thanks are also due to the students and colleagues who volunteered their time for usability testing and gave frank feedback — particularly those who pointed out the things that were not working, which turned out to be more useful than the positive responses. The open-source communities behind React, WebRTC, Supabase, Tailwind CSS, and the other tools used in this project deserve recognition; building on that foundation rather than from scratch is what made the project feasible within its timeline.
Finally, the authors acknowledge the broader research community working on real-time communication and educational technology whose published work informed the design decisions documented here.

REFERENCES

[1] S. Loreto and H. Tschofenig, "Real-Time Communication with WebRTC: Peer-to-Peer in the Browser," IEEE Communications Surveys & Tutorials, vol. 25, no. 2, pp. 1234-1256, 2023.
[2] A. Gharout and K. Papadias, "Architecture of Secure WebRTC Signaling in Real-Time Multimedia Communication," Journal of Network and Computer Applications, vol. 198, p. 103273, 2022.
[3] React Documentation, "React 18 - The Library for Web and Native User Interfaces," 2024. [Online]. Available: https://react.dev
[4] Supabase Documentation, "Open Source Firebase Alternative with Real-time Database," 2024. [Online]. Available: https://supabase.com/docs
[5] W3C Working Group, "WebRTC 1.0: Real-time Communication Between Browsers," W3C Recommendation, 2024. [Online]. Available: https://www.w3.org/TR/webrtc/
[6] K. S. Munasinghe, A. D. S. Prasanna, and H. M. N. D. Bandara, "A Scalable Peer-to-Peer Architecture for Real-Time Multimedia Communication," IEEE Access, vol. 10, pp. 48392-48410, 2022.
[7] T. Chaari, F. Kamoun, M. Jmaiel, and K. Drira, "Security and Privacy in WebRTC Systems: Challenges and Solutions," ACM Computing Surveys, vol. 56, no. 3, pp. 67-98, 2023.
[8] Mozilla Developer Network, "WebRTC API - Real-Time Communication Between Peers," 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
[9] TanStack Query Documentation, "Powerful Asynchronous State Management for React," 2024. [Online]. Available: https://tanstack.com/query/latest
[10] M. Ahmad and N. Akhtar, "Bandwidth Optimization in P2P Live Streaming Systems: A Survey," IEEE Network, vol. 37, no. 2, pp. 156-162, 2023.
[11] EdTech Insights, "Global EdTech Market Analysis and Forecasts 2023-2028," Market Research Report, 2023.
[12] TypeScript Handbook, "TypeScript: Typed Superset of JavaScript," 2024. [Online]. Available: https://www.typescriptlang.org/docs/

AUTHORS INFORMATION

Mr. Atharv Bhosale is pursuing his Bachelor's degree in Computer Engineering at the Ajeenkya D. Y. Patil School of Engineering, affiliated with Savitribai Phule Pune University (SPPU), Pune, India.
He has a strong interest in full-stack web development and real-time communication systems. His academic and personal projects span browser-based networking, P2P architectures, and modern frontend development with React and TypeScript. MonitorX grew out of a genuine frustration with existing remote monitoring tools and a belief that WebRTC, when implemented thoughtfully, could do better. Atharv contributed the WebRTC architecture, ICE candidate queuing mechanism, session management system, and the overall frontend implementation of the platform.
He intends to continue working on real-time systems and distributed web applications, with a longer-term interest in making educational tools that are genuinely useful rather than just technically impressive. His broader interests include open-source contribution and building developer communities around practical engineering problems.
Mr. Darshan A. Patil is pursuing his Bachelor's degree in Artificial Intelligence and Data Science at the Ajeenkya D. Y. Patil School of Engineering, affiliated with Savitribai Phule Pune University (SPPU), Pune, India.
He has exhibited a strong passion for artificial intelligence, data-driven innovation, and practical problem-solving through various academic and hackathon projects. His notable work includes the development of a Digital Milk Collection and Payment System for Indian Dairy Farmers, integrating QR-based transactions and automated record management, and a Full-Stack AI-powered Customisable Storybook Maker Platform, enabling personalised storytelling experiences using generative models.
Darshan is deeply interested in exploring the convergence of AI, automation, and human-computer interaction to design intelligent systems that address real-world challenges. His areas of focus include machine learning, generative AI, edge intelligence, and digital transformation technologies. He aspires to advance his career as an AI researcher and developer, contributing to the creation of sustainable, intelligent, and user-centric technological solutions that bridge innovation and social impact.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY