American Research Journal of Computer Science and Information Technology

Volume 8, Issue 1, 24-30 Pages
Research Article | Open Access
ISSN (Online)- 2572-2921
DOI : 10.21694/2572-2921.25005

Architectural Approaches to Building Real-Time Web Applications Based on WebSockets, SSE, and WebRTC

Andrei Chmelev

Senior Software Engineer, Lowe's Indian Trail, NC 28079, US.

ABSTRACT

This article reviews contemporary architectural approaches to building real-time web applications that require instant data updates. It highlights a comparative analysis of key technologies—WebSockets, Server-Sent Events (SSE), and WebRTC—providing guidance on selecting the most suitable solution for scenarios such as chat systems, online editors, and multiplayer games. In addition, it explores strategies for scaling and load balancing real-time connections, as well as mechanisms to ensure reliable data delivery. Typical architectures—such as Node.js with Redis Pub/Sub or Kafka—are presented, along with recommended methods for monitoring and logging real-time data streams. Code snippets illustrate practical implementation techniques to help developers create robust and efficient real-time features within various infrastructures.

KEYWORDS: *real-time web applications, websockets, server-sent events, webrtc, scalability, load balancing, reliability, pub/sub, monitoring, logging*

INTRODUCTION

The development of network technologies and the growing demands of users for interactivity make the implementation of real-time data update mechanisms critically important for web applications. Chats, online document editors, notification systems, and multiplayer games all require effective approaches to real-time message exchange. Modern protocols and tools such as WebSockets [1], Server-Sent Events (SSE) [2], and WebRTC [3] allow developers to choose the optimal solution depending on the use case—whether two-way communication, media (audio/video) transmission, or broadcast notifications are required.

In addition to choosing the main communication protocol, it is now extremely important to design mechanisms for scaling and load balancing of real-time connections under high loads, as well as to ensure reliable data delivery. For this purpose, message brokers (Redis Pub/Sub [4], Kafka [5]) and various fault-tolerance approaches (ACK/Retry, idempotency, etc.) are widely used. Monitoring and logging tools (Prometheus [9], Grafana [10], and distributed tracing systems [11,12]) have also become essential, enabling timely detection of issues and effective management of complex microservice architectures.

RESEARCH OBJECTIVES

The goal of this article is to systematize knowledge about modern methods and tools for creating real-time web applications. In particular, it addresses the following tasks:

1. Comparative analysis of key protocols: Determine under which conditions WebSockets, SSE, and WebRTC are most effective, as well as their advantages and limitations.
2. Scaling and load balancing strategies: Describe approaches to horizontal scaling, including sticky sessions, decentralized state storage, and the use of message brokers.
3. Reliable delivery mechanisms: Examine practical methods for ensuring guaranteed message delivery, including Pub/Sub patterns, acknowledgments (ACK), retries, and idempotency.
4. Practical examples and recommendations: Demonstrate through specific code snippets and diagrams how these solutions can be implemented in real-world infrastructures, as well as how to configure monitoring and logging.

RESEARCH METHODOLOGY

In this work, a review-analytical approach with elements of comparative analysis of key technologies for creating real-time web applications was applied. The methodology included several stages.

1. At the first stage, the official standards for WebSockets, SSE, and WebRTC were analyzed, as well as documentation for Redis and Kafka. Their technical capabilities, limitations, and main usage scenarios were studied.

2. For the comparative analysis, data from open tests and architectural recommendations were collected, taking into account interaction models, scaling (sticky sessions, NAT, SFU/MCU), and various types of loads.
3. Additionally, practical cases based on Node.js, Redis Pub/Sub, Kafka, and monitoring systems (Prometheus, Grafana, Jaeger, OpenTelemetry) were considered. Real solutions were examined to identify typical bottlenecks and to test the effectiveness of the proposed approaches.
4. At the final stage, recommendations were formulated on the choice of technologies, the setup of horizontal scaling, and ensuring reliable delivery (Pub/Sub, ACK, retries, idempotency). All findings were based on comparing theoretical data with practical observations.

Combining these steps made it possible to substantiate the solutions under consideration and to provide a holistic view of the design and operation of real-time web applications in distributed systems.

Technologies for two-way communication in web applications

In the modern web, three approaches are widely used to implement real-time functionality: WebSockets, Server-Sent Events (SSE), and WebRTC. Each one addresses specific tasks — from full-fledged two-way message exchange to transmitting audio and video streams directly between browsers. The particular choice depends on the scenario (chats, online games, collaborative editors, video conferences) and infrastructure features (load balancing, presence of proxy servers, NAT traversal support, etc.). A comparative analysis of the key characteristics and applicability of these technologies is presented in Table 1.

Table 1. Comparative overview of WebSockets, SSE, and WebRTC

Criterion	WebSockets	SSE (Server-Sent Events)	WebRTC
Type of connection	Full-fledged two-way connection (full-duplex) after HTTP Upgrade	One-way data stream server → client, uses text/event-stream	P2P connection between clients (with possible involvement of STUN/TURN servers)
Use cases	Chats, online games, real-time notification systems, financial exchanges	News feeds, notifications, chats with infrequent client-to-server data, monitoring	Audio/video conferences, direct file exchanges, interactive streams (voice, video)
Advantages	- Asynchronous full-duplex exchange- Minimal latency - Broad browser support	- Simplicity of implementation and deployment - Works over regular HTTP/2- Automatic “reconnect”	- Reduces server load, data goes “client-to-client”- Supports audio, video, file sharing
Drawbacks	- Requires special attention to load balancing (sticky sessions) - Can cause proxying difficulties	- One-way (can’t send data from client to server through the same channel) - Less efficient for large-scale broadcasting	- Complex NAT traversal setup (STUN/TURN)- For large broadcasts, server components (SFU/MCU) are still needed
Scaling features	- Need to ensure the connection is tied to a specific server or use a distributed cache	- Possible to use CDN/edge servers for SSE retransmission- Straightforward “horizontal” scaling with proper configuration	- Need to scale the TURN server (when many peers are behind NAT) - Large projects often introduce SFU/MCU

WebSockets are convenient for two-way transmission of small messages in real time (e.g., in chats), SSE is well suited for simple “push” notifications from the server to the client, and WebRTC solves the problem of direct audio and video transmission without heavy server load.

Typical architectures and reliability mechanisms

This section describes the most common approaches to organizing and scaling real-time applications, as well as mechanisms for improving the reliability of message delivery. In practice, multiple technologies are often combined — for example, Node.js + Redis for real-time broadcasting (Pub/Sub), Kafka for guaranteed delivery and event analysis (event streaming), and advanced ACK/Retry mechanisms with idempotency. A comparative analysis of the presented solutions is provided in Table 2.

Node.js + Redis Pub/Sub

One of the common solutions is to use Node.js to handle incoming connections (via WebSockets or SSE) and Redis as a Pub/Sub channel for broadcasting messages. Below in (Figure. 1) is a simplified diagram showing how a message is sent from one client to another via Redis’s Pub/Sub mechanism:

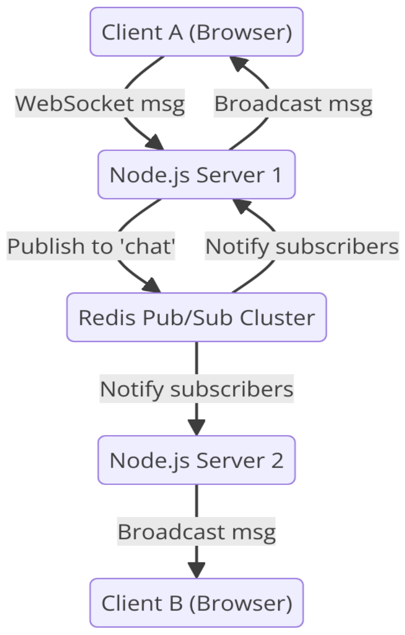

Figure 1. Node.js + Redis Pub/Sub interaction diagram.

Client A sends a message via WebSocket to the Node.js server, which publishes it to a Redis channel (for example, “chat”). Redis notifies all servers subscribed to that channel, and those servers then transmit the received data to their connected clients, for instance, Client B. If necessary, the originating server confirms delivery to Client A or relays updated information to other connected clients. This setup simplifies horizontal scaling and, with Redis clustering, ensures high fault tolerance.

Using Kafka or similar message brokers

In larger systems, Kafka or similar solutions (RabbitMQ, NATS, etc.) are often employed to ensure reliable delivery. Kafka provides “guaranteed” delivery and message storage in topics [5], allowing messages to be re-read in the event of a failure and thus eliminating the risk of losing important data. Each node (Node.js or any other) can both publish messages to topics and subscribe to them, passing updates to clients via WebSockets or SSE. Figure 2 shows a simplified interaction scheme when using Kafka as a message broker:

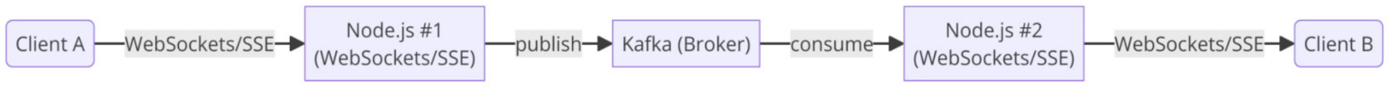

Figure 2. Simplified interaction diagram when using Kafka.

Client A sends a message via WebSockets (or SSE) to the Node.js #1 node, which “produces” this message to the corresponding Kafka topic. Node.js #2, having “consumed” the same topic, receives the new message from Kafka and forwards it to its client, Client B, via WebSockets or SSE. Thanks to message storage in topics and Kafka’s acknowledgment mechanisms, delivery is guaranteed even in the event of failures, which is especially crucial for high-load and distributed systems.

Acknowledgment and retry mechanisms

For mission-critical messages (for example, transaction information), basic Pub/Sub functionality may not suffice. Additional mechanisms are needed for delivery acknowledgments, idempotency, and message log storage. Figure 3 shows a simplified sequence diagram illustrating the process of publication, ACK, and potential message redelivery.

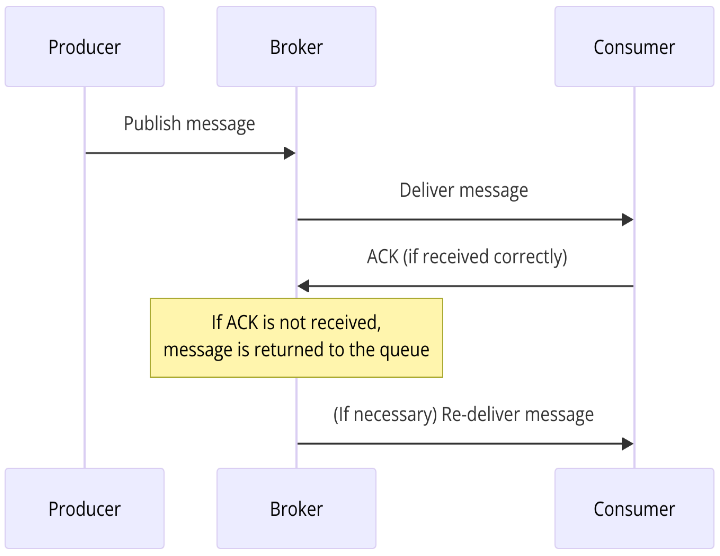
Producer
Broker
Consumer
Publish message
Deliver message
ACK (if received correctly)
If ACK is not received, message is returned to the queue
(If necessary) Re-deliver message
Producer
Broker
Consumer

Figure 3. Acknowledgment (ACK) and retry mechanism in a message broker.

The Producer publishes a message to the Broker (Kafka, RabbitMQ, or another solution), which delivers the message to the Consumer (subscriber). Upon successful receipt, the Consumer sends an ACK back to the Broker. If the acknowledgment is not received within the specified time, the Broker returns the message to the queue and redelivers it (retry). The Consumer (client) must explicitly acknowledge (ACK) receipt of the message; otherwise, a redelivery is initiated. Each message is accompanied by a unique identifier so that redelivery does not produce duplicates (idempotency). In the case of Kafka, journal-based storage allows you to "rewind" message processing to the required offset and reread unread data. This mechanism ensures reliable delivery of critical events and minimizes the risk of message loss or duplication.

Table 2. Comparison of Node.js + Redis, Kafka, and ACK/Retry mechanisms

Solution / Mechanism	Advantages	Disadvantages
Node.js + Redis (Pub/Sub)	- Easy to deploy (Node.js applications + Redis cluster) - Convenient for horizontal scaling - Fast message exchange between multiple servers	- At high loads, Redis clustering is required - No built-in guaranteed delivery; additional queuing (see ACK/Retry) is needed
Kafka (or equivalents)	- "Guaranteed" delivery with the ability to re-read messages - Horizontal scaling by adding brokers-Message history storage (topics)	- More complex infrastructure (managing clusters, partitions) - High resource requirements for large volumes - Requires detailed configuration (ZooKeeper / KRaft, ACL, etc.)
ACK/Retry + Idempotency	- Eliminates loss or duplication of critical messages - Ability to "rewind" state (in case of Kafka) - Guaranteed delivery in case of failures	- Complicates application or broker logic - Must track delivery statuses and handle retries - Requires unique identifiers and message log storage

SCALING AND LOAD BALANCING OF REAL-TIME CONNECTIONS

Horizontal scaling and sticky sessions

When using WebSockets, it is important to maintain the client's "attachment" (sticky sessions) to a particular server throughout the entire connection. This allows for correct handling of incoming messages and sending responses without reconnecting to another node. However, this approach complicates load distribution among servers. In more advanced scenarios, a "decentralized" structure can be organized, where any node can handle requests from any client, and the connection state is stored in a shared distributed memory (for example, Redis).

Edge servers and CDNs for SSE

For implementing broadcast SSE streams, scaling can be done using CDNs or edge servers (NGINX, Cloudflare, etc.) [7,8]. This approach makes it possible to "bring" streams closer to users, reducing latency and offloading the central server. It is especially relevant when delivering updates to a large number of subscribers.

Load balancing for WebRTC

In WebRTC, if there is no server intermediary (SFU/MCU), most of the traffic goes directly between clients (P2P), and STUN/TURN [3] is used to overcome NAT. As the number of connections and the volume of transmitted media data increase, the load shifts to the TURN server, which must be scaled (for example, by using a Coturn cluster) and balanced. This approach ensures stable connections and minimal latency even in large projects with thousands of simultaneously connected users.

A generalized scaling scheme

Below (Figure. 4) is a schematic diagram that demonstrates the balancing principles for three real-time interaction technologies: WebSockets (with sticky sessions or decentralized state storage), SSE (through CDN/edge servers), and WebRTC (with a dedicated STUN/TURN server).

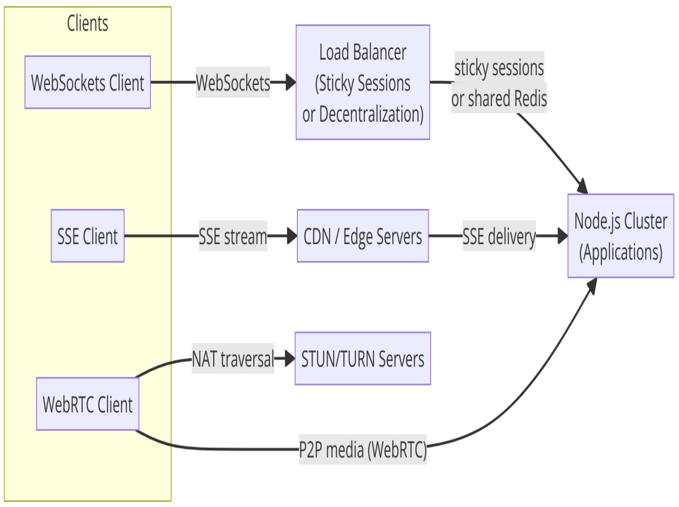

Figure 4. A generalized diagram of real-time connection scaling.

This approach to scaling real-time connections takes into account the specifics of each protocol and makes it possible to efficiently distribute the load in systems with a large number of concurrent users.

Monitoring and logging of real-time streams

Effective monitoring, logging, and tracing tools make it possible to promptly identify bottlenecks, improve fault tolerance, and ensure predictable service quality [9–12].

Figure 5 shows a summary diagram illustrating the main channels for collecting data on the operation of a real-time system.

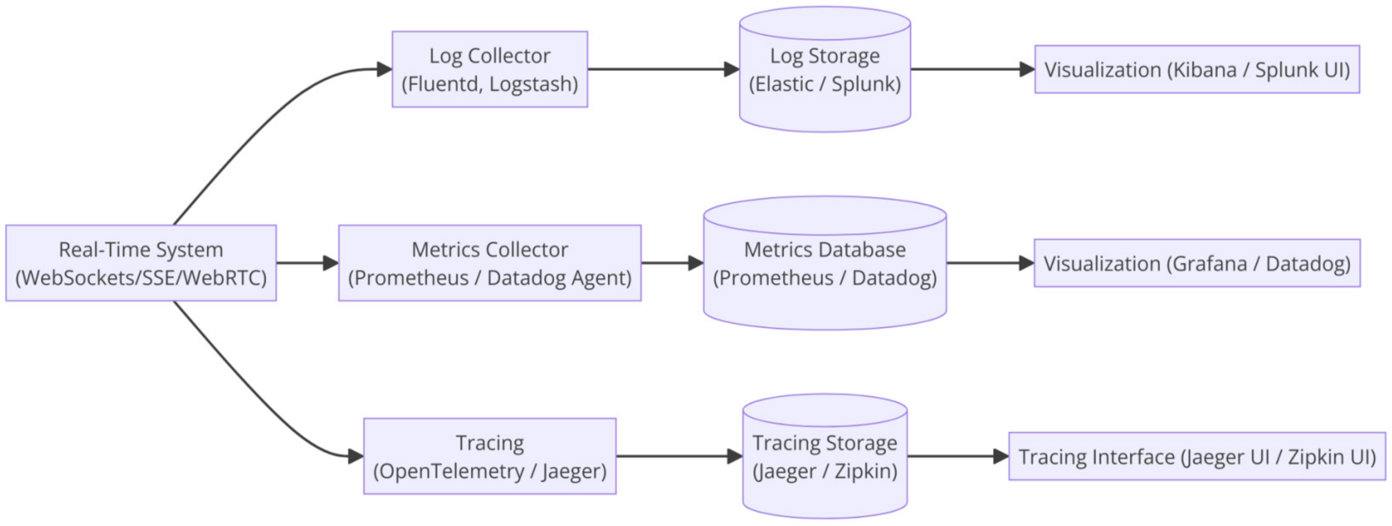
Log Collector (Fluentd, Logstash)
Log Storage (Elastic / Splunk)
Visualization (Kibana / Splunk UI)
Real-Time System (WebSockets/SSE/WebRTC)
Metrics Collector (Prometheus / Datadog Agent)
Metrics Database (Prometheus / Datadog)
Visualization (Grafana / Datadog)
Tracing (OpenTelemetry / Jaeger)
Tracing Storage (Jaeger / Zipkin)
Tracing Interface (Jaeger UI / Zipkin UI)

Figure 5. Collection and analysis of metrics, logs, and tracing in a real-time system.

Real-time metrics

It is recommended to track the following key metrics:

- Number of active connections (via WebSocket, SSE, WebRTC).
- Number of messages per second (incoming and outgoing).
- Average round-trip time (RTT) for sending/receiving messages.

Prometheus [9] (in conjunction with Grafana [10]) or other SaaS solutions (Datadog, New Relic) are commonly used for collecting and visualizing metrics. An agent (or exporter) is deployed alongside the application to gather technical data and send it to the metrics server.

Event logging

When debugging and investigating incidents, it is important to maintain logs containing information about connection statuses and sent/received messages.

- For large volumes of data, it is better to use centralized systems such as the Elastic Stack, Splunk, or Graylog, which enable convenient filtering of logs by key fields (e.g., session_id, user_id).
- It is crucial to strike a balance between the level of detail in logs and the load on the system: excessive logging can slow down the application.

Distributed Tracing

In a microservice architecture, messages can pass through a chain of services, and simple logging is often not enough.

- Distributed tracing (Jaeger [11], Zipkin, OpenTelemetry [12]) reveals the path of a message between services as well as the processing time at each stage.
- This helps detect bottlenecks and long-running segments (hot spots), ultimately optimizing the entire real-time flow.

A comprehensive approach (metrics + logs + tracing) provides an all-encompassing understanding of how the real-time system operates and enables rapid response to issues that arise in production.

Code Examples

Let's look at brief examples of implementing real-time functionality in Node.js using various approaches—from the WebSockets + Redis combination to SSE (Server-Sent Events).

Node.js WebSocket Server with Redis Pub/Sub

Below is a simplified example of a Server.js file, in which Node.js handles incoming WebSocket connections, and Redis is used for message broadcasting (Pub/Sub):

```
const http = require('http');
const WebSocket = require('ws');
const redis = require('redis');
// Create an HTTP server
const server = http.createServer();
const wss = new WebSocket.Server({ server });
// Connect to Redis
const redisSub = redis.createClient();
const redisPub = redis.createClient();
// Subscribe to the 'chat' channel
redisSub.subscribe('chat');
// When a message is received from Redis, forward it to all connected clients
redisSub.on('message', (channel, message) => {
 if (channel === 'chat') {
  wss.clients.forEach(client => {
   if (client.readyState === WebSocket.OPEN) {
    client.send(message);
   }
  });
 }
});
// Handle incoming WebSocket connections
wss.on('connection', ws => {
 // When a message is received from the client, publish it to Redis
 ws.on('message', msg => {
  redisPub.publish('chat', msg);
 });
});
server.listen(3000, () => {
 console.log('WebSocket + Redis server listening on port 3000');
});
```

Here, every message received from a client (via WebSocket) is “replicated” through the chat channel in Redis, after which all servers (when there are multiple) get notified and relay the update to their connected clients.

SSE Example (code snippet)

The following snippet shows how to set up a simple SSE (Server-Sent Events) server in Node.js. This server sends a new “event” with the current time to clients every 5 seconds:

```
const http = require('http');

http.createServer((req, res) => {
 if (req.url === '/stream') {
  res.writeHead(200, {
   'Content-Type': 'text/event-stream',
   'Cache-Control': 'no-cache',
   'Connection': 'keep-alive',
  });
    // Send an event every 5 seconds
  const intervalId = setInterval(() => {
   const data = { time: new Date().toISOString() };
   res.write(`data: ${JSON.stringify(data)}\n\n`);
  }, 5000);
  // Free resources when the connection closes
  req.on('close', () => {
   clearInterval(intervalId);
  });
 }
}).listen(4000, () => {
 console.log('SSE server running on port 4000');
});
```

To receive SSE messages in the browser, simply connect to http://localhost:4000/stream via EventSource:

```
<script>
 const evtSource = new EventSource('/stream');
 evtSource.onmessage = (e) => {
  console.log('New SSE data:', e.data);
 };
</script>
```

The client will automatically receive updates every 5 seconds. When the tab is closed or the connection is lost, EventSource will attempt to reestablish the connection, making SSE convenient for tasks involving regular data broadcasts from server to client.

CONCLUSION

Developing real-time web applications requires a deep understanding of network protocols and architectural solutions. WebSockets provide full-fledged two-way communication, SSE simplifies one-way data streaming from server to client, and WebRTC enables direct P2P interaction (including audio and video streams). The choice of a specific approach depends on the particular use case, the required transmission speed, the volume of data, and the network topology.

In high-load environments, well-thought-out scaling and fault-tolerance mechanisms are essential for success: using message brokers (Redis Pub/Sub, Kafka), implementing acknowledgment mechanisms (ACK, retry), and organizing either sticky sessions or decentralized state storage when working with WebSockets. To maintain a stable service level, it is necessary to track key metrics in real time (RTT, number of connections, traffic volume) and maintain centralized logging; in complex microservice architectures, distributed tracing should be used.

Thus, a comprehensive approach that includes choosing the optimal technology (WebSockets, SSE, or WebRTC), a reliable Pub/Sub infrastructure, and monitoring tools makes it possible to build scalable, fault-tolerant, and truly "live" web systems that meet the requirements of today's internet.

REFERENCES

1. Fette I., Melnikov A. The WebSocket Protocol (IETF RFC 6455). 2011.Available from: https://datatracker.ietf.org/doc/html/rfc6455 Accessed: 01.09.2025
2. Hickson I. Server-Sent Events. WHATWG. n.d. Available from: https://html.spec.whatwg.org/multipage/server-sent-events.html
3. Bergkvist A., Burnett D., Jennings C., Narayanan A. WebRTC 1.0: Real-time Communication Between Browsers. W3C Working Draft. n.d. Available from: https://www.w3.org/TR/webrtc/
4. Redis Labs. Redis Pub/Sub Documentation. 2025. Available from: https://redis.io/docs/manual/pubsub/
5. Apache Kafka. Apache Kafka Documentation. 2025. Available from: https://kafka.apache.org/documentation/
6. Node.js Documentation. 2025. Available from: https://nodejs.org/en/docs
7. NGINX Documentation. 2025. Available from: https://nginx.org/en/docs/
8. Cloudflare Documentation 2025. Available from: https://developers.cloudflare.com/
9. Prometheus Documentation. 2025. Available from: https://prometheus.io/
10. Grafana Documentation. 2025. Available from: https://grafana.com/
11. Jaeger Documentation. 2025. Available from: https://www.jaegertracing.io/docs/
12. OpenTelemetry Documentation. 2025. Available from: https://opentelemetry.io/

Citation: Andrei Chmelev, "Architectural Approaches to Building Real-Time Web Applications Based on WebSockets, SSE, and WebRTC", American Research Journal of Computer Science and Information Technology, Vol 8, no. 1, 2025, pp. 24-30.