WHITEPAPER

# THE PROS AND CONS OF BUILDING REAL-TIME APPLICATIONS WITH SOCKET.IO

PubNub

## Introduction

More and more businesses are leveraging chatbots and video conferencing to power real-time customer interactions. As customers embrace these new technologies, development teams must ensure applications can support increasing demand. But that's easier said than done—numerous performance and security issues come with unregulated traffic spikes.

Open-source libraries like Socket.io make it easier for dev teams to quickly and reliably implement real-time capabilities. However, socket infrastructure isn't reliable when managing increasing workloads. So, how can dev teams ensure applications can meet growing demands in realtime?

Before we jump into the details, let's start by understanding what it means to scale your application.

## What is scalability?

Scalability is a system's ability to adapt its performance to changes in the number of users, transactions, or data volume. It allows systems to handle two scenarios efficiently:

1. When your workload increases, you can scale up to support new requirements. Your application's performance remains consistent, and customers can enjoy high-quality user experiences.
2. When your workload decreases, you can scale down so the application uses only the required resources and runs cost-effectively.

## Challenges of scaling real-time applications

Scaling real-time applications like a chat app, chat room, or multiplayer game comes with challenges.

### Performance issues

Scaling up allows your system to manage increasing workloads and adds additional server load. Estimating workloads accurately enables dev teams to evaluate if the current server architecture can support the extra workload.

### Infrastructure limitations

As the number of users increases, your application may underperform. Available resources may not be able to support the surge in demand. Ensure your infrastructure can handle the expanding workload to offer exceptional user experiences.

### Data management challenges

Effective data management fuels real-time capabilities. Design your system to easily scale up to handle increasing data volumes with more users and requests.

### Increased costs

Scalability can increase application development costs. Addressing all the above challenges adds to the overall cost, considering investments in new infrastructure, hardware, software, and labor.

## Benefits of a scalable real-time system

The benefits scalable real-time systems offer make them worth the investment.

### Improved convenience

Developers can save time on additional development by designing the system to scale server capacity to meet workload requirements with just a few clicks.

### Increased speed and flexibility

Scalability allows real-time applications to respond faster to spikes in demand. Extra resources are only used when needed.

### Reduced cost

Although implementing scalable solutions can be expensive, you can design your applications to scale *only when necessary*. This way, you only pay for what you use and avoid purchasing extra equipment.

## What is Socket.io?

Created in 2010, Socket.io is an open-source library that facilitates full-duplex, bi-directional communication between web clients and servers. It's very similar to WebSockets. In fact, it builds on the WebSockets protocol, providing additional capabilities like automatic reconnection and fallback to HTTP long polling.

Although most browsers today support WebSockets, Socket.io is still viable. So even if you decide to use WebSockets for your web application eventually, you'll need to implement features like reconnection, acknowledgments, and broadcasting.

## Key features of Socket.io

Socket.io offers battle-tested features for creating reliable real-time applications.

### HTTP long-polling fallback

HTTP long polling is a variation of standard polling. It duplicates an HTTP server pushing messages to a client (or browser) more efficiently.

The Socket.io framework first tries to establish a long polling connection, and then tries to upgrade to a WebSocket connection if possible. It uses this approach because establishing a WebSocket connection isn't guaranteed with firewalls and antivirus software in play.

### Automatic reconnection

When using WebSockets, the connection can be interrupted under particular conditions without the client and the server knowing.

Socket.io works around this by including a heartbeat mechanism that periodically checks the connection status. If the client gets disconnected, it automatically reconnects to the server and uses a back-off delay to prevent multiple reconnection attempts.

### Packet buffering

If the server disconnects from the server, Socket.io automatically continues buffering events and sends them when the client reconnects. However, this could result in a huge spike in events. Using connected attributes for the Socket instance and volatile events, you can prevent this.

### Acknowledgments

Socket.io offers a request-response application programming interface (API) called Acknowledgements. It is a convenient way to send an event and receive a response. You can even design custom responses to events and add request timeouts to make the process more efficient.

### Broadcasting

As the name suggests, Broadcasting is the process of sending an event to all connected clients or a subset of clients. This also works when scaling to multiple Socket.io servers.

### Multiplexing

Multiplexes (called Namespaces in Socket.io) allow you to split the logic of your application over a single shared connection. It is useful when creating channels only select users can access, like admin or membership channels. Each multiplex will have its own event handlers (emitters and listeners), rooms, and middleware.

## How does Socket.io work?

Socket.io uses two transport methods to enable socket connections:
- HTTP long-polling
- WebSockets

Socket.io automatically selects the best available option, depending on browser and network capabilities. But usually, the HTTP long-poll connection is established first, followed by the WebSocket connection.

*NOTE: Socket.io goes with HTTP long polling first because, although more popular, WebSockets aren't supported on all browsers. So, long-polling is done first to guarantee successful connections and reduce load times.*

When the WebSockets connection is established, HTTP long-polling will act as the fallback so you can confidently continue delivering great real-time experiences.

## Socket.io framework

The Socket.io codebase is divided into two distinct layers: Engine.io and the Socket.io protocol.

Engine.io is the underlying transport layer used by Socket.IO. It's a realtime bidirectional communication engine that enables a persistent connection between the server and the client. Engine.io provides the foundation for Socket.IO's real-time capabilities by handling the low-level communication protocol and managing the connection between the client and the server.

Engine.io uses various transport mechanisms, such as WebSocket, HTTP long-polling, and AJAX, to ensure reliable and efficient client and server communication. It automatically selects the most suitable transport mechanism based on the capabilities of the client and the server.

WebSocket provides a full-duplex communication channel over a single TCP connection. This allows for real-time bidirectional communication with low latency and overhead.

In cases where WebSocket is not supported or available, Engine.io falls back to HTTP long-polling, which involves the client continuously sending requests to the server and the server holding the response until new data is available. This allows for real-time communication even in environments where WebSocket is not supported, such as older web browsers or restrictive network configurations. AJAX is another fallback mechanism used by Engine.io. It involves sending asynchronous HTTP requests from the client to the server regularly to check for new data. While AJAX-based communication is not as efficient as WebSocket or long-polling, it can still provide a basic level of real-time functionality in situations where other mechanisms are unavailable.

The Socket.io protocol delivers additional features (automatic reconnection, packet buffering, acknowledgments, etc.) using the communication channel established by Engine.io
However, you need to host and maintain your cluster of servers when building out your Socket.io application.

Check out how PubNub can help you establish a reliable Socket.io framework.

# Architectural Patterns & Considerations

## Socket.io Architectural Patterns

When building real-time apps, developers often turn to Socket.IO for its powerful features and flexibility. However, it's important to consider the architectural patterns that can be implemented with Socket.IO to ensure a scalable and secure platform. Here are some to consider:

### Server-Centric Architecture

In this pattern, the server plays a central role in handling and managing real-time communication. Clients connect to the server using Socket.IO and send messages that are broadcast to other connected clients. The server is responsible for maintaining the state of each client and managing the connections. This architecture is useful for applications where the server needs full control over the communication and ensure data consistency.

To set up a server-centric architecture with Socket.IO, you can follow the steps outlined below:

#### STEP 1: Install Socket.IO

First, make sure you have Node.js installed on your system. Then, you can install Socket.IO by running the following command in your terminal or command prompt:

```
npm install socket.io
```

#### STEP 2: Initialize Socket.IO in your server code

In your server-side code, import the Socket.IO module and initialize it with your server instance. Here's an example using Express.js as the server framework: *(see next page)*

```
const express = require(‘express’);
const app = express();
const http = require(‘http’).createServer(app);
const io = require(‘socket.io’)(http);
// Your server code here
http.listen(3000, () => {
    console.log(‘Server listening on port 3000’);
});
```

### STEP 3: Handle client connections and messages

With Socket.IO initialized, you can start handling client connections and messages. Here’s an example of how you can listen for client connections and receive and send messages:

```
io.on(‘connection’, (socket) => {
    console.log(‘A client connected’);
    // Listen for incoming messages from the client
    socket.on(‘chat message’, (message) => {
        console.log(‘Received message:’, message);
        // Broadcast the message to all connected clients
        io.emit(‘chat message’, message);
    });
    // Handle client disconnections
    (socket) => {
        console.log(‘A client disconnected’);
    });
});
```

## STEP 4: Set up client-side Socket.IO connection

You can set up a Socket.IO connection on the client-side by including the Socket.IO library in your HTML file and connecting to the server. Here's an example:

```
// Connect to the server
const socket = io();
// Handle form submission
document.getElementById(‘chat-form’).addEventListener(‘submit’, (e) => {
    e.preventDefault();
    const messageInput = document.getElementById(‘message-input’);
    const message = messageInput.value;
    // Send the message to the server
    socket.emit(‘chat message’, message);
    // Clear the input field
    messageInput.value = ‘’;
});
// Listen for incoming messages from the server
socket.on(‘chat message’, (message) => {
    const messageElement = documentquerySelector(‘#messages’);
    const newMessage = document.createElement(‘li’);
    newMessage.textContent = message;
    messageElement.appendChild(newMessage);
});
```

## STEP 5: Enhance the chat application

You can enhance your chat application by adding features like user authentication, private messaging, and typing indicators. Here are a few examples:

### User authentication:

You can use a library like Passport.js to handle user authentication and authorization in your chat application. This will allow you to authenticate users and restrict access to certain features or chat rooms.

### Private messaging:

You can implement private messaging by creating separate chat rooms for individual users or groups. When a user sends a chat message, you can use Socket.IO to only send the message to the specific recipient(s).

**Typing indicators:**

To show typing indicators in your chat application, you can listen for the "typing" event on the client side and emit the event when a user starts or stops typing. On the recipient's end, you can display a typing indicator to indicate that the sender is currently typing a message.

### Peer-to-Peer Architecture

In a peer-to-peer architecture, Socket.IO establishes direct connections between clients without relying heavily on the server. Clients can send messages directly to each other without going through the server, which reduces latency and server load. This architecture is suitable for applications where clients need to communicate with each other directly, such as multiplayer gaming or file-sharing applications.

To set up a Peer-to-Peer (P2P) architecture with Socket.IO, you can follow these steps:

## STEP 1: Understand Peer-to-Peer Architecture

Peer-to-peer architecture allows direct communication between clients without needing a central server. In this architecture, each client can act as a client and a server, enabling them to send and receive data directly.

## STEP 2: Set up Socket.IO Server

First, you must set up a Socket.IO server to handle the initial connection and facilitate the P2P communication. You can use Node.js and the Socket.IO library to create the server. Install Socket.IO using npm:

```
npm install socket.io
```

Create a server.js file and import Socket.IO:

```
const io = require('socket.io')();
```

Set up the connection event listener:

```
io.on('connection', (socket) => {
    // Handle incoming connections
});
```

## STEP 3: Establish P2P Connection

Inside the connection event listener, you can handle the logic for establishing the P2P connection between clients. When a client connects to the server, you can save its socket ID and listen for specific events.

To establish a P2P connection, you can emit a custom event from the client side when a user wants to initiate a connection. For example, when a user wants to start a chat with another user:

```
socket.emit('start-chat', otherUserId);
```

On the server side, you can listen for this event and handle the logic to establish the P2P connection between the two clients. You can use the socket IDs to send messages directly between the clients without involving the server.

## STEP 4: Handle P2P Communication

Once the P2P connection is established, you can handle the communication between clients. You can listen for custom events on both the client and server sides to exchange messages, files, or any other data.

For example, when a client wants to send a message to another client: *(see next page)*

```
socket.emit('send-message', { recipientId: otherUserId, message: 'Hello!' });
```

On the server-side, you can listen for this event and send the message to the recipient client:

```
socket.on('send-message', ({ recipientId, message }) => {
    // Find the recipient client by their socket ID
    const recipientSocket = io.sockets.connected[recipientId];
    // Send the message to the recipient client
    recipientSocket.emit('receive-message', { senderId: socket.id, message });
});
```

### STEP 5: Handle P2P Disconnection

Lastly, you need to handle P2P disconnections. When a client disconnects from the server, you should remove its socket ID from the saved connections. You can use the 'disconnect' event to handle this:

```
socket.on('disconnect', () => {
    // Remove the socket ID from the saved connections
});
```

This code snippet provides a basic example of how to set up a real-time chat application using the Socket.IO library in JavaScript. The steps outlined guide developers through establishing peer-to-peer connections, handling communication between clients, and managing disconnections.

### Hybrid Architecture

The hybrid architecture combines both server-centric and peer-to-peer patterns. Clients connect to the server using Socket.IO but can also establish direct connections when necessary. This allows for a balance between centralized control and direct communication between clients. Hybrid architectures, such as video conferencing or social networking platforms, are commonly used in applications requiring real-time broadcasting and one-on-one communication.

To set up a hybrid architecture with Socket.IO, you can follow these steps:

### STEP 1: Design your architecture

Before starting the implementation, it is important to design your hybrid architecture. Determine which components of your application will be running on the server side and which will be running on the client side. This will help you understand how Socket.IO fits into your overall architecture.

### STEP 2: Set up the server

On the server side, you must set up a Socket.IO server to handle real-time communication. You can use a framework like Node.js with Express to create the server. Install the Socket.IO library and initialize it in your server code.

### STEP 3: Establish peer-to-peer connections

To establish peer-to-peer connections, you must save the socket IDs of connected clients on the server. When a client connects to the server, their socket ID is saved. This can be done using the 'connection' event in Socket.IO.

### STEP 4: Handle peer-to-peer communication

Once the peer-to-peer connection is established, clients can communicate directly without involving the server. Clients can send messages to each other by emitting a custom event, such as 'send-message', from the client side. Listen for this event on the server side and send the message to the recipient client using their socket ID.

### STEP 5: Handle P2P disconnection

When a client disconnects from the server, their socket ID should be removed from the saved connections to avoid memory leaks. This can be done using the 'disconnect' event in Socket.IO. Handle this event and perform the necessary cleanup by removing the disconnected client's socket ID from the saved connections.

In addition to the steps mentioned above, developers can also leverage the features provided by Socket.IO, such as rooms and namespaces, to create more complex and advanced real-time applications.

Rooms allow for grouping clients and sending messages to specific groups, while namespaces provide a way to separate different parts of the application and handle communication within those parts separately.

Socket.IO also supports various transport mechanisms, such as WebSocket, HTTP long-polling, and AJAX, ensuring that the application can adapt to different network conditions and provide a seamless real-time experience for users.

# Architectural Considerations

## Horizontal vs. vertical scaling

When an existing system fails to handle increasing workloads, possibly the most common and effective scaling options for any cloud application are horizontal and vertical scaling.

Horizontal scaling refers to adding additional servers or nodes to your infrastructure to support growing demands. Meanwhile, vertical scaling adds new resources to the existing system to manage the increasing workload.

You can use both approaches to scale your Socket.io application, but if you're looking to scale indefinitely, vertical scaling may limit how much you can grow.

Horizontal scaling may be the better option for scaling Socket.io, especially if you want to future-proof your application and reduce the chances of downtime. However, horizontal scaling also introduces some technical complexity. Servers must share the burden evenly to avoid any latency. This is where load balancers and reverse proxies come in.

## Load balancers and reverse proxies

Both reverse proxy and load balancers are intermediaries in a client-server architecture, working to make data exchange more efficient.

### Load balancer

A load balancer distributes incoming messages among a group of servers while returning the response from the selected server to the appropriate client. It tries to get the most out of each server's capacity. It offers benefits like:

- Preventing server overload to ensure consistent performance
- Ensuring quick responses to provide a great user experience

### Reverse proxy

Reverse proxies accept requests from a client using edge devices and forward them to the correct server. They offer benefits like:

- Preventing malicious attacks targeting server information
- Giving you the freedom to configure backend infrastructure

## Difference between load balancers and reverse proxies

Although, at first glance, they perform similar functions, load balancers and reverse proxies differ in a very important aspect. Load balancers are commonly deployed in instances involving multiple servers, while reverse proxies work best with just one web or application server. This difference also defines how Socket.io applications use them.

**LOAD BALANCERS ARE USED WITH WEBSOCKETS BECAUSE:**

They offer better fault tolerance and reliability.

They ensure even distribution of server load.

They minimize latency and maximize throughput.

**REVERSE PROXIES ARE IDEAL FOR HTTP LONG-POLLING BECAUSE:**

Great fallback to load balancers.

They work very well with APIs.

They can maintain connections to a single server.

## Best practices for implementing load balancing and reverse proxies

- When implementing load balancers, choose the right algorithm for your use case when distributing requests across servers. This ensures minimal latency and bandwidth costs during data exchange.
- Remember to set proper expiration dates for your reverse proxy caches so your application doesn't use outdated cache data to deliver real-time experiences.
- Configure security requirements for load balancers and reverse proxies to ensure your Socket.io application is safe from external attacks.

**A quick note for those following along**

It should be noted that the code we shared earlier in this white paper designs a single, non-durable system, and therefore, will not support load balancing without additional considerations.

One solution would be to implement a Sharding strategy that uses a Shard Key to allow connectivity to users who need to exchange information. This is required for a production scale system which enables high availability for a real-time system.

Another solution would be to simply use PubNub. Instead of needing to create a consistent, key-based load-balancing deployment, PubNub is built in a way that is designed for high concurrency and availability.

## Microservices

Microservices are smaller services that can be combined to create a more extensive application. These smaller services make it easier to scale as you only need to work on the microservice that needs updates while others continue to function independently.

However, microservices bring plenty of development and workflow complexities:

- Many individual services make it difficult to trace bugs or errors.
- Establishing proper communication channels and processes to streamline workflows is vital for faster development. Microservices require multiple teams with varying expertise.
- Group testing microservices is difficult because they usually don't use the same programming language. Ensure you have the right testing tools available.

## Best practices for building scalable Socket.io applications

To get the best out of your Socket.io applications, make sure to manage the following:

### Connection optimization

Whenever you send or receive a message from the backend, you'll expend resources to open, maintain, and close connections to the database. Your application's performance may deteriorate because of the large overhead.

Connection pooling uses a reusable database connection cache to store data. Using cache data, applications can scale efficiently to manage increases in server requests instead of opening and closing connections every time.

Connection pooling consists of two types of connections:

- Active connections that the application is using
- Idle connections that the application has available for use

When a new request comes in, the pool manager will look for any idle connections to handle the request. If all connections are active, it will add a new connection to the pool. If the pool is at its maximum capacity, the new requests will be queued until a connection becomes available.

### State management

An application's "state" is how it tracks and handles system information at any time. Ensuring an application functions correctly and provides a seamless user experience is essential.

Poor state management can hinder application scaling because data may be scattered or not properly synchronized. This means introducing new updates or features will become more difficult.

Using a stateless architecture can circumvent this issue. That's because the state of older transactions isn't stored on the server side or referenced in subsequent transactions. Instead, this architecture stores data on the client side while sending "reminders" to the server about previous steps and additional information.

Designing and implementing stateless architecture is relatively straightforward. Still, a few things to remember to ensure it functions optimally:

- If the workload increases exponentially, share the load evenly among the servers. You can do this using a load balancer.
- Session-related bugs are hard to fix, as cookies and other session data are stored on the server. Try to avoid sessions as much as possible.

### Scalable databases

Every application backend is connected to a database to manage user data. But your database has limits, and as your application becomes more popular, ensure your database can also handle the increased workload effectively and cost-effectively.

Consider opting for database applications with automated scaling capabilities like MongoDB and Cassandra to scale effortlessly. They can automatically adjust their capacity to handle sudden workload spikes, making them very useful in large-scale applications.

## Handling real-time traffic spikes

Traffic spikes are increases in workload significantly more than the application's average. You may experience spikes in various forms, including:

- Database load spikes
- Concurrent user increases
- Database storage limitations
- Multiple HTTP requests
- Cache overload

Though you can't avoid traffic spikes, you can prepare your application to handle them.

### STEP 1: Define your capacity

Understanding your application's load limits ensures you always deliver the best user experience. You can measure your load limit by establishing two baselines:

- The maximum load your application can realistically handle without compromising performance.
- The average load your system can handle under normal conditions.

Methods like load testing and analyzing historical data can help you establish the application's capacity by measuring key performance indicators (KPIs) like response time, throughput, error rate, and availability.

## STEP 2: Forecast your demand

Estimating how much load your system will have to manage over time will help you plan your scaling strategy and prepare for any unexpected scenarios. Using data on seasonal trends, company plans, and user behavior, you can create forecasting models and scenario planning to predict future application workloads.

## STEP 3: Plan your resources

Once you've established your baseline and predicted workloads, the next step is to assess if your hardware, software, and supporting network components can handle the workload increase. You can allocate resources effectively using capacity planning models and simulations. They will also help predict future costs, performance issues, and the security risks associated with scaling your application.

## STEP 4: Monitor your system

Despite in-depth planning and flawless execution, there is always a chance of problems when dealing with unregulated traffic spikes. Continuous monitoring is the best way to detect any anomalies affecting your system. You can monitor your system performance by analyzing system logs, setting alerts, and creating dashboards to identify and troubleshoot any unexpected behavior.

## Real-world use cases: How SaaS companies use PubNub with Socket.io connections to prevent account sharing

Subscription-based SaaS companies, like Netflix and Amazon Prime, regularly deal with the problem of account sharing, where multiple users use the same account. To tackle this, SaaS companies look for a security pattern that allows only one connection per authenticated user while using a data stream network like SocketIO for NodeJS.

PubNub's Data Stream Network uses one of two methods to prevent multiple logins on one account:

- Force Remote WebSocket Logout: To revoke previous login permissions when a new user logs in.
- Block Access While WebSocket Subscription in Use: To prevent new logins when a user is already logged in.

PubNub has the core services to build a reliable geo-redundant signaling solution for WebSockets, XMPP, BOSH, Comet, HTTP long-polling, etc. It is compatible with most programming languages, including Android Java, iOS Objective-C, and JavaScript Web/Mobile.

Check out our complete guide here.

## Monitoring and optimizing performance in scalable real-time applications

Performance and network security are essential for delivering phenomenal user experiences. Real-time applications must regularly track the application's health using real-time monitoring tools, which track and record data on KPIs like:

- Bandwidth usage
- Latency
- Error states
- Server response time
- CPU usage
- Network traffic volume

These tools use devices like sensors, probes, monitoring servers, and cloud-based monitoring platforms to track these metrics effectively. The aggregated data is usually presented in customizable analytics dashboards for easy comprehension and troubleshooting. You can also integrate with existing diagnostics and incident management systems for a more holistic understanding.

Every metric being tracked has a predefined threshold that, if exceeded, will trigger a warning or alert in the system. This warning signal lets you know of imminent failure so you can take preventive measures before your system fails.

Development teams should continuously monitor the effectiveness and efficiency of their real-time monitoring strategy, regularly reviewing monitoring parameters, alerting thresholds, and responses.

## Security considerations in scalable real-time applications

Sharing data in real time between two or more devices opens up a can of security threats. The more aware development teams and organizations are of these threats, the better their chances of avoiding security incidents.

Here are a few common security threats for real-time applications:

### Cross-site scripting attacks

Cross-site scripting (XSS) attacks hack into privileged accounts by forging cookies to imitate valid users. Once logged in, malicious actors can use the accounts to alter content and perform other actions. There are three types of XSS attacks:

- Reflected attacks that use dummy links and sites to send malicious scripts to the victim's browser
- Document Object Model-based attacks that inject malicious payloads into a webpage by manipulating the client's browser
- Stored XSS attacks that use unsanitized user inputs to target scripts permanently stored on target servers

### Buffer overflow attacks

In development, a buffer is a region that temporarily holds data being moved from one place to another. A buffer overflow attack exploits vulnerabilities when more data is added than it can hold, allowing attackers to crash, control, or modify the system. In-house applications are the most vulnerable to buffer overflows. Most commercial applications have identified and released patches to mitigate buffer vulnerabilities.

### Broken access control vulnerabilities

Access control is a security protocol that decides which users can access certain resources in a system. Without efficient access control, malicious users can masquerade as valid users to infiltrate the system. You can protect your system by introducing principles of least privilege and role-based access control. These limit user access rights to the bare minimum necessary for job function.

## Best practices for securing your Socket.io application at scale

Socket.io is vulnerable to external attacks. To get the most out of Socket.io, take steps to avoid common pitfalls and errors.

### Use Socket.io middlewares

Socket.io offers middlewares that can perform various functions to protect your data from malicious attacks. Middleware functions can be used for:

- Logging
- Authentication/authorization
- Rate limiting

### Avoid tunneling

Socket.io uses tunneling to route HTTP/HTTPS requests from a public server to your local server. However, using public networks with Socket.io opens up your application to malicious attacks. Avoid public networks whenever possible. Instead, use other secure and verified protocols.

### Use SSL/TLS encryption

Secure Sockets Layer (SSL) is a standard protocol to create secure connections between two devices or applications in a network. It prevents hackers from stealing personal and financial data. Transport Layer Security (TLS) is a more advanced version of SSL. It supports encrypted communication channels that address SSL vulnerabilities. TLS allows for more efficient authentication, faster handshakes, additional support messages, and more advanced message authentication.

## How Socket.io and PubNub are evolving to meet future scalability needs

Socket.io alone is great for establishing real-time data streaming connections for Node.js. But when used with PubNub, Socket.io can enhance connections using other real-time features like:

**Data streaming**
Achieve publication and subscription to real-time data streams in less than one-tenth of a second.

**Presence**
Determine who is present by actively monitoring and detecting the connection status of users and devices.

**Storage and playback**
Effectively store, retrieve, and replay messages in realtime in the order that they occurred.

**Analytics**
Visualize and oversee real-time traffic and usage for comprehensive insights.

**Cross platform**
Serialize and deserialize intricate objects automatically, even when working across diverse programming languages and platforms.

**Security and access management**
Enhance security with AES data encryption and implement a robust grant/revoke framework to ensure only authorized users can subscribe to real-time data streams.

## Case studies: Augmenting Socket.io applications with PubNub

### Case Study 1: How IntelliScape.io uses PubNub functions to power its global IoR network

IntelliScape.io is a platform offering Internet of Recognition (IoR) solutions. Initially, they relied on Google's Firebase and socket development tools like Socket.io for data integration. However, building real-time data stream networks to scale their systems was extremely complex and expensive.

Eventually, the team decided to go with PubNub, as it offered 50+ software development kits (SDK) provisioning access keys, channels, and functions to transmit IoR analytics events to customer dashboards in realtime.

With PubNub, IntelliScape.io significantly accelerated development cycles, often achieving production-ready solutions in less than a week. PubNub also offered other clear advantages:

1. Integrated within server nodes to enable real-time voice data conveyance with minimal latency
2. Powered real-time responses to voice requests for IntelliScape.io's voice recognition service, Galaxy
3. Integrated data seamlessly with enterprise platforms like Microsoft Azure Hub and IBM's Bluemix

PubNub allowed IntelliScape.io to stay laser-focused on its customers' needs by maintaining its real-time infrastructure, making it an invaluable component of its IoR platform.

### Case Study 2: Disprz uses PubNub to empower a more knowledgeable workforce

Disprz is an employee development and engagement platform with a mission to enhance sales and customer service through personalized and gamified knowledge delivery.

Disprz initially built the platform in-house using Socket.io and other open-source components but soon faced scalability and performance challenges as they introduced new features like group chat rooms, send messages, and a collaborative whiteboard.

PubNub allowed Disprz to deliver these real-time services on top of Socket.io protocols. The company transitioned from the in-house Socket.io stack to PubNub services seamlessly, thanks to PubNub's excellent documentation and responsive customer support.

Since adopting PubNub, Disprz has achieved its scalability goals without compromising performance, even as its user base grows by 30% quarter over quarter.

Looking ahead, Disprz is poised for international expansion, targeting the U.S. market while continuing to grow its real-time user base in India. PubNub's global redundancy, featuring 15 points of presence worldwide, positions PubNub as a crucial partner to support Disprz's ambitious growth plans.

## The future of scalable real-time applications

Users expect real-time interactions, and developers are rushing to meet this demand, driving innovation in the real-time application space. Keeping up with the latest trends is the best way to provide users with the best user experience.

### 1. Case-specific databases

Previously, all data, regardless of type or use case, was stored in a single data store. This was inefficient and dev teams moved to customized database applications that enable specific use cases. This approach turned out to be more effective, flexible, and future-proof. Moreover, it complied with data governance and compliance standards.

### 2. Low- or no-code platforms

Low-code or no-code platforms are the latest trend in application development. They enable even non-technical users to develop applications without any prior development experience. They also allow organizations to create and deploy useful applications quickly and easily. However, you may need to compromise on flexibility since you're limited by the capabilities of the application itself.

### 3. Enforcement of data governance policies

With easier access to data, governments and consumers are concerned with how organizations use and safeguard data. Ensuring compliance with data governance policies is critical for organizations to continue their data streaming operations.

## Plug into Socket.io and scale your real-time applications

The importance of scalability in real-time applications cannot be overstated. It allows applications to adapt to changing workloads while delivering high-quality user experiences cost-effectively. However, scalable real-time applications introduce development and workflow complexities. Dev teams will find overcoming these complexities worthwhile, considering scalability's numerous performance and user experience benefits.

PubNub's integration with Socket.io enhances real-time concurrent connections by providing features like rapid data streaming, presence monitoring, storage and playback, analytics, cross-platform compatibility, and robust security and access management.

Reach out to us to learn more about how PubNub can take your Socket.io real-time application to the next level. Or start your free trial to see the magic yourself.