JURNAL EMACS
[Engineering, MAthematics and Computer Science] Vol.7 No.1 January 2025: 109-120
e-ISSN: 2686-2573
DOI: 10.21512/emacsjournal.v7i1.12770

A Horizontally Scalable WebSocket Architecture for Cost-Effective Online Examination Proctoring System on AWS Cloud Infrastructure

Eko Cahyo Nugroho
Computer Science Department, School of Computer Science,
Bina Nusantara University,
Jakarta, Indonesia 11480
eko.nugroho003@binus.ac.id

Correspondence: eko.nugroho003@binus.ac.id

Abstract *- In this research work we present the cost-effective implementation of a WebSocket server with a horizontal scaling feature on AWS Cloud Service. Sokrates System, a SaaS provider for schools in Indonesia, faces challenges with AWS API Gateway for establishing WebSocket connections as it proves relatively expensive for their client schools. This research focuses on reducing infrastructure costs while maintaining technology quality. The solution presented in this study proposes an on-premise WebSocket server deployed at AWS EC2 instances. The server utilizes Node. js's cluster module to make the most out of the CPU's cores and has also implemented a Redis pub/sub mechanism to easily horizontal scale it to many EC2 instances. The system architecture utilizes DynamoDB to store students' proctoring status recorded on the first attempt at the quiz. Then, the real status update is delivered by WebSocket message. Testing results show the system can handle 10,000 concurrent users with just 12 t3a.small instances, achieving an average latency of 45ms and 95th percentile latency of 92ms. Compared to using API Gateway as the WebSocket server, this solution achieves a 22.1% reduction in monthly infrastructure costs. In the production environment, it demonstrates effective real-time monitoring capabilities for online examinations, including student activity tracking, automated disconnection detection, and proctor-student interaction features, providing schools with a reliable and scalable proctoring solution.*

Keywords: Server Management; LMS Proctoring; Cloud Computing; Message Broker; Horizontal Scaling

I. INTRODUCTION

The evolution of e-learning technology has experienced significant acceleration, particularly since the COVID-19 pandemic, fundamentally transforming how educational institutions deliver learning and assessment. One of the critical challenges in online learning is maintaining exam security through proper proctoring systems. Cutting-edge Learning Management Systems (LMS) involve the implementation of proctoring methods that are able to not only track student activity in real-time, but they can also detect any suspicious acts and guarantee truthful behavior at online examinations.

Proctoring systems have evolved beyond simple video monitoring to incorporate biometric and multimodal technologies, such as face, voice, and fingerprint recognition. These advancements are designed to authenticate users more reliably and ensure the security of online examinations (Han et al., 2024). Also, the integration of proctoring systems with LMS platforms facilitates seamless long-distance assessment processes while maintaining robust examination. However, these developments are not without any challenges, including ethical and data privacy issues. Proctoring systems must focus on these concerns by ensuring

transparency, respecting student privacy, and complying with data protection.

The normal way of using real-time proctoring systems on cloud platforms like AWS usually depends on WebSocket connections via API Gateway.

Figure 1. API Gateway Architecture (Sarat Dyuthi et al., 2024)

The WebSocket setup in AWS API Gateway uses a serverless architecture. Each time there is a connection, message, or disconnection, it triggers a Lambda function. When a client starts a WebSocket connection, API Gateway generates a unique connection ID and keeps track of the connection state. Each message received via this connection calls a Lambda function, which handles the message and may send replies to other connected clients using the API Gateway's management API. This design offers good scalability and requires little infrastructure management, but it runs on a pay-per-use basis, which can result in high costs.

This method brings high costs for schools. As per AWS pricing info in the official site (Amazon Web Services, n.d., WebSocket APIs section), in the Asia Pacific (Singapore) area, the WebSocket API costs $1.15 for every million messages for the first billion messages and $0.95 for every million messages after that. There's also a fee of $0.288 for each million connection minutes. AWS does offer a free tier that gives one million messages and 750,000 connection minutes for a year, but that is not enough for large projects. For example, a school with 1,000 students taking a 3-hour test, sending updates every 5 seconds, would create around 2.16 million messages and 180,000 connection minutes in one session. This leads to high operating costs, which can be a financial burden for many schools.

These financial issues make it important to look for new ways to design proctoring systems. This study tackles these issues by suggesting a different setup that uses a WebSocket server with smart scaling deployed on AWS EC2. It uses Node.js clustering for better resource use and Redis pub/sub to relay WebSocket messages to specific clients. This method not only reduces the costs but also provides the stability and scalability needed for large online exam proctoring.
This research can help deal with technical and cost issues that schools face when using online proctoring systems. Using AWS cloud services well and keeping costs low allows schools to keep exams secure and scalable without high infrastructure costs. The suggested proctoring system supports the main goals of making online learning safe, scalable, and focused on students.

WebSocket technology has become important in many areas, especially in real-time applications like proctoring systems. This part looks at studies that show the benefits and uses of WebSocket for improving real-time communication and data sharing.

A key study related to websocket and API Gateway implementation (Sarat Dyuthi et al., 2024) examines how to set up real-time event processing using API Gateway with WebSocket APIs. The study focuses on the two-way communication feature of WebSocket, allowing quick data transfer between clients and servers. This is very useful for proctoring systems that need real-time oversight and interaction between exam supervisors and exam participants. The authors talk about how to manage connection states, deal with errors, and ensure safe access, which are all important for keeping proctoring environments secure.

Also, another study related to optimize WebSocket (Maulana et al., 2019) looks at improving data transfer speeds in real-time chat apps with WebSocket technology.

```
websocket = new
WebSocket("ws://localhost:9000/serve
r.php");
websocket.onopen = function(evt) {
/* Perintah */ };
websocket.onclose = function(evt) {
/* Perintah */ };
websocket.onmessage = function(evt)
{ /* Perintah */ };
websocket.onerror = function(evt) {
/* Perintah */ };
websocket.send(message);
websocket.close();
```

Figure 2. Websocket Connection Handler (Maulana et al., 2019)

The results show that WebSocket cuts down latency much better than traditional AJAX methods. Data reveals that WebSocket uses only 9.63% of data, while AJAX can use up to 90.37%. This makes WebSocket a better option for applications needing fast and responsive communication. This efficiency is crucial in a proctoring system, where quick notifications can greatly affect the exam process. In these settings, any delays in communication can cause confusion or even security problems, risking the assessment's integrity.

WebSocket keeps a connection open, which allows real-time data sharing, letting examiners watch students all the time without needing to send requests repeatedly. This constant connection improves the system's responsiveness and cuts down the work involved in starting new connections for each interaction. Thus, examiners can quickly send alerts or instructions to students, making sure any problems are dealt with rapidly.

Moreover, the study points out that WebSocket can improve user experience by reducing delays and allowing a smooth flow of information. For example, during an online exam, if a student has a technical problem or wants to ask a question, they can quickly talk to the proctor through a chat system using WebSocket. This fast feedback helps create a supportive atmosphere that can ease students' worries and make the exam process smoother.

WebSocket technology not only makes communication speed better but also improves interactions between students and proctors. It allows for features like live video and real-time screen sharing in proctoring systems, which enhances how monitoring is done. By using WebSocket, developers can create proctoring solutions that focus on both security and user engagement during exams.

The basic concept of WebSocket technology and its real-time functions was explored by Pimentel and Nickerson. They showed that WebSocket supports efficient two-way communication, which is vital for transmitting data in real-time.

In proctoring systems, Han et al. conducted a thorough review of literature that showed digital proctoring is increasingly important in higher education and highlighted the need for dependable real-time monitoring. This idea is also backed by Nurpeisova et al.'s study on creating proctoring systems for online tests, which stressed the importance of real-time communication for upholding exam integrity.

Skvorc et al. looked into how well WebSocket works, checking the WebSocket protocol for web streams that operate in both directions. They shared important information about how well it works and its dependability in real-time uses. Alexeev et al. also provided key information on smart load balancing methods for WebSocket services that can grow, focusing especially on how these principles apply in environments that use distributed computing.

In the other research related to this work, proposed an efficient resource management strategy for federated cloud environments in Infrastructure as a Service (IaaS) delivery (Samha et al., 2024). To be specific, the architecture is introduced a Trust Manager (TM) that helps to evaluate the Service Level Agreement (SLA) violations between the cloud users and the service providers. The arrangement uses a Broker Manager (BM) as an intermediary that helps to the resource allocation, and service negotiation between the providers and the users.

By means of virtualization technology, the architecture is established to cater to ensuring that Virtual Machines (VMs) operate independently in a multi-tenant environment by

implementing the Banker's algorithm for resource allocation and deadlock prevention. In fact, a plus of the system is the integration of a deep Q-based algorithm for service provider ranking and selection, which enables effective resource utilization and service delivery.

The results of their experiment were better than the traditional cloud models output, adding extra information such as better average reaction time and reduced SLA exceptions. Besides, the message handling system based on brokers ensured better utilization of throughput and service request handling of cloud, and thus it just becomes the area of focus where robust message broking capabilities of scalable web applications are required.

With its valuable insights, this research brings significant benefits in terms of cloud management and message broking for online examination systems, which demand a highly efficient allocation of resources through a message delivery system that can be trusted.

In the other research related to implement message queue system, were figuring out how to optimize high-performance distributed real-time stream processing systems, which especially in this case include the improvement of message queue performance with Kafka (Jiang et al., 2019). Here's how they found out that Kafka, as a distributed messaging system, has a number of specific benefits in both horizontal scalability and throughputs, which ultimately makes it the best choice in situations where message producers and consumers operate at different speeds.

In the Jiang's research found that Kafka's performance is a key factor in the overall functioning of stream processing systems, especialy in use cases demanding the rapid data reception from producers, as well as quick delivery to consumers. Their architecture borrows Kafka's ability to support low-latency message distribution while also covering both real-time and offline message processing.

One important part of their work was using Kafka cat technology, which offered a way to handle messages without JVM. This change led to less resource use and better performance than usual message systems. Their tests showed that this change made processing quicker and cut down on network transmission costs in the message queue setup.

The research also pointed out how memory-based file systems can help improve message queue efficiency. By using memory for storing data in Kafka, they showed clear gains in reading and writing data quickly compared to using disks. This helped fix key I/O issues in high-speed messaging situations. This technique worked especially well in situations where quick message processing and delivery were needed, achieving read/write speeds up to six times faster than regular disk systems..

This research looked at Kafka use, but the main ideas and ways to improve performance apply to Redis too. Redis has in-memory storage and pub/sub functions that fit well for using similar message broking systems in online exam proctoring. This provides similar advantages, like fast response times and high performance.

Recent uses of WebSocket in different areas have shown good results. Soewito et al. showed how well WebSocket works for real-time applications (Soewito et al., 2019), and Sarat Dyuthi's research on setting up real-time event processing with AWS and WebSocket APIs gives useful information about cloud-based uses (Sarat Dyuthi et al., 2019).

This collection of research gives a solid base for making a WebSocket server system that can grow horizontally using Redis as a message broker in AWS for proctoring systems. The studies show that both the theory and real-world uses of the important technologies work well, while also pointing out possible problems and ways to address them in similar situations.

II. METHODS

This study uses a complete way to create and assess a WebSocket server for online exam monitoring. The method includes designing the system, developing it, and testing it to guarantee a strong and budget-friendly solution. The

system layout focuses mainly on being scalable and reliable.

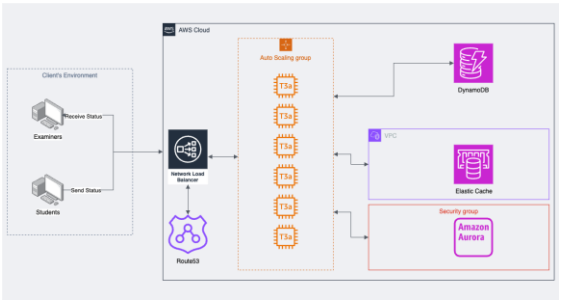

Figure 3. Designed Websocket Server Architecture

The main setup has several EC2 instances that run WebSocket servers based on Node.js when AutoScaling is Horizontally activated. However, it only has 1 EC2 instance by default when it is in low usage. These instances connect using a Redis pub/sub system to send messages in real time. Each instance uses the Node.js cluster package to make better use of CPU cores with a multithreading approach, allowing for good management of multiple connections at once. The overall system is on AWS cloud infrastructure, and the client setup includes two main types of users: examiners (proctors) and students. These users connect through WebSocket via AWS Route 53, which handles DNS routing and has high availability due to its failover functions. Incoming connections are managed by a Network Load Balancer, which helps distribute traffic evenly across several EC2 instances in an Auto Scaling group.

In the system, several t3a EC2 instances run WebSocket servers using Node.js programmed codes. These instances are part of an Auto Scaling group that changes the number of active servers based on real-time data like CPU usage, network speed, and connection counts. This ability to scale ensures that resources are used efficiently while keeping performance steady during different testing loads.

The data management part uses various AWS services together. DynamoDB acts as the main database for storing connection states and exam session information, providing stable performance even at scale. Redis, through AWS ElastiCache, helps with real-time message sharing between server instances using its pub/sub feature. A major issue in a multi-instance WebSocket server setup is the challenge of communication between instances. If a student on Instance A sends their exam status, the system must make sure that the proctor on Instance B can get those updates. Without a proper message-sharing system, messages would stay within their instances, causing communication gaps that disrupt real-time monitoring.

To fix this issue, the system uses Redis pub/sub as a way to send messages between different server instances. When a WebSocket server instance gets a message from a student and does not find the matching proctor in its list of connections, it sends the message to a Redis channel. All WebSocket server instances join this channel to get messages from other instances. When a message is received via Redis, each instance looks to see if the target proctor is connected and, if they are, sends the message through the right WebSocket connection id. For keeping examination data and user info safe, Amazon RDS Aurora offers strong relational database features in a specific security group.

```
{
    type: 'message_to_master',
    clientConnectionSocketId: 'unique_socket_id',
    message: {
        action: 'lmsQuizMonitoringStudent',
        data: {
            quiz_id: 'string',
            quiz_attempt_id: 'string',
            student_id: 'string',
            user_status: 'active|away|offline',
            question_answered: number,
            question_total: number,
            question_number_current: number,
            quiz_participant_id: 'string',
            quiz_result_id: 'string',
            score: number,
            student_name: 'string',
            time_end: 'timestamp',
            time_start: 'timestamp',
            topic_uuid: 'string'
        }
    }
}
```

Figure 4. Basic WebSocket Message Structure

The student status update message, as shown in Figure 4 is key to the communication system in proctoring. Each message has important data about the exam session, allowing real-time tracking of student actions. The 'type' field, labeled 'message_to_master,' means this message needs to go to the master node for routing between instances since the code implements a multithreading in a single instance. The 'clientConnectionSocketId' is a

unique ID for the WebSocket connection, crucial for routing messages and managing connection states. The message details, found in the 'message' object, include information on the exam progress. The 'action' field, set to 'lmsQuizMonitoringStudent,' activates specific routines for monitoring students.

This message format helps to track student progress, activity, and exam timing. The 'user_status' field is important for proctoring, as it allows for quick identification of suspicious actions like switching tabs or exiting the exam window. When status updates come in, the system can instantly alert proctors, no matter which server they are on, due to the Redis pub/sub system.

```
// Redis subscription setup on each instance
redisSubs.subscribe(channelName);

redisSubs.on('message', (channel, message) => {
    if (channel === channelName) {
        const parsedMessage = CircularJSON.parse(message);
        if (parsedMessage.message.sender !== clientId) {
            // Process and forward message to connected proctors
            // if they are found in this instance
        }
    }
});
```

Figure 5. Redis pub/sub prorgam codes

The Redis subscription system is important for communication between different instances, as seen in Figure 5. Each WebSocket server instance connects to a shared Redis channel, forming a network for broadcasting messages. The code example shows how this subscription works, with 'redisSubs.subscribe(channelName)' setting up the subscription to a shared channel that all instances use.

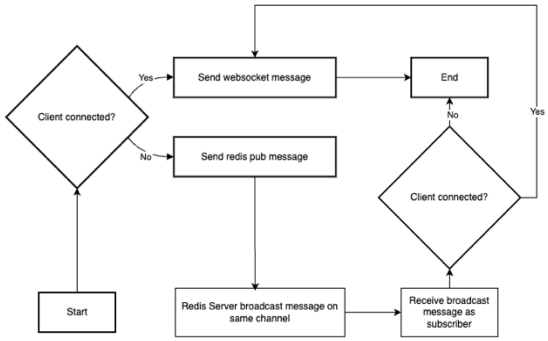

Figure 6. Websocket to Redis Flowchart

Figure 6 shows the message routing in the WebSocket server setup for cross-instance communication. When sending a message to a client, the system checks if the client is connected to the current instance. If the client is in the instance's connection list, the message is sent directly through the open WebSocket connection, ensuring quick delivery in real time.

But if the target client is not linked to the current instance, the system starts the Redis pub/sub mechanism. The message gets sent to a specific Redis channel, allowing it to spread to all WebSocket server instances when horizontal autoscaling is activated and triggered to create more than one EC2 instance. Each instance, as a subscriber to the Redis channel, gets the broadcast message and looks in its local connection pool for the target client. If an instance sees the target client in its connections, it sends the message through the right WebSocket connection ID. This routing system makes sure the message is delivered no matter which instance the target client is linked to.

The flow shows that the system can keep smooth communication in real time between different server instances. This method is very important for exam proctoring, where updates on student status need to get to proctors quickly, no matter how the instances are spread out. This setup builds a messaging network that maintains the real-time features of WebSocket communication and allows for growth with several server instances triggered.

The security part is implemented in the architecture. The system works in a Virtual Private Cloud (VPC) for network safety and isolation. Security groups manage the traffic that goes in and out of EC2 instances and make sure that instances can only communicate with Elastic Load Balancer. Also, SSL/TLS encryption protects all WebSocket connections with no default port number set. This layered approach to security works to keep data safe and ensure communication stays intact during the exam process.

The design of the architecture focuses on performance, security, and saving costs. By using auto-scaling and smart resource use, the

system can manage many connections at once while keeping costs low. This method offers a clear edge over old API Gateway systems, especially for organizations that run many exam sessions at the same time with huge traffic.

The research uses a comprehensive evaluation method to check how well the new WebSocket server works. The evaluation looks at three main things: performance, scalability, and cost. The performance testing focuses on checking WebSocket time responses to see how steady and reliable real-time communications are. This involves running test clients that mimic student exam activities by sending status updates every 5 seconds. The system logs the exact times when each message is sent and received, which helps calculate message delivery time patterns. This method permits a thorough review of the system's real-time communication performance under different loads and server types.

To check how well the architecture can handle more users, the method uses controlled load tests that look at how the system scales up automatically. The test setup relies on AWS CloudWatch alarms set with certain CPU usage limits to initiate scaling actions. During the tests, the number of simultaneous connections is slowly raised, while keeping an eye on many performance indicators, such as how instances are created and ended, response times during scaling changes, and total resource use. This approach offers information on how well the system can keep performance steady while responding to different load changes.

The cost analysis method sets up a way to compare the economic efficiency of the suggested solution with the previous API Gateway setup. This includes keeping track of operational data, such as how often each student sends messages, total time connected, usage patterns, and amounts of data transferred. The approach involves gathering cost data from both systems, allowing for a clear comparison of costs for the same workload.

III. RESULTS AND DISCUSSION

The performance testing of the WebSocket server setup was done with Apache JMeter, simulating many students and proctors during different examination sessions. The test cases aimed to check how well messages were delivered and how reliable communication was between different instances.

Test Configuration:

- Test Duration: 30 minutes
- Concurrent Users: 1,000 simulated students
- Message Frequency: 1 message every 5 seconds per student
- Server Configuration: 2 t3a.small EC2 instances

Here is the result of the stress test as shown in Table 1 :

Table 1. Stress Test Result

Metric	Average	90th%	95th%	99th%
Message Latency (ms)	45	78	92	115
Round-trip Time (ms)	89	125	148	187
CPU Utilization (%)	65	72	78	82
Memory Usage (%)	58	67	73	79

WebSocket message latency and round-trip time were measured using Apache JMeter and the WebSocket Samplers plugin. To find message latency, the timestamp_sent from the testing WebSocket message was compared with the server receipt time. Round-trip time was calculated from the time the message was sent until the acknowledgment was received back by the test client. CPU and memory usage metrics were gathered via AWS CloudWatch Dashboard with detailed monitoring turned on as shown in Figure 7.

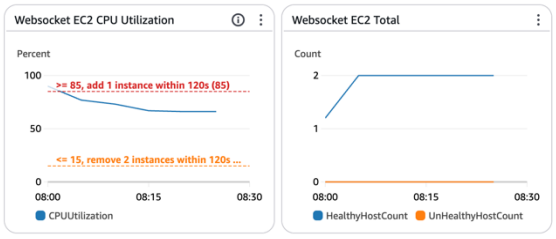

Figure 7. Cloudwatch Dashboard During Stress Test

The CloudWatch metrics were combined for all EC2 instances in the Auto Scaling group, giving total resource use data during the test time. The dashboard was set up to show real-time metrics and past data, allowing a review of resource use trends under different load situations.

The test results show steady performance in the distributed system. The average message delay is 45ms, which shows that message routing is working well, even when messages had to go through the Redis pub/sub system for delivery between instances. The 95th percentile latency of 92ms shows that most messages were sent in good time for proctoring exams.

After the first performance tests, the architecture was set up in a production setting for a real online exam with 10,000 students. This real-world use helped to understand how well the system works and how it can grow under real exam conditions.

Figure 8 shows that the system used resources well, needing only 12 (twelve) t3a.small instances for 10,000 users at once. This means around 833 WebSocket connections per instance, which is much better than earlier estimates. The Auto Scaling group kept resources evenly spread out, adding instances automatically as more students joined when the exam began based on the load at the time.

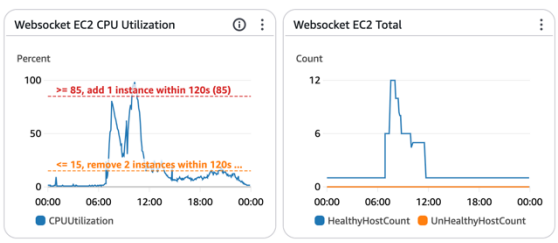

Figure 8. Real usage of Websocket EC2 instances

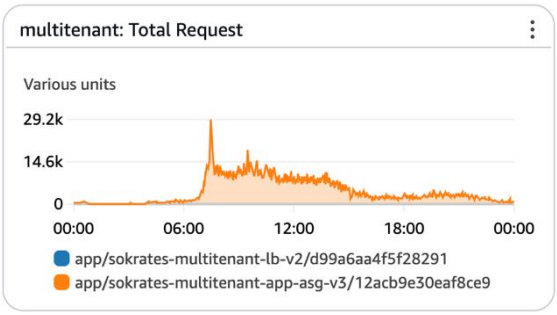

Figure 9. Incoming Request During Exam

Figure 9 shows the traffic pattern seen in the big test run. The graph displays the number of HTTP requests delivered to LMS system requests the system handled in a 24-hour time frame, with a notable spike during the exam times. This data also describes how many users are using the LMS during the exam.

The traffic pattern shows how the system handles sudden jumps in connection requests, like the spike graph increase at 06:00 when students started logging into the exam system. The setup managed this surge well, keeping performance stable during the busiest times. The graph indicates that the auto-scaling feature responded effectively to the traffic, with the system managing the initial spike smoothly and maintaining steady performance throughout the high-traffic times. The slow drop in traffic after 12:00 reflects the end of the exams, with the system reducing resources as the load lightened. This traffic pattern confirms the system's ability to deal with quick scaling needs and long periods of high load while using resources efficiently.

This real-world use showed that the architecture is effective, reliable, and scalable. It can manage 10,000 users at the same time and adaptively provide the EC2 t3a.small instances, which have the same scalability and reliability as serverless WebSocket API Gateway. The findings prove that this architecture can dependably handle large testing events while keeping resource use and costs low.

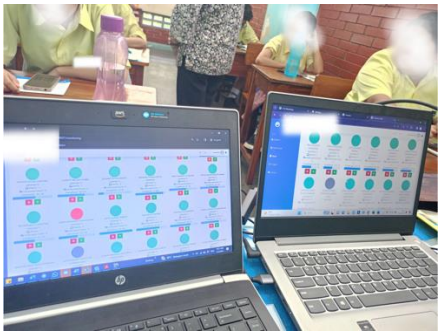

Figure 10. Monitoring Dashboard on the Examiner Side

Figure 10 shows the interface for monitoring used by exam proctors in the real situation in the exam room. It displays how the WebSocket setup works for giving quick updates on student status. The interface uses colors to quickly indicate student activities:

- Green Icon: Indicates an "active" status, showing that the student is currently open and stays in the examination interface
- Yellow Icon: Represents an "away" status, triggered when students switch browser tabs or minimize the examination window
- Red Icon: Signals an "offline" status, activated when the connection is lost or the examination page is closed

The monitoring dashboard provides comprehensive real-time information for each student:

- The current question number being attempted
- Total questions answered
- Time remaining in the examination
- Current score (upon submission)
- Count of any "away" status events

This real-time monitoring function works because of the WebSocket's good message routing system. Every time there is a status change, a WebSocket message is sent right away. This can happen through direct communication or via Redis, making sure proctors get updates quickly, usually within milliseconds of what students do. While running a big exam, the system handled about 2,000 status updates each second at peak times, and the average delivery delay stayed below 100ms.

The speed of the interface shows that the architecture can meet real-time monitoring needs properly, giving proctors instant awareness of student actions and possible exam integrity concerns. This setup shows how the WebSocket architecture is useful for keeping exams fair through thorough real-time monitoring.

Table 2. Cost Comparison 1 Month Period

	API Gateway Cost ($)	EC2 Instance Cost ($)	Total Cost Include All Used Services
Using Websocket API Gateway	$1,345.74	$402.75	$8,352.35
Using Horizontal Scalling EC2 Websocket Server	**$133.61**	$695.77	**$6,505.84**

Table 2 shows a cost comparison between the old API Gateway WebSocket setup and the new EC2-based WebSocket server design that can scale. The analysis looks at one month YoY of use in the Learning Management System (LMS) with the highest number of exam participants in that month.

Previously, the LMS used serverless API Gateway WebSocket API for monitoring exams in real time. This setup gave some serverless advantages but led to high expenses due to API Gateway's pricing structure that depended on how many messages were sent and how long connections lasted. Each month, the API Gateway costs roughly hit $1,345.74, and extra EC2 instances added $402.75 as the backend of the LMS system, making the total monthly infrastructure cost $8,352.35, which included other AWS services.

The move to the EC2-based WebSocket server setup that can scale horizontally showed clear cost savings. Although more EC2 instances were needed ($695.77) to manage the WebSocket connections directly, the expenses for the API Gateway dropped significantly to

$133.61 (mostly for other API endpoints). This resulted in a total monthly infrastructure cost of $6,505.84, which is a 22.1% reduction in total expenses.

Key cost implications:

- API Gateway costs reduced by 90.1% ($1,345.74 to $133.61)
- EC2 costs increased by 72.7% ($402.75 to $695.77)
- Net monthly savings: $1,846.51 (22.1% reduction)

This cost analysis shows that the new architecture is cost-efficient and keeps similar or better performance than the previous serverless WebSocket API Gateway. The big drop in API Gateway costs outweighs the higher EC2 costs, proving that the change in architecture boosts technical abilities and offers notable cost savings worth the trade-off.

IV. CONCLUSION

This study shows a successful setup of a WebSocket server that can scale horizontally for online test supervision in real time. It points out that the suggested setup handles technical issues of monitoring a lot of users at once and also considers the budget limits of schools.

The implementation results validate several key achievements:

- First, the architecture managed many connections at the same time, supporting up to 10,000 participants in an exam using just 12 (twelve) t3a.small EC2 instances. The message routing system based on Redis worked well to keep communication happening in real-time across different server instances. Message delivery delays stayed under 100ms, ensuring status updates for exam monitoring were quick.
- Second, the system showed good scalability traits, changing resources as needed while keeping performance steady. The design's capacity to manage quick increases in connections, proven by the traffic analysis showing highs of 29,200 simultaneous requests, indicates its dependability for large online exam events.
- Third, the cost analysis shows notable economic advantages, with the new design reducing monthly infrastructure costs by 22.1% when compared to the old serverless WebSocket API Gateway setup. This decrease in cost, accomplished while keeping or enhancing performance indicators, proves that schools or educational institutions can adopt strong real-time proctoring solutions without facing high operational costs.

The study helps the field by offering a practical and low-cost method for supervising online exams in real time on a large scale. The design effectively meets both technical needs and financial limits, which is useful for schools using online exam systems. Future studies can look better performance in term of data reliability and throughput by comparing between Redis and Kafka for distributing the websocket messages within instances since Kafka can provide more consistent data and higher throughput compared to Redis.

REFERENCES

Alexeev, V. A., Domashnev, P. V., Lavrukhina, T. V., & Nazarkin, O. A. (2019). The Design Principles of Intelligent Load Balancing for Scalable WebSocket Services Used with Grid Computing. *Procedia Computer Science*, *150*, 61–68. https://doi.org/10.1016/j.procs.2019.02.014

Alimudin, A., M, A. F., Sarinastiti, W., Yuwono, W., Winarno, I., Santoso, R., Murdaningtyas, C. D., Ilyas, M. I., & Muktasib, M. R. (2024). Implementation of Automatic Proctoring in Online Exam System. *2024 International Electronics Symposium (IES)*, 698–702. https://doi.org/10.1109/IES63037.2024.10665805

Arvindhan, M., & Anand, A. (2019). Scheming an Proficient Auto Scaling Technique for Minimizing Response Time in Load Balancing on Amazon AWS Cloud.

SSRN Electronic Journal. https://doi.org/10.2139/ssrn.3390801

Castaño, M., Noeller, C., & Sharma, R. (2021). Implementing remotely proctored testing in nursing education. *Teaching and Learning in Nursing*, *16*(2), 156–161. https://doi.org/10.1016/j.teln.2020.10.008

Eka Putra, F. P., Muslim, F., Hasanah, N., Holipah, Paradina, R., & Alim, R. (2024). Analisis Komparasi Protokol Websocket dan MQTT Dalam Proses Push Notification. *Jurnal Sistim Informasi Dan Teknologi*, 63–72. https://doi.org/10.60083/jsisfotek.v5i4.325

Friendly, Sembiring, A. P., Faza, S., Lukcyhasnita, A., & Destiadi, R. (2023). Design and Implementation of IOT Connection With Websocket Using PHP. *International Journal of Research in Vocational Studies (IJRVOCAS)*, *2*(4), 94–98. https://doi.org/10.53893/ijrvocas.v2i4.173

Han, S., Nikou, S., & Yilma Ayele, W. (2024). Digital proctoring in higher education: a systematic literature review. *International Journal of Educational Management*, *38*(1), 265–285. https://doi.org/10.1108/IJEM-12-2022-0522

Juansen, M., & Simatupang, S. (2023). Integrasi Mesin Absensi dan Pusher Notification pada Sistem Informasi Akademik Sekolah Untuk Monitoring Absensi Real-Time. *Journal of Computer System and Informatics (JoSYC)*, *4*(4), 1028–1035. https://doi.org/10.47065/josyc.v4i4.3840

Khoda Parast, F., Sindhav, C., Nikam, S., Izadi Yekta, H., Kent, K. B., & Hakak, S. (2022). Cloud computing security: A survey of service-based models. *Computers & Security*, *114*, 102580. https://doi.org/10.1016/j.cose.2021.102580

Lazidis, A., Tsakos, K., & Petrakis, E. G. M. (2022). Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. *Internet of Things*, *19*, 100538. https://doi.org/10.1016/j.iot.2022.100538

Maharjan, R., Chy, M. S. H., Arju, M. A., & Cerny, T. (2023). Benchmarking Message Queues. *Telecom*, *4*(2), 298–312. https://doi.org/10.3390/telecom4020018

Maulana, A. R., & Rahmatulloh, A. (2019). Websocket untuk Optimasi Kecepatan Data Transfer pada Real Time Chatting. *Innovation in Research of Informatics (INNOVATICS)*, *1*(1). https://doi.org/10.37058/innovatics.v1i1.667

Nguyen, X. H., Le-Pham, V. M., Than, T. T., & Nguyen, M. S. (2022). PROCTORING ONLINE EXAM USING IOT TECHNOLOGY. *2022 9th NAFOSTED Conference on Information and Computer Science (NICS)*, 7–12. https://doi.org/10.1109/NICS56915.2022.10013409

Nurpeisova, A., Shaushenova, A., Mutalova, Z., Ongarbayeva, M., Niyazbekova, S., Bekenova, A., Zhumaliyeva, L., & Zhumasseitova, S. (2023). Research on the Development of a Proctoring System for Conducting Online Exams in Kazakhstan. *Computation*, *11*(6), 120. https://doi.org/10.3390/computation11060120

Palumbo, F., Aceto, G., Botta, A., Ciuonzo, D., Persico, V., & Pescapé, A. (2021). Characterization and analysis of cloud-to-user latency: The case of Azure and AWS. *Computer Networks*, *184*, 107693. https://doi.org/10.1016/j.comnet.2020.107693

Samha, A. K. (2024). Strategies for efficient resource management in federated cloud environments supporting Infrastructure as a Service (IaaS). *Journal of Engineering Research*, *12*(2), 101–114. https://doi.org/10.1016/j.jer.2023.10.031

Sarat Dyuthi, K. S. (2024). Configuring Real-Time Event Processing of Api Gateway with Aws and Websocket Api's. *Journal of Informatics Education and Research*, *4*(3). https://doi.org/10.52783/jier.v4i3.1711

Schoenmakers, B., & Wens, J. (2021). Efficiency, Usability, and Outcomes of Proctored Next-Level Exams for

Proficiency Testing in Primary Care Education: Observational Study. *JMIR Formative Research*, *5*(8), e23834. https://doi.org/10.2196/23834

Smith, C. D., Atawala, N., Klatt, C. A., & Klatt, E. C. (2022). A review of web-based application of online learning in pathology and laboratory medicine. *Journal of Pathology Informatics*, *13*, 100132. https://doi.org/10.1016/j.jpi.2022.100132

Soewito, B., Christian, Gunawan, F. E., Diana, & Kusuma, I. G. P. (2019). Websocket to Support Real Time Smart Home Applications. *Procedia Computer Science*, *157*, 560–566. https://doi.org/10.1016/j.procs.2019.09.014

Tanaem, P. F., David Manuputty, A., & Wijaya, A. F. (2022). STARS: Websocket Design and Implementation. *2022 International Seminar on Application for Technology of Information and Communication (ISemantic)*, 167–171. https://doi.org/10.1109/iSemantic55962.2022.9920451

Taniar, D., Barthelemy, J., & Cheng, L. (2021). Research on Real-time Data Transmission between IoT Gateway and Cloud Platform based on Two-way Communication Technology. *International Journal of Smartcare Home*, *1*(1), 61–74. https://doi.org/10.21742/26531941.1.1.06

Wei Jiang, Liu-Gen Xu, Hai-Bo Hu, Yue Ma (2019). Improvement Design for Distributed Real-Time Stream Processing Systems. *Journal of Electronic Science and Technology*, 3-12. 10.11989/JEST.1674-862X.80904011

Amazon Web Services. (n.d.). Amazon API Gateway pricing | API management | Amazon Web Services. https://aws.amazon.com/api-gateway/pricing/