IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15314

RishiBuild – A Scalable Chat Application Using Docker and AWS ElastiCache

A. Manikandan[1], Mr. S. S. Saravana Kumar[2]

Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore, Tamil Nadu, India[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore, Tamil Nadu, India[2]

Abstract: Real-time communication has become a fundamental requirement in modern cloud-based applications. This project presents RishiBuild, a scalable chat application designed and implemented using WebSocket-based communication, Docker containerization, and Amazon Web Services (AWS) cloud infrastructure. The backend server, built with Node.js and Express.js, is deployed inside a Docker container hosted on an AWS EC2 instance. AWS ElastiCache (Redis) is integrated as a distributed in-memory caching layer, enabling fast message synchronization across multiple server instances through a Publish/Subscribe (Pub/Sub) mechanism. The system supports horizontal scalability, environment consistency, and secure cloud deployment through controlled AWS Security Group configuration. Testing results confirm that the application successfully handles concurrent WebSocket connections with low latency and stable performance. The project demonstrates practical implementation of modern DevOps practices, distributed caching, and cloud-native architecture suitable for real-world deployment scenarios.

Keywords: WebSocket, Docker, AWS ElastiCache, Redis, Node.js, Real-Time Communication, Cloud Deployment, Distributed Caching, Scalability.

I. INTRODUCTION

Real-time communication systems have become essential in modern digital applications. Messaging platforms, collaboration tools, and social networking applications depend heavily on instant communication mechanisms. Unlike traditional web applications that operate on a request-response model, chat systems require persistent connections and immediate data exchange.

Traditional monolithic chat architectures rely on HTTP polling mechanisms, which introduce significant latency, high network overhead, and poor scalability under increased user load. These limitations make conventional approaches unsuitable for applications requiring concurrent multi-user communication with low-latency message delivery.

This project proposes and implements RishiBuild, a scalable real-time chat application that addresses these limitations through the integration of WebSocket technology, containerization with Docker, and cloud deployment on Amazon Web Services (AWS). The system leverages AWS ElastiCache (Redis) as a distributed in-memory caching layer to synchronize messages across backend instances and support horizontal scalability.

The primary contributions of this project include: (1) implementation of persistent WebSocket communication using Node.js and Express.js; (2) containerized deployment using Docker for environment consistency and portability; (3) integration of Redis Pub/Sub for distributed message broadcasting; (4) cloud hosting on AWS EC2 with secure Security Group configuration; and (5) comprehensive functional and performance validation of the deployed system.

II. RELATED WORK

Several approaches to real-time communication have been studied and implemented in prior work. Early chat systems relied on HTTP polling, where clients repeatedly query the server at fixed intervals to check for new messages. While straightforward to implement, polling introduces unnecessary server load and network traffic, resulting in high latency and inefficient bandwidth utilization.

Long polling improved upon this model by keeping the HTTP connection open until the server had new data to return. However, this approach still suffers from connection overhead and does not eliminate the latency inherent in repeated

DOI: 10.17148/IJARCCE.2026.15314

request-response cycles. WebSocket technology was introduced as a full-duplex communication protocol that maintains a persistent connection between client and server, significantly reducing latency and eliminating redundant connections. Research on containerization demonstrates that Docker substantially improves deployment reliability and eliminates environment-related inconsistencies. By bundling the application alongside its runtime dependencies, Docker resolves the widely documented 'works on my machine' problem and ensures consistent behavior across development, testing, and production environments.

In-memory data stores such as Redis have been extensively studied for performance-critical applications. Redis supports sub-millisecond read/write latency, making it significantly faster than disk-based databases for real-time use cases. The Redis Publish/Subscribe mechanism has been adopted in distributed architectures to enable message broadcasting across multiple backend instances without requiring direct server-to-server communication.

Existing literature identifies a gap in practical, cloud-deployed implementations that combine WebSocket communication, Docker containerization, and Redis-based distributed caching in a unified architecture. This project addresses that gap by providing a fully functional, cloud-hosted reference implementation with comprehensive documentation and validated test results.

III. OBJECTIVES AND CHALLENGES

The primary objectives of this project are: (1) to design and implement a real-time chat application using WebSocket protocol for persistent bidirectional communication; (2) to containerize the application using Docker for consistent and portable deployment; (3) to deploy the system on AWS EC2 and integrate AWS ElastiCache (Redis) for distributed message synchronization; (4) to configure secure cloud networking using AWS Security Groups; and (5) to validate the system through functional, integration, deployment, and performance testing.

Development Challenges

The primary technical challenge encountered during implementation involved establishing a stable connection between the Node.js backend running inside a Docker container on AWS EC2 and the Redis endpoint provided by AWS ElastiCache. Security Group misconfiguration initially prevented the application container from reaching the Redis endpoint on port 6379. This was resolved by carefully defining inbound rules to allow traffic from the EC2 instance's security group to the ElastiCache subnet group.

A second challenge involved configuring port exposure within the Docker container to make the application accessible through the EC2 instance's public IP address. The Dockerfile required explicit port 3000 exposure, and the EC2 Security Group required a corresponding inbound rule to permit external traffic. These configurations were iteratively validated until consistent public accessibility was confirmed.

Managing environment consistency between local development and cloud deployment was addressed through Docker's dependency isolation model. The Dockerfile packages the Node.js runtime, application source code, and all required npm packages into a single container image, ensuring that the application behaves identically regardless of the underlying host environment.

IV. SYSTEM ARCHITECTURE

The proposed system architecture integrates WebSocket-based communication, Docker containerization, and AWS cloud infrastructure into a cohesive, scalable platform. Unlike traditional centralized client-server architectures, this design supports horizontal expansion by decoupling message synchronization from individual server instances through Redis Pub/Sub.

Users access the chat interface through a web browser, which establishes a persistent WebSocket connection with the backend Node.js server. The backend is packaged in a Docker container deployed on an AWS EC2 instance. When a user sends a message, the server publishes it to a Redis channel on AWS ElastiCache. All subscribed server instances receive the broadcast and forward the message to their connected clients, ensuring all users receive updates in real time.

The architecture adheres to a stateless backend design, meaning that individual server instances do not store session data locally. All shared state is managed through Redis, which enables the system to scale horizontally by adding additional EC2 instances and Docker containers without modifying the core application logic.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15314

Table 1. System Architecture Components

Layer	Technology	Role
Client Interface	HTML, CSS, JavaScript	Browser-based chat UI with WebSocket client
WebSocket Communication	ws (Node.js)	Persistent full-duplex client-server messaging
Backend Server	Node.js, Express.js	Connection management and message processing
Containerization	Docker, Dockerfile	Packages application with consistent runtime
Distributed Caching	Redis (AWS ElastiCache)	In-memory Pub/Sub message synchronization
Compute Infrastructure	AWS EC2 (Ubuntu 22.04)	Hosts Docker container with public IP access
Security	AWS Security Groups	Controlled inbound port access (22, 3000, 6379)

V. IMPLEMENTATION

The implementation is organized into five functional modules, each addressing a distinct layer of the chat application architecture. This modular structure ensures that individual components can be updated and tested independently without disrupting the overall system.

A. User Interface Module

The frontend interface is implemented using HTML and JavaScript, providing a minimal chat window with a message input field and send button. The client establishes a WebSocket connection with the backend server upon page load. Incoming messages are dynamically appended to the chat display area without requiring page refresh. Input validation prevents transmission of empty or malformed messages.

B. Backend Server Module

The backend server is developed using Node.js and Express.js. Express initializes the HTTP server, which is subsequently used to attach the WebSocket server instance. The server manages active WebSocket connections, processes incoming messages asynchronously using Node.js's event-driven non-blocking architecture, and broadcasts messages to all connected clients. The non-blocking model ensures that multiple concurrent users can send and receive messages without queuing delays.

C. Redis Integration Module

Redis is integrated using the AWS ElastiCache managed service running Redis OSS engine on a cache.t3.micro node. The backend server establishes two Redis client connections: one for publishing messages to a designated chat channel and one for subscribing to incoming messages. When a message is received from a WebSocket client, it is published to the Redis channel. The subscriber connection receives the broadcast and forwards it to all active WebSocket connections on that server instance. This Pub/Sub design supports multi-instance deployments where Redis ensures message consistency across all nodes.

D. Docker Containerization Module

The application is containerized using a Dockerfile based on the official Node.js 18 image. The Dockerfile sets the working directory, copies package configuration files, installs npm dependencies, copies the application source code, exposes port 3000, and defines the server startup command. The resulting container image encapsulates the complete runtime environment, ensuring that the application behaves consistently on both local systems and cloud infrastructure.

E. AWS Cloud Infrastructure Module

The application is deployed on an AWS EC2 instance running Ubuntu 22.04 LTS, selected for its stability and compatibility with Docker. The t3.micro instance type, eligible under the AWS Free Tier, provides sufficient compute resources for development and demonstration purposes. AWS Security Groups are configured to permit inbound SSH access on port 22, application access on port 3000, and restricted Redis communication on port 6379 within the VPC. AWS ElastiCache provides the managed Redis node with encryption in transit enabled and a primary endpoint accessible exclusively from the EC2 instance.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15314

VI. RESULTS AND EVALUATION

The system was evaluated through comprehensive functional, integration, deployment, performance, and security testing. All test cases executed successfully, confirming that the application meets its defined requirements under normal operational conditions.

Table 2. Functional Test Case Results

Test Case ID	Test Description	Expected Result	Actual Result	Status
TC01	Load application in browser	Interface loads successfully	Interface loaded correctly	PASS
TC02	Establish WebSocket connection	Client connects to server	Connection established	PASS
TC03	Send message from client	Message transmitted to server	Message delivered	PASS
TC04	Message broadcasting	All clients receive message	Broadcast successful	PASS
TC05	Multiple user communication	Messages visible to all users	System synchronized	PASS
TC06	Redis connection validation	Backend connects to Redis endpoint	Connection established	PASS
TC07	Docker container startup	Application runs inside container	Container executed	PASS
TC08	EC2 deployment test	Application accessible via public IP	Application accessible	PASS
TC09	Server restart test	Application restarts without failure	System restarted successfully	PASS
TC10	Security group validation	Only allowed ports are open	Ports configured correctly	PASS

Performance evaluation confirmed that messages are delivered to all connected clients with minimal observable latency under concurrent usage. The Node.js event-driven architecture maintained stable CPU utilization during multi-user testing, and the Redis in-memory store ensured sub-millisecond message synchronization. Container restart testing validated that the Docker deployment recovers reliably following EC2 instance reboots.

Scalability was validated architecturally through the stateless backend design and Redis Pub/Sub synchronization. While a single EC2 instance was used for this demonstration, the architecture supports horizontal expansion by deploying additional Docker container replicas connected to the same Redis endpoint without requiring application-level code changes.

VII. CONCLUSION

This project successfully designed and implemented RishiBuild, a scalable real-time chat application that demonstrates the practical integration of WebSocket communication, Docker containerization, and AWS cloud infrastructure. The system achieves low-latency message delivery, consistent containerized deployment, distributed synchronization through Redis Pub/Sub, and secure cloud configuration using AWS Security Groups.

All ten functional test cases passed, confirming that the application meets its performance, reliability, and scalability objectives. The stateless backend architecture and Redis-managed message synchronization provide a foundation that supports horizontal scaling without architectural redesign. The project validates that combining Node.js, Docker, AWS

DOI: 10.17148/IJARCCE.2026.15314

EC2, and AWS ElastiCache constitutes a cost-effective and technically sound approach to building cloud-native real-time communication systems.

The implementation serves as a reproducible reference architecture for distributed chat systems and demonstrates modern DevOps practices applicable to production-grade deployments.

VIII. FUTURE ENHANCEMENTS

Several enhancements can extend the capabilities of this system. User authentication using JWT-based login and registration would enable identity tracking and private messaging. Integration of a persistent database such as MongoDB or PostgreSQL would support message history storage and retrieval. Deployment behind an AWS Application Load Balancer across multiple EC2 instances would eliminate the current single point of failure and provide high availability. A CI/CD pipeline using GitHub Actions or GitLab CI/CD would automate Docker image builds and deployments on each code commit, reducing manual intervention and improving release velocity. Integration of AWS CloudWatch monitoring and alerting would provide real-time visibility into application performance, container health, and error rates. Enabling Redis cluster mode with multiple nodes would further enhance message throughput and fault tolerance for high-traffic deployments. HTTPS and encrypted WebSocket (WSS) configuration using SSL certificates would secure all client-server communication in production environments.

REFERENCES

[1]. Tanenbaum, A. S., & Van Steen, M. (2017). Distributed Systems: Principles and Paradigms. Pearson Education.
[2]. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 2014(239).
[3]. Coulouris, G., Dollimore, J., & Kindberg, T. (2011). Distributed Systems: Concepts and Design. Addison-Wesley.
[4]. Node.js Official Documentation. (2024). Node.js API Reference. https://nodejs.org/en/docs
[5]. Express.js Official Documentation. (2024). Express.js Framework. https://expressjs.com
[6]. Redis Official Documentation. (2024). Redis Pub/Sub and Configuration. https://redis.io/docs
[7]. Amazon Web Services. (2024). Amazon EC2 User Guide for Linux Instances. https://docs.aws.amazon.com/ec2/
[8]. Amazon Web Services. (2024). Amazon ElastiCache for Redis Documentation. https://docs.aws.amazon.com/elasticache/
[9]. Docker Official Documentation. (2024). Dockerfile Reference. https://docs.docker.com
[10]. MDN Web Docs. (2024). WebSocket API Documentation. https://developer.mozilla.org

