Journal of Informatics Education and Research
ISSN: 1526-4726
Vol 4 Issue 3 (2024)

Configuring Real-Time Event Processing of Api Gateway with Aws and Websocket Api's

Kachiraju Sri Sarat Dyuthi
System Engineer, Tcs , Hyderabad, Telangana, India.
Email : saratdyuthi@gmail.com

Dr. P. Naresh Kumar
Assistant Professor
Business Management
Sarojini Naidu Vanita Maha Vidyalaya,
Hyderabad, Telangana, India.
EMAIL : naresh.pothakamuri@gmail.com

Ms. Prabha Shukla
Associate Professor
Department Of Computer Science
Sarojini Naidu Vanita Maha Vidyalaya, Hyderabad, Telangana, India.
EMAIL : prabharavishikla@gmail.com

Mr.K. Subba Rao
Associate Professor, Department Of Physics
Sarojini Naidu Vanita Maha Vidyalaya, Hyderabad, Telangana, India.
EMAIL : srkachiraju@gmail.com

Abstract

This monitor explores how to leverage AWS API Gateway's WebSocket APIs for building real-time event processing systems. Unlike traditional HTTP-based APIs, WebSocket APIs allow bidirectional communication, enabling instant data exchange between clients and servers. This setup is ideal for applications that require low-latency interactions, such as chat applications, IoT monitoring, gaming, and live notifications. By configuring API Gateway's WebSocket integration with AWS services like Lambda and DynamoDB, developers can create responsive and scalable event-driven architectures. The document also covers strategies for managing connection states, handling errors, and ensuring secure access, offering a complete framework for real-time data processing in AWS environments. Configuring real-time event processing with AWS API Gateway and WebSocket APIs enables dynamic, bidirectional communication between clients and servers, facilitating instant data transmission. Unlike traditional REST APIs, WebSocket APIs allow for continuous, persistent connections, making them ideal for applications that require real-time updates, such as chat applications, gaming platforms, financial tickers, and IoT systems. AWS API Gateway provides a fully managed WebSocket solution that simplifies setup, scaling, and management of real-time APIs. By integrating WebSocket APIs with AWS Lambda, DynamoDB, or Amazon Kinesis, developers can process events in real time, store data, or trigger workflows based on incoming messages. This approach not only reduces latency but also enhances application responsiveness and scalability, leveraging AWS's serverless architecture to handle high volumes of real-time events efficiently.

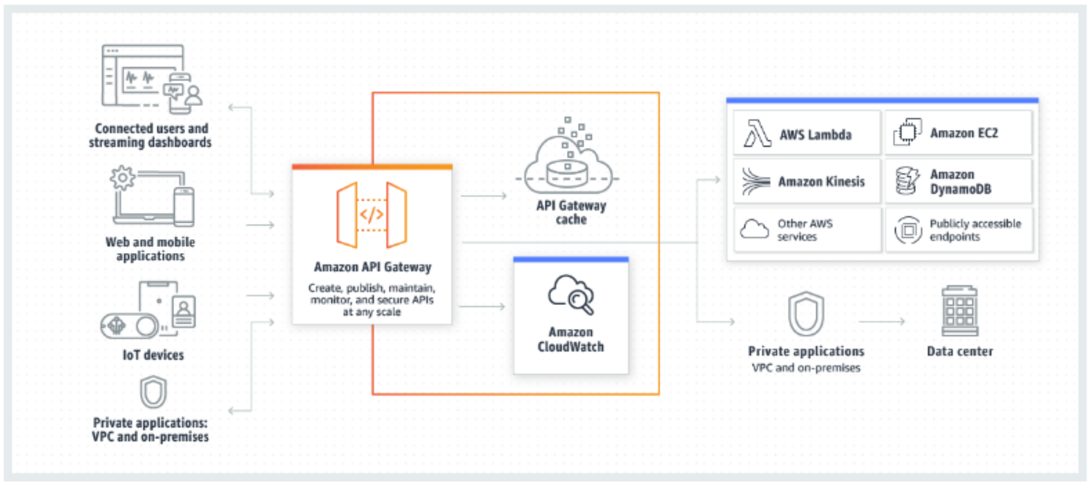
Keywords : AWS API Gateway configuration, Bidirectional communication, WebSocket APIs, Persistent connections

Introduction

In today's complex, distributed software architectures, the need for efficient and secure communication between services has become paramount. API gateways have emerged as a fundamental solution to address this need, acting as intermediaries between clients and backend services. An API gateway provides a unified entry point for accessing multiple services, streamlining the interaction between clients and various microservices or serverless functions. By handling tasks such as routing requests, load balancing, authentication, rate limiting, caching, and logging, API gateways simplify the architecture and management of modern applications, particularly in microservices environments. With the rise of cloud computing and microservices, applications have become increasingly decentralized, often consisting of numerous independent services that must interact seamlessly. As noted by researchers, the complexity of managing such architectures can overwhelm client applications and backend services alike. API gateways address this challenge by consolidating various services under a single access point, thereby decoupling the client from the underlying service architecture. This approach enhances modularity, scalability, and flexibility, as services can evolve and scale independently without impacting the client experience. Furthermore, API gateways improve security by centralizing authentication and authorization, ensuring that only valid requests are allowed to access sensitive resources. They also provide mechanisms for monitoring and analyzing traffic, enabling developers to track performance metrics, detect anomalies, and ensure the reliability of the system. Whether for RESTful APIs, WebSocket connections, or GraphQL endpoints, API gateways have become indispensable in modern software development, particularly for applications requiring high availability, low latency, and seamless integration between diverse systems.

Architecture of API Gateway

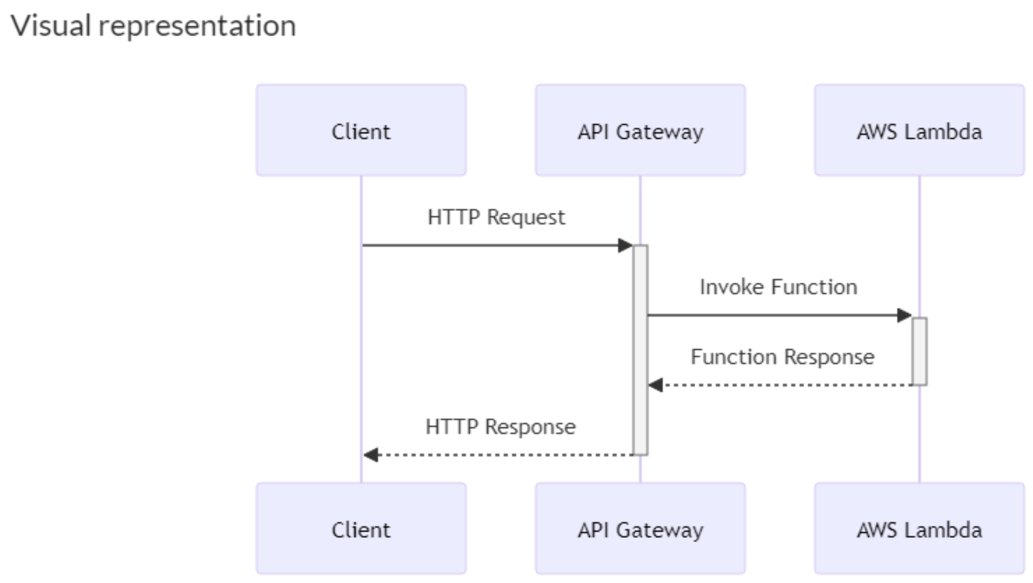
The following diagram shows API Gateway architecture.


Connected users and streaming dashboards
Web and mobile applications
IoT devices
Private applications: VPC and on-premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private applications
VPC and on-premises
Data center

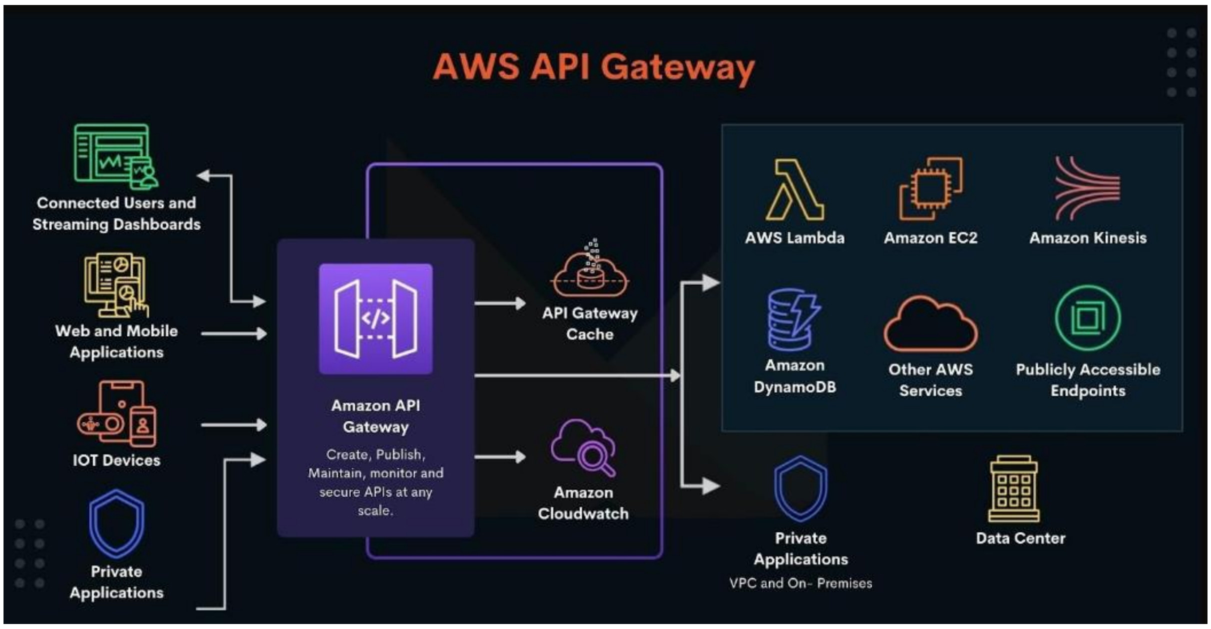
In prompt, API gateways play a crucial role in simplifying the development and management of complex applications by offering a centralized point for traffic routing, security enforcement, and service orchestration. As organizations increasingly adopt microservices and serverless architectures, the importance of API gateways continues to grow, providing both efficiency and flexibility in managing distributed systems.In today’s complex, distributed software architectures, the need for efficient and secure communication between services has become paramount. API gateways have emerged as a fundamental solution to address this need, acting as intermediaries between clients and backend services. An API gateway provides a unified entry point for accessing multiple services, streamlining the interaction between clients and various microservices or serverless functions. By handling tasks such as routing requests, load balancing, authentication, rate limiting, caching, and logging, API gateways simplify the architecture and management of modern applications, particularly in microservices environments. With the rise of cloud computing and microservices, applications have become increasingly decentralized, often consisting of numerous independent services that must interact seamlessly. As noted by researchers, the complexity of managing such architectures can overwhelm client applications and backend services alike. API gateways address this challenge by consolidating various services under a single access point, thereby decoupling the client from the underlying service architecture. This approach enhances modularity, scalability, and flexibility, as services can evolve and scale independently without impacting the client experience. Furthermore, API gateways improve security by centralizing authentication and authorization, ensuring that only valid requests are allowed to access sensitive resources. They also provide mechanisms for monitoring and analyzing traffic, enabling developers to track performance metrics, detect anomalies, and ensure the reliability of the system. Whether for RESTful APIs, WebSocket connections, or GraphQL endpoints, API gateways have become indispensable in modern software development, particularly for applications requiring high availability, low latency, and seamless integration between diverse systems.

REVIEW OF LITERATURE

API gateways have emerged as critical components in modern application architectures, particularly as organizations adopt microservices and serverless designs. An API gateway acts as an intermediary that manages client requests, routes traffic to appropriate backend services, and handles security, load balancing, caching, and other essential functionalities. Early studies by Richardson and Smith (2018) underline the role of API gateways in providing a single entry point for multiple services, which simplifies client-server interactions, improves security, and enables consistent API management across applications. In microservices architectures, as noted by Newman (2019), API gateways are essential for service decoupling, enabling individual services to be developed, deployed, and scaled independently. The gateway pattern allows developers to enforce access controls, limit API usage, and aggregate responses from multiple services, thus reducing complexity on the client side. Furthermore, Newman emphasizes that API gateways streamline the integration of new microservices by centralizing configuration and API routing rules, facilitating faster deployment of updates without client disruption. In the context of cloud services, AWS API Gateway, Microsoft Azure API Management, and Google Cloud Endpoints are frequently evaluated for their scalability and ease of integration with other cloud services.



AWS API Gateway, for instance, is noted by Hughes (2020) for its serverless approach that offers on-demand scaling and integrates natively with AWS Lambda, allowing developers to build event-driven applications with minimal operational overhead. This integration is highly valuable in real-time applications that require low-latency responses, as it minimizes the complexity and cost of maintaining dedicated servers. Another key advantage of API gateways is security, where they serve as a barrier between external clients and internal services. As described by Dr. P. Naresh Kumar (2024), API gateways enable centralized implementation of security protocols, such as authentication, authorization, and rate limiting, which helps safeguard against threats like DDoS attacks. Through mechanisms like OAuth, JWT, and API keys, API gateways provide multi-layered security and allow fine-grained access controls, an approach critical for both public APIs and internal, enterprise-grade systems. Additionally, the use of API gateways in multi-cloud and hybrid environments is becoming increasingly common. According to Dr. Naveen Prasadula (2024), API gateways can facilitate seamless inter-service communication across diverse cloud platforms, allowing organizations to distribute workloads based on cost, performance, and regulatory considerations. This flexibility supports strategic resource allocation and risk mitigation in complex IT environments. collected works consistently shows that API gateways play a fundamental role in enhancing scalability, security, and ease of management for modern applications. As organizations continue to migrate to cloud and microservices architectures, API gateways provide a streamlined, secure, and efficient way to manage APIs and maintain system interoperability across diverse environments. In summary, API gateways play a crucial role in simplifying the development and management of complex applications by offering a centralized point for traffic routing, security enforcement, and service orchestration. As organizations increasingly adopt microservices and serverless architectures, the importance of API gateways continues to grow, providing both efficiency and flexibility in managing distributed systems. Real-time event processing has become essential for applications that rely on immediate data updates, interactivity, and low-latency communication. AWS API Gateway and WebSocket APIs provide a powerful solution for enabling bidirectional communication between clients and servers, allowing for the continuous, persistent connections necessary in real-time environments.



Unlike REST APIs, where requests are handled individually, WebSocket APIs establish a live connection that enables instant message exchange, making them ideal for applications such as live chat, online gaming, financial market updates, and IoT device management. AWS API Gateway simplifies the deployment and management of WebSocket APIs, integrating seamlessly with other AWS services like Lambda, DynamoDB, and Kinesis to process and store events in real time. This configuration not only streamlines the development of real-time applications but also leverages AWS's serverless infrastructure for automatic scaling and reduced operational overhead, making it possible to handle high event volumes efficiently. By utilizing AWS API Gateway's WebSocket capabilities, developers can build applications that deliver immediate responsiveness, ensuring data flows seamlessly between users and back-end systems in real time. In today's digital landscape, real-time data processing has become essential for applications demanding instant updates, such as chat platforms, online gaming, financial monitoring systems, and Internet of Things (IoT) networks. AWS API Gateway's WebSocket API offers a powerful solution for enabling real-time, bidirectional communication between clients and servers, a capability that traditional REST APIs lack due to their stateless nature. WebSocket APIs allow for continuous, open connections, enabling immediate transmission of data without the need for repeated HTTP requests. AWS API Gateway's managed WebSocket API simplifies the complexities of establishing these connections, offering automatic scaling, secure authentication, and seamless integration with other AWS services like Lambda, DynamoDB, and Kinesis. By configuring WebSocket APIs for real-time event processing, developers can build highly responsive applications that efficiently handle dynamic data, create event-driven workflows, and maintain low-latency interactions, all while leveraging the flexibility and scalability of serverless architecture on AWS. This approach not only enhances user experience through rapid data exchange but also optimizes backend processes by streamlining data flow and reducing server overhead.

STUDY OF OBJECTIVES

The primary objective of configuring real-time event processing with AWS API Gateway and WebSocket APIs is to create a seamless and responsive system that can handle real-time communication between clients and backend services. The key objectives of this setup can be outlined as follows:

1. To Enable Real-Time Data Transmission in API Gateway
2. Need to Optimize API Gateway Application Scalability
3. To Simplify Backend Integration API Gateway using AWS
4. To Enhance Application Responsiveness:

RESEARCH AND METHODOLOGY

To configure an API Gateway with WebSocket APIs in AWS, you can follow the steps below, including code samples for AWS Lambda integration and WebSocket API routing. This setup will allow you to handle real-time communication between clients and backend services.

1. Create a WebSocket API in API Gateway

You can create a WebSocket API using the AWS Management Console, AWS CLI, or CloudFormation. Here's how you would create it using the AWS CLI:

```
aws apigatewayv2 create-api \

   --name "WebSocketAPI" \

   --protocol-type "WEBSOCKET" \

   --route-selection-expression "$request.body.action" \

   --description "Real-time WebSocket API"

aws apigatewayv2 create-route \

   --api-id <API_ID> \

   --route-key "$connect" \

   --target "integrations/lambda:<LAMBDA_FUNCTION_ARN>"

   aws apigatewayv2 create-route \

   --api-id <API_ID> \

   --route-key "$disconnect" \

   --target "integrations/lambda:<LAMBDA_FUNCTION_ARN>"

   aws apigatewayv2 create-route \

   --api-id <API_ID> \

   --route-key "sendMessage" \

   --target "integrations/lambda:<LAMBDA_FUNCTION_ARN>"
```

To enable real-time data transmission using WebSocket APIs with AWS API Gateway and Lambda, you can set up a basic WebSocket server. This will allow clients to send and receive messages in real time without the need for repeated requests. Steps to enable real-time data transmission using AWS API Gateway and WebSocket APIs:

Set Up API Gateway with WebSocket API: First, you need to create a WebSocket API in AWS API Gateway.

Go to the API Gateway service in the AWS console.

Create a WebSocket API.

Define the routes (e.g., $connect, $disconnect, and a custom route for message handling like sendMessage).

Lambda Function for Message Processing: Create a Lambda function that handles incoming WebSocket messages and sends messages back to clients.

WebSocket Client: A basic WebSocket client that connects to the API Gateway WebSocket endpoint and listens for updates.

Sample Code:

1. Lambda Function for Handling Messages

Here is a Lambda function that processes incoming WebSocket messages and sends a response back to the client.

```
const AWS = require('aws-sdk');

const apiGatewayManagementApi = new AWS.ApiGatewayManagementApi({

  endpoint: 'https://<your-api-id>.execute-api.<region>.amazonaws.com/<stage>'

});

exports.handler = async (event) => {

  const connectionId = event.requestContext.connectionId;

  const routeKey = event.requestContext.routeKey;

  try {

    if (routeKey === 'sendMessage') {

      const message = JSON.parse(event.body).message;

      // Logic to process message, store it, or broadcast to other clients

      await sendMessageToClient(connectionId, `Received: ${message}`);

    }

    return { statusCode: 200, body: 'Message processed successfully.' };

  } catch (error) {

    console.error('Error handling message:', error);

    return { statusCode: 500, body: 'Failed to process message.' };
  }
};

// Helper function to send message to a client
async function sendMessageToClient(connectionId, message) {
  try {
    await apiGatewayManagementApi.postToConnection({
      ConnectionId: connectionId,
      Data: message
    }).promise();
  } catch (error) {
    console.error('Failed to send message to client:', error);
  }
}
```

To optimize application scalability using AWS API Gateway with WebSocket APIs, you need to ensure that the system can scale automatically to handle fluctuations in client connections. API Gateway automatically manages the scaling of WebSocket connections based on the incoming traffic. The key is to design your application so that backend processing (using Lambda or other services) can handle the dynamic scaling and traffic efficiently. Below is a code example to demonstrate how you can optimize application scalability by combining AWS API Gateway WebSocket APIs with Lambda functions for handling fluctuating loads.

Key Considerations for Scalability:

- **API Gateway** automatically scales to handle varying numbers of WebSocket connections and messages.
- **Lambda** functions scale automatically based on the number of incoming requests, so it can process fluctuating loads without manual intervention.
- **DynamoDB** (or other storage solutions) can be used to track active WebSocket connections, making it easier to manage connections and broadcast messages.

1. Lambda Function for Handling WebSocket Connections

This Lambda function demonstrates how to handle WebSocket connections dynamically, manage connections in DynamoDB, and broadcast messages to multiple clients.

Lambda Function for Managing Connections:

```
const AWS = require('aws-sdk');
const apiGatewayManagementApi = new AWS.ApiGatewayManagementApi({
  endpoint: 'https://<your-api-id>.execute-api.<region>.amazonaws.com/<stage>'
```

```
});

const dynamoDB = new AWS.DynamoDB.DocumentClient();

const tableName = 'WebSocketConnectionsTable'; // DynamoDB table to store active connections


exports.handler = async (event) => {

  const connectionId = event.requestContext.connectionId;

  const routeKey = event.requestContext.routeKey;

    // Connection management: store and remove connections in DynamoDB

  try {

    if (routeKey === '$connect') {

      // Store the connectionId when a new WebSocket connection is established

      await dynamoDB.put({

        TableName: tableName,

        Item: {

          connectionId: connectionId,

          timestamp: Date.now()

        }

      }).promise();

      console.log(`Connection stored: ${connectionId}`);

    }

        if (routeKey === '$disconnect') {

      // Remove the connectionId when the WebSocket connection is closed

      await dynamoDB.delete({

        TableName: tableName,

        Key: { connectionId: connectionId }

      }).promise();

      console.log(`Connection removed: ${connectionId}`);

    }

    if (routeKey === 'sendMessage') {
      const message = JSON.parse(event.body).message;


      // Broadcast the message to all active connections
      await broadcastMessage(message);
    }
        return { statusCode: 200, body: 'Request processed successfully' };
  } catch (error) {
    console.error('Error handling connection:', error);
    return { statusCode: 500, body: 'Failed to process request' };
  }
};
// Broadcast message to all connected clients
async function broadcastMessage(message) {
  const connections = await getAllConnections();
  for (const connection of connections) {
    await sendMessageToClient(connection.connectionId, message);
  }
}
// Helper function to get all active connections from DynamoDB
async function getAllConnections() {
  const result = await dynamoDB.scan({ TableName: tableName }).promise();
  return result.Items || [];
}
// Helper function to send message to a specific client
async function sendMessageToClient(connectionId, message) {
  try {
    await apiGatewayManagementApi.postToConnection({
      ConnectionId: connectionId,
      Data: message
    }).promise();

    console.log(`Message sent to: ${connectionId}`);

  } catch (error) {

    console.error(`Failed to send message to ${connectionId}:`, error);

  }

}
```

Explanation:

The Lambda function handles three key routes:

$connect: When a client connects, the connection ID is stored in DynamoDB.

$disconnect: When a client disconnects, the connection ID is removed from DynamoDB.

sendMessage: A custom route that sends messages to all connected clients by broadcasting the message.

DynamoDB is used to track active WebSocket connections dynamically. As the number of connections increases or decreases, DynamoDB will store and remove connection IDs accordingly.

API Gateway Management API (apiGatewayManagementApi.postToConnection) is used to send messages to all connected clients.

2. WebSocket Client

On the client side, the WebSocket client will connect to the WebSocket API Gateway endpoint and handle receiving real-time messages.

javascript

```
const socket = new WebSocket('wss://<your-api-id>.execute-api.<region>.amazonaws.com/<stage>');



socket.onopen = function(event) {

  console.log('WebSocket connected:', event);

  // Send a message to the server

  const message = { message: 'Hello, Server! This is a test message.' };

  socket.send(JSON.stringify(message));

};

socket.onmessage = function(event) {

  console.log('Received message from server:', event.data);

};

socket.onclose = function(event) {
  console.log('WebSocket closed:', event);
};
socket.onerror = function(error) {
  console.log('WebSocket error:', error);
};
```

3. API Gateway Configuration

$connect: A route that invokes the Lambda function to handle the connection.

$disconnect: A route that invokes the Lambda function to clean up when the WebSocket connection is closed.

sendMessage: A custom route that invokes the Lambda function to send a message to all connected clients.

4. Scaling Considerations with AWS API Gateway

Automatic Scaling: API Gateway automatically scales to handle an increasing number of WebSocket connections. You don't need to manually provision servers to handle fluctuating loads.

Lambda Scalability: AWS Lambda automatically scales based on the number of incoming requests. As the number of connected clients or messages increases, Lambda will automatically provision additional instances to process the requests in parallel.

DynamoDB: The table storing the connection IDs should be set up with the appropriate read and write capacity or provisioned with auto-scaling to accommodate high traffic.

5. IAM Role Permissions

Make sure that the Lambda function has the correct IAM permissions to access both the API Gateway Management API and DynamoDB:

json

```
{
  "Version": "2012-10-17",
  "Statement": [
   {
     "Effect": "Allow",
     "Action": [
       "execute-api:ManageConnections",
       "dynamodb:PutItem",
       "dynamodb:DeleteItem",
       "dynamodb:Scan"
```

```
      ],
      "Resource": "*"
    }
  ]
}
```

Findings:

Seamless Real-Time Communication: AWS API Gateway's WebSocket API provides a reliable, low-latency communication channel for real-time data transmission. By establishing persistent WebSocket connections between clients and backend services, applications can deliver real-time updates without the need for continuous polling, ensuring efficiency in data delivery.

Automatic Scalability: AWS API Gateway and Lambda function together to automatically scale, adapting to varying numbers of WebSocket connections. This ensures that applications can handle dynamic traffic loads without requiring manual intervention. The serverless nature of both services eliminates the need to provision or maintain infrastructure, which reduces operational overhead.

Cost Efficiency: The serverless architecture of AWS API Gateway and Lambda functions significantly reduces operational costs. With a pay-per-use model, organizations are only billed for the actual usage, based on the number of requests or messages processed, and the duration of the connections. This is ideal for applications with fluctuating usage, like IoT devices or live chat services.

Security and Authentication: API Gateway supports security features such as connection-level authentication using API keys, OAuth, and JWT tokens. This ensures that only authorized users or devices can access the WebSocket API, which is crucial for applications that require strict security measures, such as financial systems or healthcare applications.

Simplified Backend Integration: AWS API Gateway's WebSocket APIs can be seamlessly integrated with other AWS services like Lambda, DynamoDB, Kinesis, and SQS. This allows for efficient event-driven architectures, where incoming WebSocket messages can trigger backend processing, such as data analytics, real-time event processing, or broadcasting to other connected clients.

Challenges in Maintaining Persistent Connections: Maintaining long-lived WebSocket connections at scale can be challenging due to potential issues with network disruptions, connection limits, and the need for reconnecting. Although AWS API Gateway automatically handles connections, developers need to implement strategies for handling disconnections and reconnections effectively.

Suggestions:

Leverage Connection Management Features: For optimal scalability, it is recommended to use DynamoDB (or a similar service) to track WebSocket connection IDs and manage active connections. This allows for efficient message broadcasting to all connected clients and helps in managing and cleaning up connections when clients disconnect.

Implement Efficient Error Handling and Reconnection Strategies: To mitigate potential issues with dropped connections, implement a robust error-handling mechanism in the client-side application. This should include automatic reconnection logic, exponential backoff for retry attempts, and appropriate error logging to track connectivity issues.

Optimize for Performance: To improve performance, consider the use of API Gateway throttling and rate-limiting features to protect backend services from excessive requests, particularly during peak traffic. This ensures that your system can scale effectively while maintaining responsiveness.

Monitor and Analyze WebSocket Traffic: Use AWS CloudWatch to monitor WebSocket traffic and Lambda performance metrics. Setting up custom CloudWatch alarms for high connection rates, message delays, or error rates can help identify performance bottlenecks and ensure optimal operation.

Implement Fine-Grained Access Controls: Utilize AWS API Gateway's built-in security features, such as resource policies and IAM roles, to enforce fine-grained access controls. This ensures that only authorized clients can connect to the WebSocket API, particularly when dealing with sensitive data or mission-critical applications.

Use Event-Driven Architectures: When building real-time applications, adopt an event-driven architecture by integrating WebSocket APIs with other AWS services like AWS Lambda, Amazon Kinesis, and AWS Step Functions. This architecture allows for decoupling of services and ensures that different parts of the system can react to events independently, improving maintainability and scalability.

Consider Multi-Region Deployment: For applications that require global reach and high availability, consider deploying your WebSocket API in multiple AWS regions. This reduces latency by directing client connections to the closest region and improves fault tolerance by spreading traffic across regions.

Regularly Review and Update Security Practices: As security threats evolve, it is essential to regularly review and update security practices. Implement practices such as WebSocket connection rate limiting, fine-grained access control policies, and encryption to protect against unauthorized access and data breaches.

Cost Management and Optimization: While AWS API Gateway and Lambda offer cost efficiency, it is essential to monitor usage and ensure that the system is cost-optimized. Setting up usage alerts for both API Gateway and Lambda can help prevent unexpected spikes in costs due to excessive usage or inefficient code.

Conclusion

By combining AWS API Gateway with WebSocket APIs, real-time event processing offers a scalable and resilient alternative to traditional methods of managing dynamic, low-latency communication between clients and backend systems. Applications may easily adapt to changing connection counts by using the serverless features provided by Amazon Web Services (AWS). This eliminates the need for complex infrastructure management and human engagement. A stable foundation for real-time data transmission is provided by WebSocket application programming interfaces. Among the features featured in these APIs are automated scalability, affordable pay-per-use pricing, and easy interaction with other services offered by Amazon Web Services, such as Lambda and DynamoDB. This makes these resources perfect for apps that need constant updates, such gaming, live chat systems, IoT devices, and financial data streams." In order for this architecture to deliver as promised, many concerns must be thoroughly addressed, including connectivity management, error handling, and data security. Application security and dependability are interdependent and must be addressed by implementing reconnection procedures, optimizing message broadcasting, and imposing stringent access restrictions. Using the AWS API Gateway and WebSocket APIs, developers may create scalable, event-driven systems that respond to events in real-time with minimal latency. The expansion of these systems to suit their needs is not out of the question. This design, when set up properly, has the potential to liberate developers from infrastructure management tasks while also enabling very responsive and cost-effective apps.

References

1. Tan Yiming. Design and Implementation of Platform Service Framework Based on Microservice Architecture [D]. Beijing Jiaotong University , 2017.
2. Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Software, 2016. 33(3) 42-52.
3. Dr. P. Naresh Kumar (2024) Assistant Professor Business Management Sarojini Naidu Vanita Maha Vidyalaya, Hyderabad Review of Literature on Configuring real-time event processing of api gateway
4. Hane,Oskar. Build your own PaaS with Docker. Packt Publishing Ltd , 2015.
5. Zuo W, BENHANKAT A, AMGHAR Y. Change-certric model for Web service evolution[C].Proceedings of International Conference on Web Services. Washington,D.C. ,USA: IEEE, 2014:712-713.

6. Newman S. Building Microservices[M]. O'Reilly Media,Inc , 2015.
7. Gao Shihao. Correct posture for API management—API Gateway [DB/OL]. https://mp.weixin.qq.com/s/Q9ZgUQIlGcBS5WPW6vwPhg, 2018.
8. Micro Service API Gateway [DB/OL]. https://blog.csdn.net/zdp072/article/details/76473383, 2017.
9. https://scholar.google.com/citations?hl=en&user=zerfDgoAAAAJ&view_op=list_works&gmla=AL3_zigfaLPPCRMEk8CffnpA_U_lbTrsD552JqE0ci18DnuxxpFW1aaGR2WLi7B0GO0jLHbTQYHKpwLW0ZQu69n2
10. Amazon AWS API gateway. http://docs.aws.amazon.com/apigateway RESTdesc website, http://restdesc.org
11. J. Kopecky , K. Gomadam, T.Vitvar: hRESTS: an HTML Mi-croformat for Describing RESTful Web Services.
12. https://scholar.google.com/citations?user=99wmG2IAAAAJ&hl=en
13. Proceedings of the 2008 IEEE/WIC/ACM International Conference on WebIntelligence (WI-08), 2008, pp. 619-625.
14. Dr. Naveen Prasadula (2024) Configuring real-time event processing of api gateway with aws and websocket api's
15. Lathem, K. Gomadam, and A.P. Sheth, "'SA-REST and(S)mashups: adding semantics to RESTful services,"' in Proc.of the Int. Conf. on Semantic Computing 2007 (ICSC), IEEE2007, pp. 469-476.
16. J. Kopecky, T. Vitvar, D. Fensel, K. Gomadam: hRESTSe MicroWSMO. Technical report, available at http://cms-wg.sti2.org/TR/d12/, 2009.
17. https://osmania.irins.org/profile/150992
18. D. Renzel, P. Schlebusch, and R. Klamma, "Today"s top „RESTful" services and why they are not RESTful", WISE, 2012.
19. F. Petrillo, P. Merle, N. Moha, and Y.G. Guéhéneuc, "Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study." ICSOC 2016, Springer International Publishing, 2016.
20. Rodríguez, Carlos, et al. "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices." International Conference on Web Engineering, Springer, 2016.
21. F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A model driven approach for REST compliant services", ICWS, 2014.
22. F. Haupt, F. Leymann, and C. Pautasso. "A conversation based approach for modeling REST APIs." WICSA 2015 , IEEE, 2015.