Serverless AI Chatbot for Cloud Cost Optimization Using AWS SageMaker

1st Sumedh Ambapkar
Indiana University Bloomington
suambapk@iu.edu
Solution Architect and Sagemaker

2nd Krishna Prasadh Subramani
Indiana University Bloomington
krsubram@iu.edu
Integration for Cost Explorer and Lambda Dev

3rd Aryan Agrawal
Indiana University Bloomington
aryagraw@iu.edu
NLP and API Gateway Dev

4th Arju Singh
Indiana University Bloomington
singarju@iu.edu
Solution Integration and Step functions

***Abstract*—Cloud cost optimization remains a critical challenge for organizations managing dynamic workloads on AWS. Traditional cost management tools provide raw data but lack real-time insights and proactive recommendations. This paper presents a Serverless AI Chatbot leveraging Amazon Lex for natural language processing (NLP) and AWS SageMaker for predictive analysis, enabling organizations to optimize resource utilization and reduce AWS service costs.**

The proposed architecture automates data collection, processes user queries, and delivers rightsizing recommendations for EC2 instances, Lambda functions, and S3 services. By integrating a serverless architecture with machine learning capabilities, the solution offers scalability, low operational overhead, and faster response times compared to manual analysis. The chatbot supports real-time queries, providing actionable insights on underutilized resources and cost-saving opportunities.

The solution continuously improves using historical data, ensuring recommendations remain accurate and adaptable to evolving workloads. This approach empowers organizations to make data-driven decisions, enhancing cloud efficiency and minimizing waste.

***Index Terms*—Cloud Cost Optimization, Serverless Architecture, AWS SageMaker, Natural Language Processing, Machine Learning**

I. Introduction

Modern cloud environments offer scalability and flexibility, but improper resource allocation leads to inflated costs. AWS services like EC2, Lambda, and S3 often remain underutilized or misconfigured, resulting in wasted resources. Existing cost management tools require manual interpretation, creating a gap between data availability and actionable insights. To bridge this gap, we designed a Serverless AI Chatbot capable of interpreting cost-related queries, retrieving usage data, and recommending optimized configurations.

II. Problem Statement

Cloud resource management is complex, with rapidly fluctuating workloads leading to over-provisioned or underutilized services. Organizations face challenges in interpreting cost data, identifying inefficiencies, and applying rightsizing strategies promptly. Manual analysis of AWS Cost Explorer and CloudWatch data is time-consuming and prone to human error, often resulting in delayed optimization efforts and increased operational expenses. The key problems addressed include:

- Lack of real-time visibility into underutilized resources (EC2, Lambda, S3)
- Difficulty generating accurate, timely rightsizing recommendations
- Inefficient manual data processing and analysis
- Inability to adapt recommendations to evolving usage patterns. Our chatbot tackles these problems by automating data retrieval, analysis, and recommendations, enabling cost-efficient resource management.

III. Technical Approach

The chatbot system follows an event-driven architecture, integrating AWS services to process user queries asynchronously. The flow is as follows:

1) **User Interaction:** The user interacts with a frontend chatbot (or uses Postman for testing) to ask about AWS service usage or instance details.
2) **API Gateway:** Acts as a secure entry point, forwarding user requests to the first AWS Lambda function.
3) **Lambda 1 (ApiToLexHandler)**
   - Preprocesses the user query.
   - Ensures structured input (e.g., "from *date* to *date*").
   - Sends the refined request to Amazon Lex.
4) **Amazon Lex:** Matches the query to an intent:
   - **CheckAWSUsage:** Extracts:
     a. serviceName (e.g., EC2, S3).
     b. dateRange (e.g., "from 2024-01-01 to 2024-03-01").
   - **CheckInstanceSize:** Extracts:
     a. instanceIid (e.g., "i-1234567890abcdef").
   - Triggers the second Lambda function to process the request.
5) **Lambda 2 (LexToSQSHandler):**
   For CheckAWSUsage intent:

- Extracts fromDate and toDate using regex.
- Assigns a unique requestId
- Sends a structured message (with requestId, serviceName, fromDate, and toDate) to Amazon SQS.

For CheckInstanceSize intent:

- Extracts instanceId and assigns a requestId.
- Sends a structured message (with requestId and instanceId) to SQS.

6) **Amazon SQS:**

- The request for asynchronous processing and decoupling the architecture.
- Ensures scalability for handling multiple user queries.

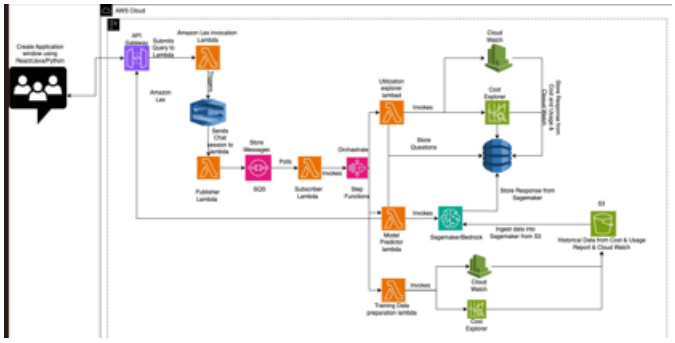


Fig. 1. Architecture Diagram.

IV. Methodology

A. Serverless AI Chatbot for Cloud Cost Optimization

Let me walk through every step of your architecture diagram, explaining each component's function and how they interact:

1) User Interface and Initial Interaction: 1. **Create Application Window (Left Side)** - Users access the application through a React/Java/Python frontend - This interface provides the chat window and visualization of cost analysis - User queries are captured here and sent to the backend services

2. **API Gateway** - Receives HTTP requests from the frontend application - Acts as the single entry point for all user interactions - Authenticates requests and routes them to appropriate Lambda functions - Submits the user's query to the Amazon Lex invocation Lambda

3. **Amazon Lex Invocation Lambda** - Processes the incoming query from API Gateway - Formats the request for Amazon Lex processing - Handles session management for ongoing conversations - Triggers Amazon Lex for natural language understanding

4. **Amazon Lex** - Parses user queries using natural language understanding - Identifies intents (what the user wants to do) and entities (specific AWS services/resources mentioned) - Extracts key parameters from user queries (e.g., time periods, cost thresholds, specific services) - Sends structured interpretation of user query back to Lambda

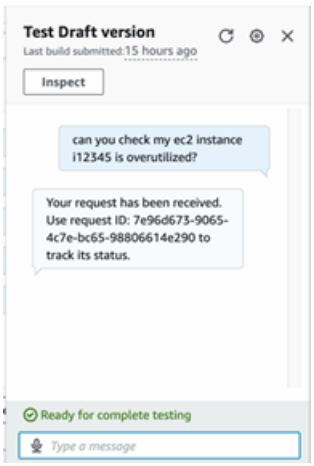


Fig. 2. ChatBot.

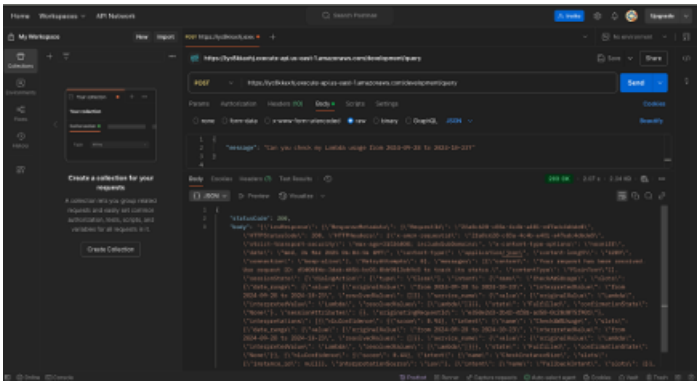


Fig. 3. Postman Request.

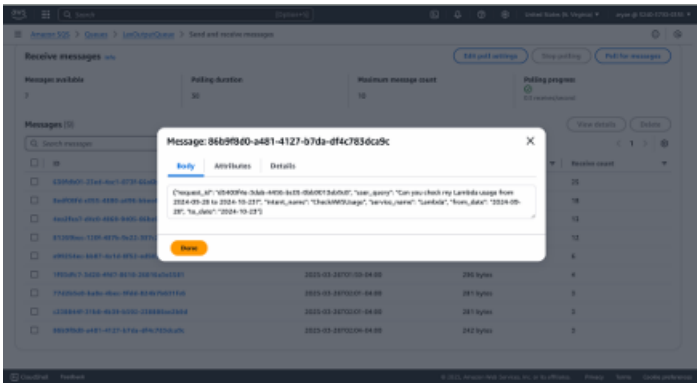


Fig. 4. Lex Output.

5. **Amazon Lex Service Relationship** - Sends the chat session to the next Lambda function for processing - Maintains context between multiple user interactions

6. **Publisher Lambda** - Receives structured data from Amazon Lex - Records the conversation history and user queries - Formats messages for storage in the database - Publishes messages to SQS for asynchronous processing

7. **SQS (Simple Queue Service)** - Stores user messages as events in a queue - Decouples message reception from processing - Ensures message durability even during high load - Polls for new messages to be processed

8. **Subscriber Lambda** - Consumes messages from the SQS queue - Determines which backend services need to be called based on the user's intent - Triggers the appropriate orchestration workflow - Maintains the state of ongoing conversations

9. **Step Functions** - Orchestrates complex workflows across multiple Lambda functions - Manages state transitions between different processing steps - Coordinates parallel execution of data retrieval and analysis tasks - Ensures reliable execution of the end-to-end processing pipeline

10. **Central Database (Blue Cylinder in Center)** - Stores all persistent data for the application - Contains user conversation history, recommendations, and query results - Enables stateful interactions across multiple Lambda invocations - Provides centralized storage for responses from various services

11. **Utilization Explorer Lambda** - Analyzes resource utilization patterns across AWS services - Identifies underutilized resources (EC2 instances, RDS databases, etc.) - Calculates potential savings from rightsizing or terminating resources - Stores questions about utilization in the central database - Invokes Cost Explorer for additional cost data

12. **Cost Explorer Integration** - Provides real-time cost breakdowns by service, region, and tag - Fetches historical spending trends for comparative analysis - Supplies data for identifying cost anomalies and unexpected spending - Sends responses back to the central database for storage - Reports usage data to CloudWatch for monitoring

13. **Model Predictor Lambda** - Invokes the SageMaker/Bedrock ML models for prediction tasks - Sends contextual information from the user query to the models - Processes model responses into actionable recommendations - Formats model outputs into natural language responses - Stores prediction results in the central database

14. **SageMaker/Bedrock Integration** - Deploys machine learning models trained for cost optimization - Uses historical usage data to refine predictions - Continuously updates recommendations based on workload patterns

V. Preliminary Results:

Model Prediction Lambda: We are still working on the logic of calling endpoint allowing parallel processing for multiple requests.

Training Data Preparation Lambda: This Lambda function retrieves cost and usage data, as well as CloudWatch metrics. We successfully triggered it to extract historical data and store it in an S3 bucket, making the data available for SageMaker to train the model. Currently, the Lambda is not running due to limited resources in our account. However, in a production environment, it can be configured to trigger with each user request or run on a scheduled interval to continually pull historical data into S3. Below is a screenshot showcasing the Lambda's output, confirming its role in collecting historical data.

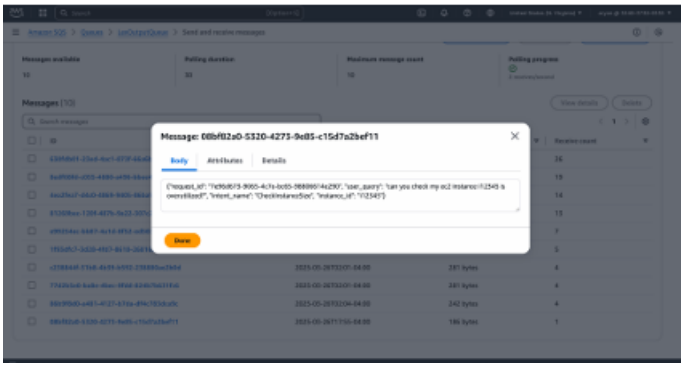


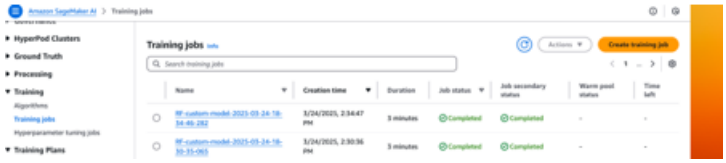
Fig. 5. Lex Output.


```
{
  "InstanceId": "i-9609253374",
  "InstanceType": "t3.small",
  "Metrics": {
    "CPUUtilization": 62.57,
    "DiskReadOps": 1421,
    "DiskWriteOps": 505,
    "NetworkIn": 42143,
    "NetworkOut": 29395
  },
  "DailyCost": "200.04"
},
{
  "InstanceId": "i-7959807241",
  "InstanceType": "t3.small",
  "Metrics": {
    "CPUUtilization": 20.28,
    "DiskReadOps": 831,
    "DiskWriteOps": 884,
    "NetworkIn": 47532,
    "NetworkOut": 34737
  },
```

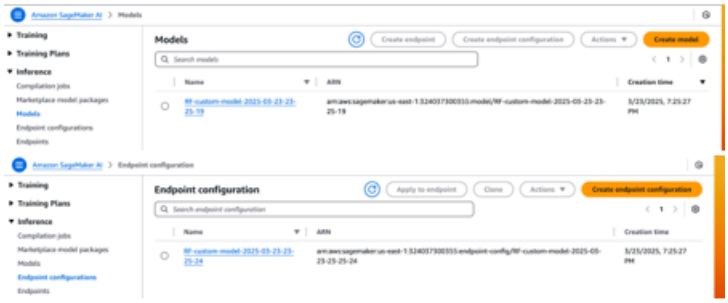

SageMaker:

We have trained the model using dummy data and deployed it on AWS SageMaker. SageMaker initiates a Training Job to train the model based on the specified parameters. In our case, we used the RandomForestClassifier model with four hyperparameters. To ensure optimal performance, we evaluated multiple models and found that the accuracy of this model closely aligned with our expectations.

During the training process, SageMaker automatically creates a Training Job in AWS, utilizing the provided hyperparameters. It fine-tunes the model by selecting the best parameter values to achieve improved accuracy and reliable predictions. Once the model is trained, we deploy this model to available for prediction for which we create the Model and endpoint which will be used by lambda for prediction.



Please find attached screenshot for your reference. **Invocation**



Lambda for SageMaker:

The Invocation Lambda function is used for parallel processing of requests and is deployed to invoke the SageMaker endpoint efficiently. The primary responsibility of this Lambda function is to extract the Instance ID from the incoming payload and retrieve relevant performance metrics from Amazon CloudWatch, such as CPU utilization, disk usage, network usage, and other key indicators. These retrieved parameters serve as input to the machine learning model, which then predicts whether the instances are underutilized or overutilized. This enables proactive resource optimization and helps in cost-efficient cloud infrastructure management.

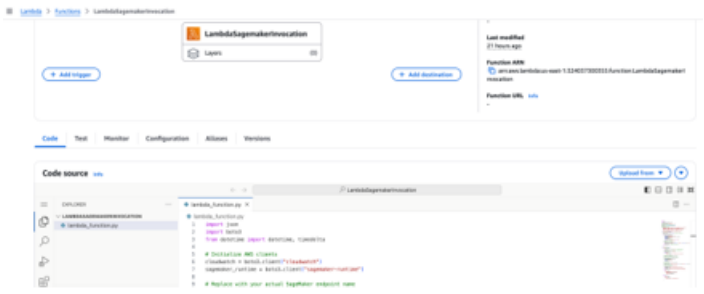


Fig. 6. Postman Request.

As we don't have multiple resources within the account, we had to create the dummy records and got the model trained. With SageMaker we are able to get the required result. Below is screenshot of the same.

- **Lex Slot Configuration:**
  a. servicename: Identifies the AWS service.
  b. daterange: Captures user-defined time frames (e.g., "between 2024-01-01 and 2024-03-01"). c. instanceid: Stores the instance identifier for size-related queries.
- **Lambda 1 (Preprocessing):**
  Ensures user input follows the correct format before reaching Lex.
- **Lambda 2 (Processing and Storage):**
  a. Extracts values from slots and stores structured messages in SQS.
  b. Uses regex to parse daterange and extract fromdate and todate.
- **Testing Considerations:**
  a. Used Postman for API testing, but it does not simulate conversation flow.
  b. Used Lex Test Console for direct chatbot testing.

VI. Roadmap

Planned activities for the remainder of the semester:

- Refine parallel processing capabilities
- Improve model prediction accuracy
- Develop robust training data pipeline
- Implement user-friendly chatbot interface
- Conduct comprehensive testing and validation
- Explore API polling or SNS notifications for request status updates

VII. Obstacles and Concerns

- Lex v2 limitation of one Lambda function per alias
- Complexity in slot extraction and date range handling
- Ensuring request ID persistence for status tracking
- Uncertainty in responding to API Gateway or Amazon Lex
- Feasibility of testing parallel processing

VIII. Expectation

Following are outcome which we are expected from the project.

- Scaling of lambdas used within the project and feasibility on handling multiple request and process messages and responses parallelly.
- Decouple the architecture to avoid failure of message loss.
- Allowing Amazon lex to handle multiple requests and allowing it scale as per requirement.
- We will be targeting to provide information regarding utilization of S3, lambdas and EC2's. This will have Cost explorer and CloudWatch specific matrix without any AI prediction.
- In another case if customer ask specific questions on right sizing, then we will provide prediction based on the recommendation from model written on sage maker.

IX. Lessons Learned

- Parallel processing in Lambda is challenging with large datasets

- Identifying correct prediction metrics requires careful analysis
- Generating training data introduces additional complexity
- Exploratory data analysis is crucial for model performance
- Preprocessing and model selection are critical steps

X. Backup Plan

- Direct output display in Lambda
- Store results in DynamoDB for retrieval
- Fallback to sequential processing if parallel processing proves challenging
- Develop a new Lex intent for checking request status

XI. Conclusion

Our Serverless AI Chatbot for Cloud Cost Optimization demonstrates promising potential in addressing complex cloud resource management challenges. By leveraging AWS services and machine learning, we aim to provide real-time, actionable insights for organizations seeking to optimize their cloud infrastructure.

References

[1] Amazon Web Services, "Amazon Lex: Build conversational interfaces for any application," AWS, [Online]. Available: https://aws.amazon.com/lex/. [Accessed: 29-Mar-2025].

[2] Amazon Web Services, "Amazon SageMaker: Build, train, and deploy machine learning models," AWS, [Online]. Available: https://aws.amazon.com/sagemaker/. [Accessed: 29-Mar-2025].

[3] Amazon Web Services, "AWS Lambda: Run code without thinking about servers," AWS, [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 29-Mar-2025].

[4] M. Richardson and M. J. J. J. P. V. Jansen, "Serverless Computing: Economic and Architectural Impact," in *Proceedings of the 2018 IEEE International Conference on Cloud Computing*, San Francisco, CA, USA, 2018, pp. 322-329. doi: 10.1109/CloudCom.2018.00057.

[5] R. S. O. McKinney, "Machine Learning for Cloud Cost Optimization: A Predictive Approach," *Journal of Cloud Computing*, vol. 6, no. 2, pp. 87-96, Apr. 2023.

[6] Amazon Web Services, "AWS Cost Explorer: Visualize and manage AWS spending," AWS, [Online]. Available: https://aws.amazon.com/aws-cost-management/aws-cost-explorer/. [Accessed: 29-Mar-2025].

[7] Amazon Web Services, "Amazon CloudWatch: Monitoring and observability for AWS cloud resources," AWS, [Online]. Available: https://aws.amazon.com/cloudwatch/. [Accessed: 29-Mar-2025].