An Industrial Oriented Mini Project (CS70PC) Report

on

BULK EMAIL MANAGEMENT USING SERVERLESS ARCHITECTURE

submitted

in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Mr. BAZARU PRIYATHAM SAI CHAND

(18261A0508)

Under the Guidance of

Dr. A. Nagesh

(Professor)

Department of Computer Science and Engineering

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

(Affiliated to Jawaharlal Nehru Technological University, Hyderabad)

Hyderabad 500075, Telangana (India).

2021-2022

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

(Affiliated to Jawaharlal Nehru Technological University, Hyderabad)

GANDIPET, HYDERABAD - 500 075. Telangana

CERTIFICATE

This is to certify that the thesis entitled **Bulk Email Management Using Serverless Architecture** is being submitted by **Bazaru Priyatham Sai Chand** bearing **Roll no. 18261A0508** in partial fulfillment for the award of **B.Tech** in **Computer Science and Engineering** to **Jawaharlal Nehru Technological University, Hyderabad** is a record of bonafide work carried out by him under my guidance and supervision.

The results embodied in this project have not been submitted to any other University or Institute for the award of any degree or diploma.

Project Guide
Dr. A. Nagesh
Professor

Head of Department
Dr. C.R.K Reddy
Professor

External Examiner

DECLARATION

This is to certify that the work reported in this project titled **"Bulk Email Management using Serverless Architecture"** is a record of work done by me in the **Department of Computer Science and Engineering, Mahatma Gandhi Institute of Technology, Hyderabad.**
No part of the work is copied from books/journals/internet and wherever the portion is taken, the same has been duly referred in the text. The report is based on the work done entirely by me and not copied from any other source.

B. PRIYATHAM SAI CHAND

(18261A0508)

ACKNOWLEDGEMENT

I would like to express my sincere thanks to **Dr.K.Jaya Shankar, Principal,MGIT**, for providing the working facilities in college.

I wish to express my sincere thanks and gratitude to **Dr.C.R.K Reddy, Professor and HOD,** Department of CSE, MGIT for all the timely support and valuable suggestions during the period of project.

I am extremely thankful to **Dr.B.Prasanthi, Sr Assistant Professor and Ms.D.Deepika, Assistant Professor,** Department of CSE, MGIT, mini project coordinators for their encouragement and support throughout the project.

I am extremely thankful and indebted to my internal guide **Dr. A. Nagesh, Professor**, Department of CSE, for his constant guidance, encouragement and moral support throughout the project.

Finally, I would also like to thank all the faculty and staff of CSE Department who helped me directly or indirectly, for completing this project.

B. PRIYATHAM SAI CHAND
(18261A0508)

TABLE OF CONTENTS

CERTIFICATE i

DECLARATION ii

ACKNOWLEDGEMENT iii

LIST OF FIGURES vi

LIST OF TABLES vii

ABSTRACT viii

1 INTRODUCTION 1

1.1 Overview 1

1.2 Different Kinds of Services 1

1.2.1 Lambda Function 1

1.2.2 Simple Email Service 1

1.2.3 Simple Notification Service 2

1.2.4 Simple Queue Service 2

1.2.5 API Gateway 2

1.2.6 CloudWatch 3

1.2.7 X-ray 3

1.3 Problem Definition 4

1.4 Proposed System 4

1.5 Requirements Specification 4

1.5.1 Software Requirements 4

1.5.2 Hardware Requirements 4

2 LITERATURE SURVEY 5

3 DESIGN METHODOLOGY OF BULK EMAIL MANAGEMENT SYSTEM 9

3.1 UML Diagrams 9

3.1.1 Use Case Diagram 10

3.2 Data Flow Diagram 10

3.3 Architecture diagram 11

4 IMPLEMENTATION 13

4.1 Frontend 13

4.2 Serverless Functions 15

4.2.1 email_batcher 15

4.2.2 sqs_mailer 15

4.2.3 bounce_mailer 15

4.2.4 bounce_db 16

4.3 API endpoints 16

4.3.1 Email sending 16

4.3.2 bounce sending 17

5 TESTING AND RESULTS 18

5.1 Performance 18

5.2 Error Handling 19

5.3 Cost 20

5.4 Results 21

5.4.1 Dashboard 21

6 CONCLUSION AND FUTURE SCOPE 22

6.1 Conclusion 22

6.2 Future Scope 22

BIBLIOGRAPHY 23

APPENDIX 25

LIST OF FIGURES

3.1 Use case Diagram 10

3.2 Data Flow Diagram of bulk email management system 11

3.3 AWS architecture of bulk email management system 12

4.1 Frontend Web Application 14

4.2 Output webpage 14

4.3 Email sending API request 17

5.1 Boxplot depicting the distrubtion of backend runtimes. 19

5.2 Different Error Messages 20

5.3 Statistics Dashboard of email services 21

LIST OF TABLES

2.1 Summary of literature survey 8

5.1 Time taken for different volumes of emails 21

ABSTRACT

Email has become one of the most important means of communications.Sending emails through conventional software like Gmail is limited to a mere thousands which is not sufficient to fit into the business needs and managing the insights of sent emails is crucial to companies to further enhance their approach. Serverless computing is any computing platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running [1].This reduces the time and knowledge required to deploy pieces of software while offering significant reduction in pricing compared to hosting the software on dedicated hardware.

Various cloud platforms such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform have adapted to this way of thinking making it easily accessible.A robust serverless architecture for sending bulk emails of the magnitude of thousands to millions using scalable model which is implemented using AWS lambda [2] and Simple Email Service(SES) for the SMTP protocol.

Challenges such as able to trace the bounce emails as well as the statistics for the emails sent. The prototype is built using the ReactJS framework along with underlying services from AWS. These tightly integrated services offer the best way to simulate the architecture and help in sending customized for every recipient based on email templates along with storing bounce emails. The effectiveness of the scalability offered by the architecture is analyzed using X-ray traces to simulate and test the bottlenecks and how to overcome them is discussed. With serverless technology, the cloud provider abstracts away the server management, provisioning servers with fine granularity, on demand, and with a pay-per-use model [3].

This project proposes an architecture for sending and management of the emails through serverless application using the AWS services and its implementation using a javascript based frontend application.

Keywords : Serverless,bounce emails,scalability.

1. INTRODUCTION

1.1 Overview

The bulk email management system is used to send emails of magnitude in thousands and more and broadcast the message to multiple email addresses with a single operation. All the required fields such as the senders' email addresses, subject, the html body and to addresses are selected and the sending is initiated with a button. Upon successful sending to the API endpoint the emails are processed and send through the AWS simple email services to all the users and the emails that were bounced are returned after the sending operation.It makes use of a Javascript based framework called ReactJS to input the fields and send it to the API hosted on the AWS service where an array of services are used to scale and perform the sending and retrieving of bounces.

In this project, the functions to send and manage emails using serverless functions is implemented and an application is created for usability.

1.2 Different Kinds of Services

The serverless functionalities utilized in this project are:

1.2.1 Lambda Function

AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. You can trigger Lambda from over 200 AWS services and software as a service (SaaS) applications, and only pay for what you use. [2]

1.2.2 Simple Email Service

Amazon Simple Email Service (SES) is a cost-effective, flexible, and scalable email service that enables developers to send mail from within any application. You can configure Amazon SES quickly to support several email use cases, including transactional, marketing, or mass email communications. Amazon SES's flexible IP deployment and email authentication options help drive higher deliverability and protect sender reputation, while sending analytics measure

the impact of each email. With Amazon SES, you can send email securely, globally, and at scale. [4]

1.2.3 Simple Notification Service

Amazon Simple Notification Service (Amazon SNS) is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication. The A2A pub/sub functionality provides topics for high-throughput, push-based, many-to-many messaging between distributed systems, microservices, and event-driven serverless applications. Using Amazon SNS topics, your publisher systems can fanout messages to a large number of subscriber systems, including Amazon SQS queues, AWS Lambda functions, HTTPS endpoints, and Amazon Kinesis Data Firehose, for parallel processing. The A2P functionality enables you to send messages to users at scale via SMS, mobile push, and email. [5]

1.2.4 Simple Queue Service

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. SQS eliminates the complexity and overhead associated with managing and operating message-oriented middleware, and empowers developers to focus on differentiating work. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent. We use the FIFO queue to process the emails to be sent exactly once and make sure duplication does not occur within the list of emails. [6]

1.2.5 API Gateway

Amazon Application Program Interface (API) Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services. Using API Gateway, you can create Representational State Transfer

(REST)ful APIs and WebSocket APIs that enable real-time two-way communication applications. API Gateway supports containerized and serverless workloads, as well as web applications. [7]

1.2.6 CloudWatch

Amazon CloudWatch is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), IT managers, and product owners. CloudWatch provides you with data and actionable insights to monitor your applications, respond to system-wide performance changes, and optimize resource utilization. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events. You get a unified view of operational health and gain complete visibility of your AWS resources, applications, and services running on AWS and on-premises. You can use CloudWatch to detect anomalous behavior in your environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep your applications running smoothly. [8]

1.2.7 X-ray

AWS X-Ray helps developers analyze and debug production, distributed applications, such as those built using a microservices architecture. With X-Ray, you can understand how your application and its underlying services are performing to identify and troubleshoot the root cause of performance issues and errors. X-Ray provides an end-to-end view of requests as they travel through your application, and shows a map of your application's underlying components. You can use X-Ray to analyze both applications in development and in production, from simple three-tier applications to complex microservices applications consisting of thousands of services. [9]

1.3 Problem Definition

Given a set of emails in the form of comma separated values with their respective subjects and email content send the respective emails to everyone on the list without provisioning any form of server and track the information about the sent emails such as the bounced emails, delivered, opened and bounce rate of the from address using a dashboard.

1.4 Proposed System

The proposed system comprises of an AWS system that can serve as a serverless backend that is invoked using a REST API request made from a javascript based frontend made using the library of ReactJS.

1.5 Requirements Specification

1.5.1 Software Requirements

1. Software : Any javascript enabled internet browser,Node.js 14.x .
2. Languages : Javascript

1.5.2 Hardware Requirements

1. Random Access Memory: 2 GB.
2. Storage: 500MB for internet browser.

2. LITERATURE SURVEY

In literature survey I have investigated various researches on this particular domain and some of them are as follows :-

1. **Serverless Applications: Why, When, and How?** [3]

Serverless computing shows good promise for efficiency and ease-of-use. Yet, there are only a few, scattered and sometimes conflicting reports on questions such as Why do so many companies adopt serverless?, When are serverless applications well suited?, and How are serverless applications currently implemented? To address these questions, we analyze 89 serverless applications from open-source projects, industrial sources, academic literature, and scientific computing—the most extensive study to date.

2. **Serverless Electronic Mail** [10]

We describe a simple approach to peer-to-peer electronic mail that would allow users of ordinary workstations and mobile devices to exchange messages without relying upon third-party mail server operators. Crucially, the system allows participants to establish and use multiple unlinked identities for communication with each other. The architecture leverages ordinary SMTP [11] for message delivery and Tor [12] for peer-to-peer communication. The design offers a robust, unintrusive method to use self-certifying Tor onion service names to bootstrap a web of trust based on public keys for end-to-end authentication and encryption, which in turn can be used to facilitate message delivery when the sender and recipient are not online simultaneously. We show how the system can interoperate with existing email systems and paradigms, allowing users to hold messages that others can retrieve via IMAP [13] or to operate as a relay between system participants and external email users. Finally, we show how it is possible to use a gossip protocol to implement mailing lists and how distributed ledger technology might be used to bootstrap consensus about shared knowledge among list members.

3. **Tracking Causal Order in AWS Lambda Applications** [14]

Serverless computing is a new cloud programming and deployment paradigm that is receiving wide-spread uptake. Serverless offerings such as Amazon Web Services (AWS) Lambda, Google Functions, and Azure Functions automatically execute simple functions uploaded by developers, in response to cloud-based event triggers. The serverless abstraction greatly simplifies integration of concurrency and parallelism into cloud applications, and enables deployment of scalable distributed systems and services at very low cost. Although a

significant first step, the serverless abstraction requires tools that software engineers can use to reason about, debug, and optimize their increasingly complex, asynchronous applications. Toward this end, we investigate the design and implementation of GammaRay, a cloud service that extracts causal dependencies across functions and through cloud services, without programmer intervention. We implement GammaRay for AWS Lambda and evaluate the overheads that it introduces for serverless micro-benchmarks and applications written in Python.

4. **Serverless Computing: Design, Implementation, and Performance** [15]

We present the design of a novel performance- oriented serverless computing platform implemented in .NET, deployed in Microsoft Azure, and utilizing Windows containers as function execution environments. Implementation challenges such as function scaling and container discovery, lifecycle, and reuse are discussed in detail. We propose metrics to evaluate the execution performance of serverless platforms and conduct tests on our prototype as well as AWS Lambda, Azure Functions, Google Cloud Functions, and IBM's deployment of Apache OpenWhisk. Our measurements show the prototype achieving greater throughput than other platforms at most concurrency levels, and we examine the scaling and instance expiration trends in the implementations. Additionally, we discuss the gaps and limitations in our current design, propose possible solutions, and highlight future research.

5. **Be wary of the economics of "Serverless" Cloud Computing** [16]

In standard cloud computing, dedicated hard- ware is replaced by dynamically allocated, pay-per- use resources, such as virtual servers. Although called "pay-per-use," these resources are typically billed based on allocation, not on actual use, potentially leading to a customer paying more than necessary. In "serverless," no resources are typically allocated or chargeable until a function is called. It's like the difference between a rental car and a taxi: you will be charged for the rental car even if you park it for a week, unlike a taxi. Moreover, some cloud providers are offering seemingly massive amounts of serverless computing at no charge. This holds the promise of the most efficient processing possible—for free or at least what seem to be attractive prices. Moreover, serverless fits with the modern approach to application construction—composing microservices rather than building hard-to-manage and scale monolithic applications. However, as with many things, the devil is in the details and the economic benefits of serverless computing heavily depend on the execution behavior and volumes of the application workloads. In the same way that pennies per day can add up to thousands of dollars eventually, low "per hit" prices can not only add up as transaction volumes increase, but

can make serverless economics unattractive relative to what have now become more traditional approaches,such as virtual machines or even dedicated hardware.

6. **Honorable Mention: Serverless architecture-a revolution in cloud computing** [17]

Emergence of cloud computing as the inevitable IT computing paradigm, the perception of the compute reference model and building of services has evolved into new dimensions. Serverless computing is an execution model in which the cloud service provider dynamically manages the allocation of compute resources of the server. The consumer is billed for the actual volume of resources consumed by them, instead paying for the pre-purchased units of compute capacity. This model evolved as a way to achieve optimum cost, minimum configuration overheads, and increases the application's ability to scale in the cloud. The prospective of te serverless compute model is well conceived by the major cloud service providers and reflected in the adoption of serverless computing paradigm. This review paper presents a comprehensive study on serverless computing architecture and also extends an experimentation of the working principle of serverless computing reference model adapted by AWS Lambda. The various research avenues in serverless computing are identified and presented.

S.no	Publish Year	Author	Title	Application	Advantage	Disadvantages
1.	2020	Simon Eismann et al.	**Serverless Applications: Why, When, and How?**	computing is isolated till it's completion	Describes case study for serverless backend	Only covers studies on adaption rather than success rate
2.	2020	Geoffrey Goodell	**Serverless Electronic Mail**	P2P network for email management	Gives better idea of SMTP in serverless networks	Considers a decentralized network based on Tor.
3.	2018	Wei-Tsung Lin et al.	**Tracking Causal Order in AWS Lambda Applications**	performance benchmarks for using lambda with various other services.	Detailed analysis on compatibility and what to expect upon combining.	Uses a niche tool called Gammaray which is superseded by X-ray traces.
4.	2017	McGrath et al.	**Serverless Computing: Design, Implementation, and Performance**	Compare and contrast various providers and approach needed	gives model architectures and design stragies	Provides solutions based on containers.
5.	2017	Adam Eivy	**Be wary of the economics of "Serverless" Cloud Computing**	Explains the cost attributed with this technology	In detail about Scalability vs Economics	Doesn't cover larger payload scenarios

Table 2.1: Summary of literature survey

3. DESIGN METHODOLOGY OF BULK EMAIL MANAGEMENT SYSTEM

3.1 UML Diagrams

The Unified Modelling Language (UML) is a standard language for writing software blueprints. UML can be used for:

- **Specifying**: It is a blueprint created by an architect prior to the construction.
- **Visualizing** : Visualizing is concerned with deep analysis of system to be constructed.
- **Constructing** : Modelling also provide us mechanism which are essential while constructing a system.
- **Documenting** : Finally, modelling justifies its importance by applying all its credentials to be bounded in a piece of paper referred as document.

The UML is a language which provides vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A modelling language is a language whose vocabulary and the rules focus on the conceptual and physical representation of a system. Modelling yields an understanding of a system.

Three types of diagrams are shown to propose the design and depict the methodology that is implemented by using the appropriate tools.

3.1.1 Use Case Diagram

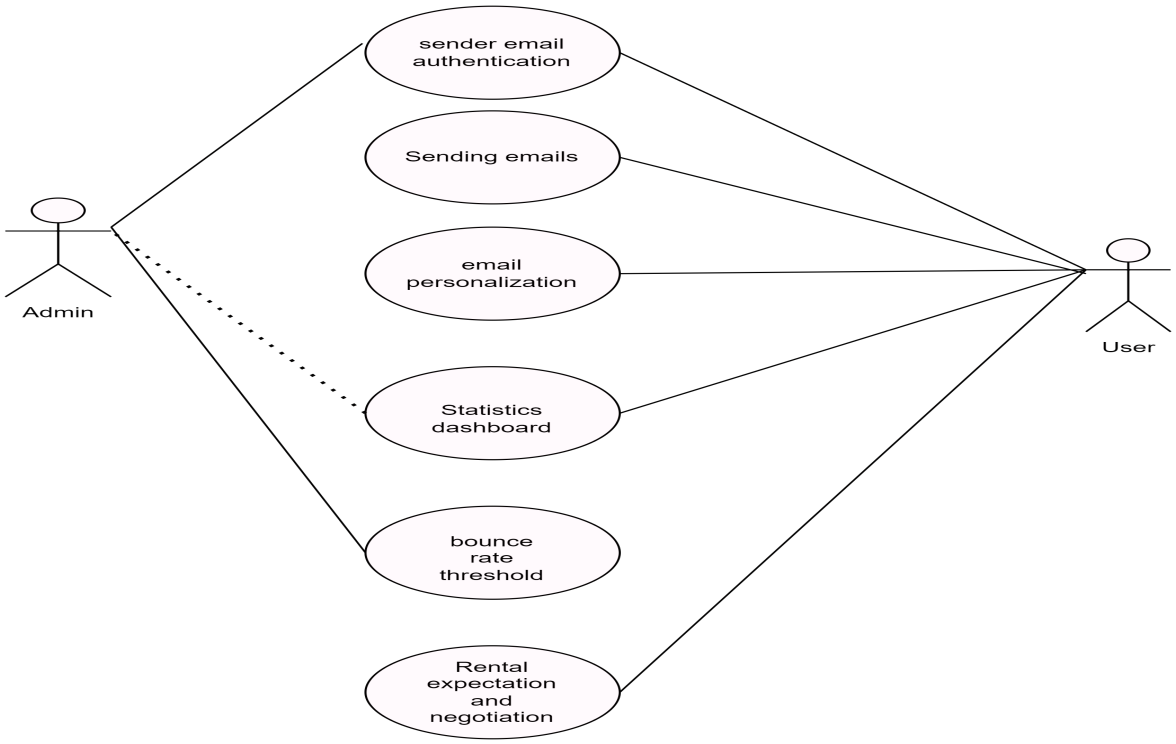

Figure 3.1: Use case Diagram

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So, when a system is analyzed together its functionalities use cases are prepared and actors are identified.

In Figure 3.1, the user gives in Input Program to the compiler and is able to view the outcomes of Syntax, Lexical analysis phases of the compilation process (compilation errors if any in the source code).

3.2 Data Flow Diagram

A data-flow diagram is a way of representing a flow of data through a process or a system (usually an information system). The DFD also provides information about the outputs and inputs of each entity and the process itself. A data-flow diagram has no control flow, there are no decision rules and no loops. Specific operations based on the data can be represented by a flowchart.

For each data flow, at least one of the endpoints (source and / or destination) must exist

in a process. The refined representation of a process can be done in another data-flow diagram, which subdivides this process into sub-processes.

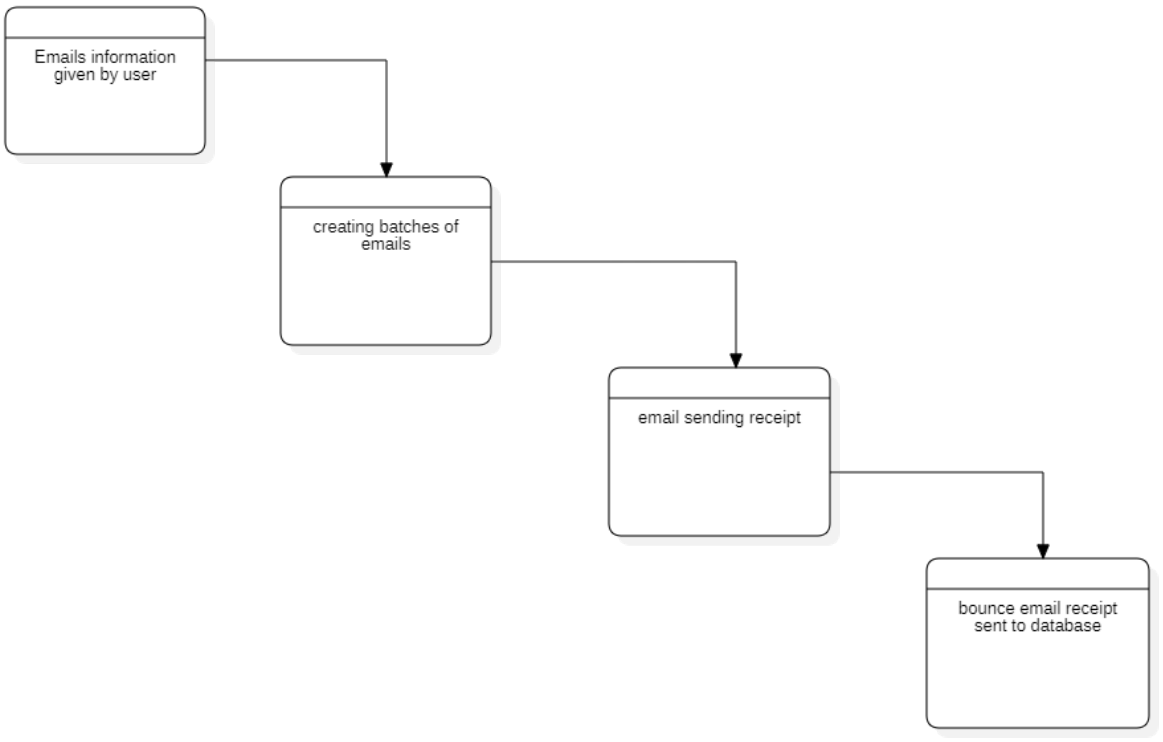

Figure 3.2: Data Flow Diagram of bulk email management system

The data flow diagram displayed in Figure 3.2 shows how the data that is entered by the user is processed till the sending of emails and further finding the bounce emails data and how they are stored finally in a database.

3.3 Architecture diagram

Architecture diagram shows how various components that exists in a network work together and are integrated with other services. User interaction to output the various methods and services used and their correlation is depicted in this diagram.

The components used can be instances of a component and the sequence in which they need to be connected to successfully achieve the desired output is shown here.

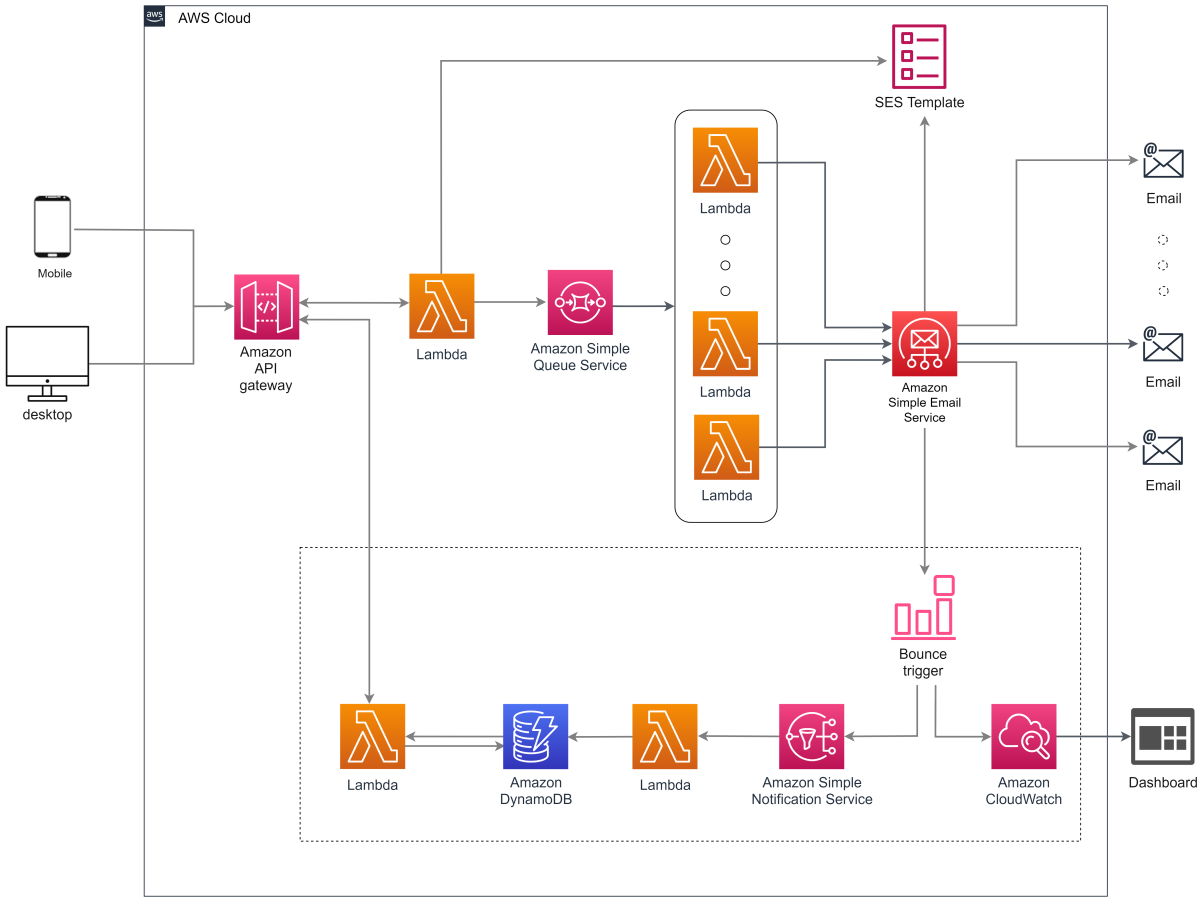

Figure 3.3: AWS architecture of bulk email management system

The data flow diagram displayed in Figure 3.3 shows how the data that is entered by the user is processed till the sending of emails and further finding the bounce emails data and how they are stored finally in a database.

The architecture is based a few constraints from the SES service and other bottlenecks that affect the sending of emails. The SES through which the emails are initially sent through sandbox to test and later needs to be upgraded by the AWS authority to permit the user to request increase in the number of emails to be sent. For the purpose of this architecture 50,000 emails have been requested which was granted by the authority.

- 14 emails/second
- 50,000 emails per day
- request data 5kb

4. IMPLEMENTATION

The architecture demonstrated in the Fig 1 with the main functionalities include sending the emails and tracking the bounces. Initially, all the data sent to the lambda using the API gateway which handles the request and response for the whole operation. Upon reaching the lambda, JavaScript Object Notation (JSON) objects from the string parameters preprocessed, the html part of the email along with the subject made into SES templates accessed by other lambdas and the emails batched no more than 50 emails per batch and placed in a Simple queue Service[5] queue. The ordering of the messages is important hence, we use a First In First Out queue, which triggers another lambda that sends emails to all the 50 recipients using a templated email function by accessing the template that we created earlier. We set a flag to mark the end of the batches deleting the template by the end of the execution.

4.1 Frontend

The web application is used to fill and send the emails to all the users at the same time. The application is built using ReactJS library and bootstrap library to style the web page. A basic layand intuitive layout is given considering the web application is mostly used as internal software within an organization or various organizations as a service. The fields that need to be filled are

1. **Emails** - Comma seperated values of the emails with spaces and that follows the standard email format.

2. **Names** - Comma separated values of the names corresponding to the emails above it is not a required field. The names will be made into empty strings in the preprocessing.

3. **Subject** - String value giving the subject. It is a mandatory field.

4. **html** - The email body that is sent to the users. It is interpreted as html code when sent to the lambda function.

5. **Sender's email** - The senders email needs to be verified by OTP with AWS console before sending the emails. After an email is verified it is added into the dropdown by the admin. Any one of the verified emails can be used.

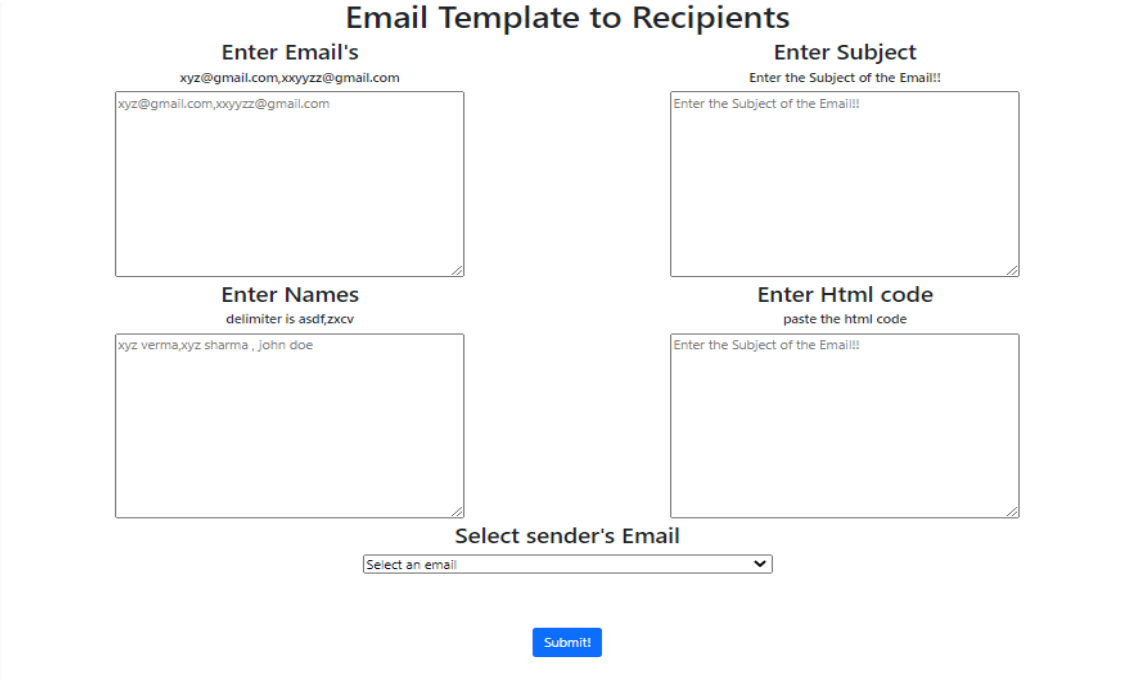

Figure 4.1: Frontend Web Application

In Figure 4.1 The web application made using the styling packages is shown. All the respective fields that were mentioned before can be filled out by the user and click on submit to initialize the preprocessing and sending of data to the serverless functions.

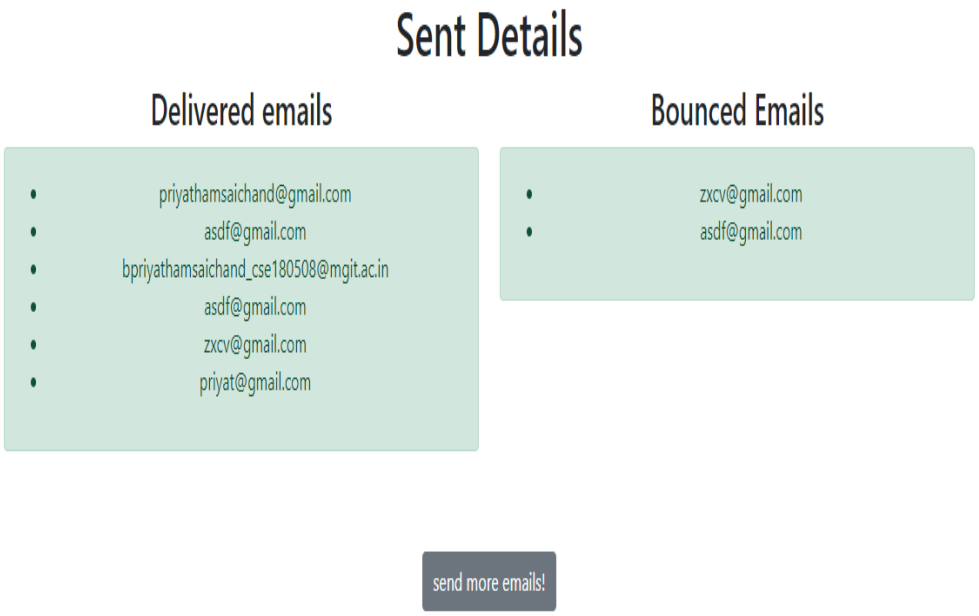

Figure 4.2: Output webpage

In Figure 4.2 The output webpage that is returned after the emails are sent is shown.

4.2 Serverless Functions

To implement the business logic that is present in the backend. Four lambda functions where created to be used to send and manage the bounces that are generated. Their functionalities are described as follows.

4.2.1 email_batcher

The purpose of this function is to receive the POST request payload with all the required information such as emails, names, sender's email and subject. It creates the email template with the HTML and sender's email. As per the constraints mentioned in design and methodology, upto 50 emails are batched into a single JSON object and sent into the FIFO queue. All the preprocessing of the string data to json objects is done in this function and is the first function executed with every API call. This lambda function is invoked as a trigger with API gateway used to handle the API endpoints of the application.

4.2.2 sqs_mailer

This function is used to send the emails obtained from the FIFO queue. It uses the template that was created by email_batcher and uses the SES service to send the emails to 50 email addresses along with their names using the sender's email addresses provided. It is triggered by the message in the queue and it can be scaled to meet the number of messages that are available in the queue. Same instance of the lambda function can be used after it is executed or seperate instances can be used based on the concurrency limit given to the lambda function which is 100 in this case.

4.2.3 bounce_mailer

After the emails are sent, The bounces are sent to the SNS service that triggers this function to collect and filter the json object that is provided and send the bounced email, timestamp to the Dynamo DB for every bounce that occurs.

4.2.4 bounce_db

Upon successful sending and return of the email_batcher function the application requests to get all the bounce emails that were accumulated in the database at the given moment using the timestamp and obtained bounce emails as a response to the POST request it sends using another API endpoint. After the bounced emails are retrieved from the database and stored in a JSON object they are cleared from the Database. For next bulk email sending.

4.3 API endpoints

In this section, we will discuss the details on how the REST API calls are made to the serverless backend and the endpoints that are used along with how the request and responses of the endpoints are configured. Two endpoints were used which are as follows

4.3.1 Email sending

We use an API gateway REST API that is connected to the lambda function of email_batcher. Thunderclient API testing extension is used with Visual Studio Code to test the API capabilities. The url of the endpoint is given by `https://slb37ny1bh.execute-api.ap-south-1.amazonaws.com/prod/email_batcher`.

The emails parameter is used to send the array of emails seperated by commas. The names parameter follows the same schema and the name of the person corresponds to the email provided and all the email addresses without a name at the end are defaulted to empty strings in the preprocessing of the Javascript Application.

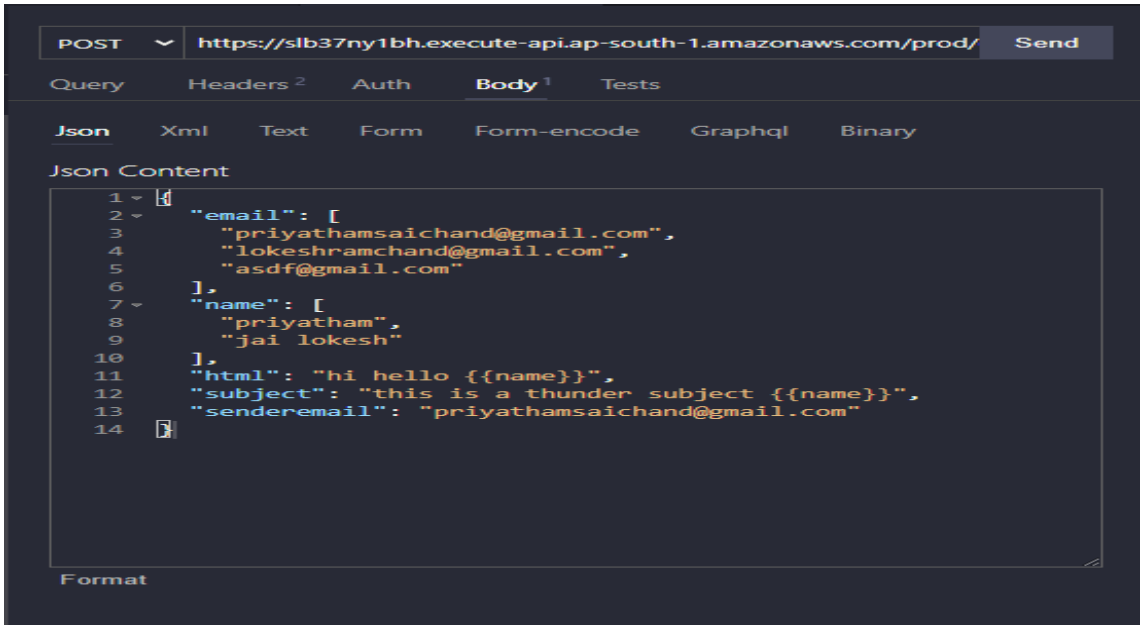

Figure 4.3: Email sending API request

In fig 4.3 we show the schema of the request API and all the parameters that needs to be sent.

4.3.2 bounce sending

We use this API to request the server for the bounce email records that were placed in Dynamo DB.We send a POST request with null parameters and receive the bounce emails. The API end point is given by.

`https://slb37ny1bh.execute-api.ap-south-1.amazonaws.com/prod/bounce_db`

The emails are returned in the form of array of emails and timestamps. Once the bounced emails arrive the delivered emails are computed using the sent emails from the sent email data and the difference of sent and bounced emails gives rise to delivered emails.

5. TESTING AND RESULTS

Basic message transfer from Author to Recipients is accomplished by using an asynchronous store-and-forward communication infrastructure in a sequence of independent transmissions through some number of mail transfer agents [18]. Two of the most common ways to analyze the architecture was to consider the time required to send all the emails and the cost associated with it. Both of the metrics are analysed in the following section.

5.1 Performance

There are two performance monitoring services available to AWS application developers: CloudWatch and X-ray. [14] The application is analyzed to see the time required to send the emails and the time taken to run the functions between the request and response to the API endpoint. This can be seen through the cloudwatch [8] logs and X-ray to find out the issues with latency over various regions and finding the best regions and how to optimize the functions.In this case, we used X-Ray as a monitoring and display service that automatically samples the entry and exit of function instances, called segments, using unique trace identifiers **`(trace_id)`** [14] [9].

Here we take the case of sending multiple emails and the time taken is sampled over 30 data points to represent in Figure 2. The time is calculated by considering the cloudwatch logs that can be used to determine the time taken to execute the last function of lambda as show in Figure 1 to send the email and the time leading to that function from the start of the API gateway request log that is stored in the logs. This experimented several times to cancel out the variations of cold start that is a common phenomenon in serverless functions. All the calculations are done to avoid any overlap between different invocations and only a single request is sent within the same region.

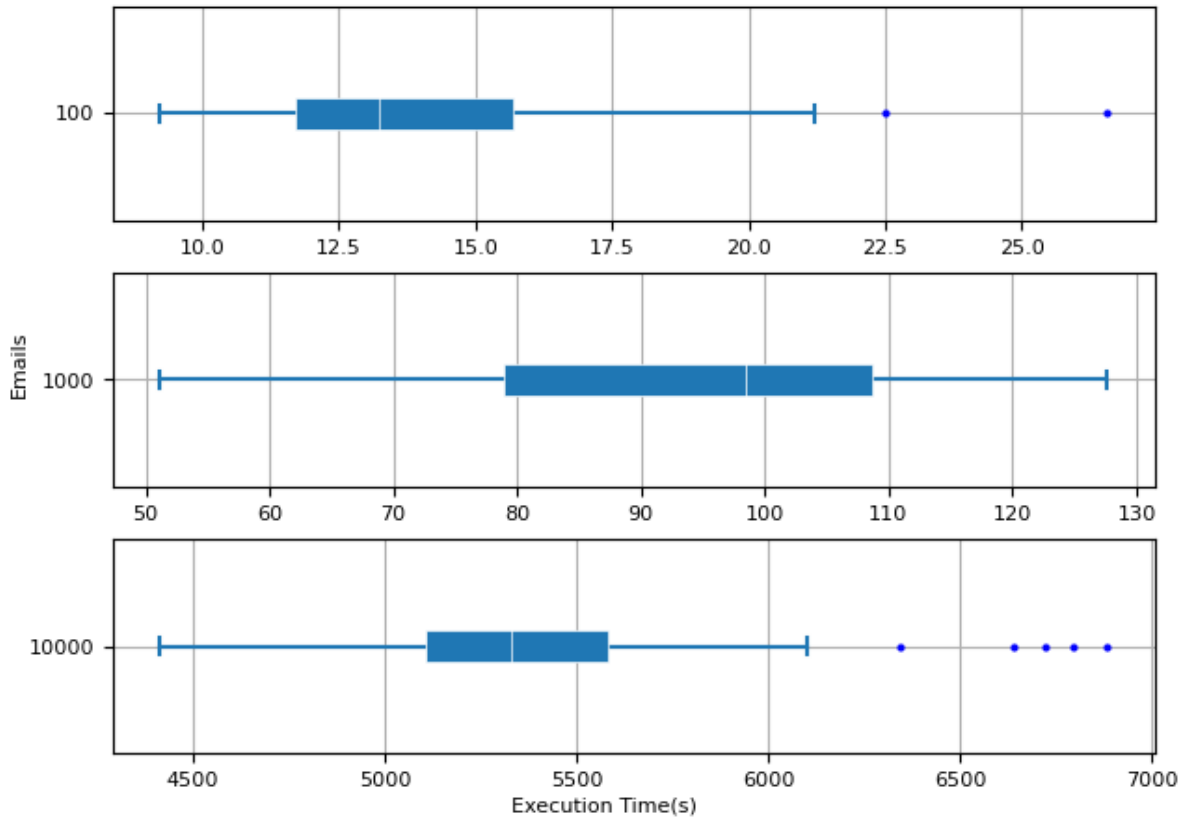

Figure 5.1: Boxplot depicting the distrubtion of backend runtimes.

In Figure 5.1 the graph depicts boxplot of runtimes of serverless backend functions after the api is invoked and when email is sent, the number of emails sent is directly proportional to the time taken. The time taken for 10,00 emails averaging at around 30 minutes is sufficient and satisfactory considering that the time taken is to derived from the last email that is sent in the list of emails.

5.2 Error Handling

Errors in the frontend of the web application is managed by notification pop-up windows that can signify corrections of problems in execution that have occurred.

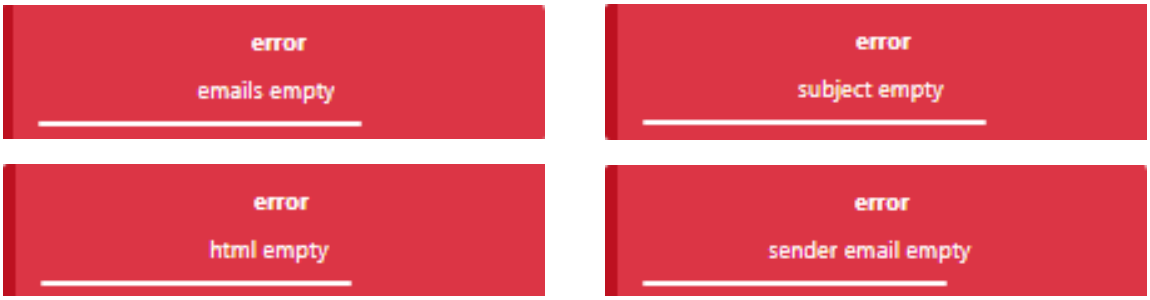

Figure 5.2: Different Error Messages

As shown in Figure 5.2 If the user leaves out important information that is required by the API. The application halts giving meaningful error messages in a pop-up box at the right corner of screen.

5.3 Cost

The cost of the services can also be a motivating factor to implement such an architecture. In our use case The following architecture can be tested with the AWS free tier and later expanded to the needs of the users. With $0.01 dollars per 1,000 emails but one needs to be wary about the associated services like the SES, SNS and the cloudwatch metrics to analyze the emails add up to cost but since they are calculated based on the GB of data it needs to pre-process the rates will not exceed the cost of emails.

Here we take the example, provided by the SES platform to estimate the cost required to implement the architecture.

You use Amazon SES to send about 250,000 emails per month. You receive 1,000 emails per month. You don't use dedicated IP addresses. Every message you send and receive is 32KB in size which results in a total of $25.98 per month [**?**] which is significantly less than competitors such as SendGrid or MailChimp who offer their own SMTP server or schedule emails to fit into the constraints of other providers to carry out email campaigns.

Amazon provides sample pricing calculations,but as your workload varies, so will the billing. [16] The SQS requests can exceed the free tier if not monitored carefully and add up to additional costs for the next 1000 requests or more based on the usage of the system.

5.4 Results

The bulk email application has successfully achieved its objective of sending emails in large volumes using the web interface that is provided passing all the test cases and error handling mechanisms. The customization that is proposed is also accomplished by the SES template.

Serial Number	Number of emails	Mean time taken in (ms)	Median time taken in (ms)
1	100	14.5	13.2
2	1,000	93.4	98.4
3	10,000	5511.5	5329.9

Table 5.1: Time taken for different volumes of emails

In Table 5.1 The time taken by different volumes of email such as 100,1000,1000 taken from the box plot in Figure 5.1 is used.

5.4.1 Dashboard

The analysis of metrics that are obtained from the sending of emails can be viewed in the form of a high level dashboard and can be shared for real time monitoring of the bounce rate and minimise it and track the success of the email campaign for a specific sending email address.

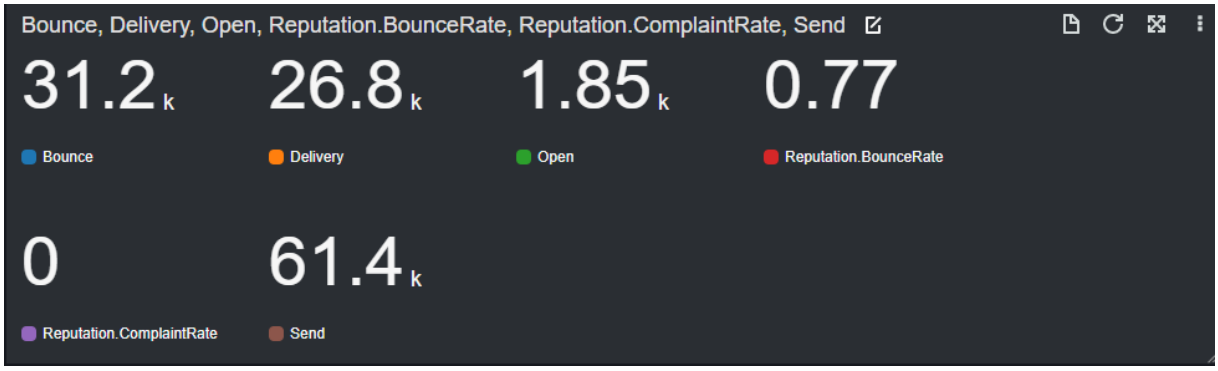

Figure 5.3: Statistics Dashboard of email services

As shown the Figure 5.3, The various mertrics such as open,sent,delivered,bounced are obtained and are updated in real time with a specific time interval. The duration and time can be changed using filtering options to analyse the campaigns effectively.

6. CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

To conclude, the architecture has been effective in sending bulk emails of various volumes with runtime that can be considered as sufficient or optimal and cost-effective. The primary goal of the architecture to be able to manage the entire system without a server being provisioned directly i.e. serverless is achieved and is promising to further develop and enhance its capabilities. The statistics displayed can be used for reach through the customers and the key idea of the architecture is to make use of serverless functions and services to manage emails with an optional fronted interface which was the case in this paper. Thus, making it a fast application to deliver emails to the users' inboxes in a short period of time. Without provisioning any IP addresses and acquiring the compute power needed to send the emails beforehand thereby making it more effective and possesses a higher level of abstraction that the conventional method.

6.2 Future Scope

The following architecture has immense scope to be integrated with various other services such as SMS service or can act as an independent backend logic to implement notficiation service to mobile application for the purposes of marketing or sharing information using other means of communication using the same logic with extensions.

Bibliography

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[2] "Aws lambda." Amazon Web Services, documentation, November 22 2021 [Online].

[3] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?," *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2020.

[4] "Amazon ses." Amazon Web Services, documentation, November 22 2021 [Online].

[5] "Amazon simple notification service." Amazon Web Services, documentation, November 22 2021 [Online].

[6] "Amazon simple queue service." Amazon Web Services, documentation, November 22 2021 [Online].

[7] "Amazon api gateway." Amazon Web Services, documentation, November 22 2021 [Online].

[8] "Amazon cloudwatch." Amazon Web Services, documentation, November 22 2021 [Online].

[9] "Amazon x-ray." Amazon Web Services, documentation, November 22 2021 [Online].

[10] G. Goodell, "Serverless electronic mail," *arXiv preprint arXiv:2007.04608*, 2020.

[11] P. Resnick, "Internet Message Format." RFC 5322, Oct. 2008.

[12] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," tech. rep., Naval Research Lab Washington DC, 2004.

[13] M. Crispin, "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1." RFC 3501, Mar. 2003.

[14] W.-T. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, and W. Xu, "Tracking causal order in aws lambda applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 50–60, IEEE, 2018.

[15] G. McGrath and P. R. Brenner, “Serverless computing: Design, implementation, and performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, IEEE, 2017.

[16] A. Eivy and J. Weinman, “Be wary of the economics of" serverless" cloud computing,” *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[17] R. A. P. Rajan, “Serverless architecture-a revolution in cloud computing,” in *2018 Tenth International Conference on Advanced Computing (ICoAC)*, pp. 88–93, IEEE, 2018.

[18] D. Crocker, “Internet Mail Architecture.” RFC 5598, July 2009.

APPENDIX

email_batcher

```
   var AWS = require('aws-sdk');
AWS.config.update({region: 'ap-south-1'});
var sqs = new AWS.SQS();
var ses = new AWS.SES();
var chunk_size = 50;
exports.handler = async (event,context) => {
// inputs from the request
    var emails = event['email']
    var names = event['name']
    console.log("received emails: ", emails)

    const params1 = {
  Template: {
    TemplateName: "template",
    HtmlPart: event['html'],
    SubjectPart: event['subject'],
    TextPart: event['html'],
  },
};

try{
var temp =  await ses.createTemplate(params1).promise();
}
catch(error){
console.log("create template ", temp)

    await ses.deleteTemplate({TemplateName: 'template' }).promise();
    await ses.createTemplate(params1).promise();

}

    var emails_count = Object.keys(emails).length;
    var names_count = Object.keys(names).length;
    if(emails_count < names_count)
        return { statusCode: 400, body: 'Error: names are more than emails'
```

```
            };
    else{

        var diff = emails_count-names_count;
        for(var i = 0; i < diff; i++){
            names.push('');
        }
    }
// batching the emails
    var final = [];
    var counter = 0;
    var portion = {
        "current_batch": (counter + 1),
        "emails": [],
        "names": [],
        "html": event.html,
        "sender_email": event.senderemail,
        "end_of_batches": false,
    };

     emails.forEach((email,index) => {
         const name = names[index]
         if (counter !== 0 && counter % chunk_size === 0) {
        final.push(portion);
     portion = {
        "current_batch": (counter/2) + 1,
        "emails": [],
        "names": [],
        "html": event.html,
        "sender_email": event.senderemail,
       "end_of_batches": false
    };
      }
      portion['emails'].push(email)
      portion['names'].push(name)
      counter++;
     })
     portion['end_of_batches'] = true
     final.push(portion);
```

```
    // sending the batches onto the queue
   var temp = {};
  for (var batch in final){
      var params = {
          MessageGroupId: batch,
    MessageBody: JSON.stringify(final[batch]),
    MessageDeduplicationId: context.getRemainingTimeInMillis().toString()
        ,
    QueueUrl: "https://sqs.ap-south-1.amazonaws.com/285448632456/
       email_batch.fifo"
  };
     temp = await sqs.sendMessage(params).promise();

  }

  return "email sending started";
```

bounce_emailer

```
var AWS = require('aws-sdk');
AWS.config.update({region: 'ap-south-1'});
const docClient = new AWS.DynamoDB.DocumentClient();
var sqs = new AWS.SQS();
exports.handler = async (event) => {
   var dest,type;
    event['Records'].forEach((value,index) =>{
        var message = JSON.parse(value['Sns']['Message'])
        console.log("message: ", message)
        console.log("event type ", message['eventType'])
        dest = message['mail']['destination']
        type = message['bounce']['bounceType']
        console.log("destinations: ", dest)

    })

   var datetime = new Date();
   const params = {
```

```
  TableName : 'emails',
  /* Item properties will depend on your application concerns */
  Item: {
     email: JSON.stringify(dest),
     date: datetime.toISOString(),
     type: type
  }
}
  // run SQL command
   try {
    var temp = await docClient.put(params).promise();
    console.log("success ",temp);
  } catch (err) {
    return err;
  }
    return "function done";
};
```

bounce_db

```
var AWS = require('aws-sdk');
AWS.config.update({region: 'ap-south-1'});
const docClient = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {

// retrieve records from db
    const getAllRecords = async (table) => {
  let params = {
    TableName: table,
  };
  let items = [];
  let data = await docClient.scan(params).promise();
  items = [...items, ...data.Items];
  while (typeof data.LastEvaluatedKey != "undefined") {
    params.ExclusiveStartKey = data.LastEvaluatedKey;
    data = await docClient.scan(params).promise();
    items = [...items, ...data.Items];
```

```
  }
  return items;
};
const deleteItem = async (table, id) => {
  var params = {
    TableName: table,
    Key: {
      email: id,
    },
  };


    let data = await docClient.delete(params).promise();
    return data;
};

var allRecords,allDelete;
// delete records from db
try {
    const tableName = "emails";
    // scan and get all items
     allRecords = await getAllRecords(tableName);
    console.log("records db ", allRecords)
    // delete one by one
    for (const item of allRecords) {
      allDelete = await deleteItem(tableName, item.email);
    }
  } catch (e) {
      console.log(e)
  }

  return allRecords;
};
```