Email Service Management System as a Microservice

[1]T Venu Madhav, [2]Ganashree K C
[1]UG Student, [2]Assistant Professor
[1]Department of Computer Science and Engineering,
[1]RV College of Engineering, Bangalore, India

Abstract: **Many businesses have adopted web-based services due the ever-growing demand for online services. The services are generally intended to service several hundreds of thousands of requests per day. This can be brought about by implementing the services as microservices. Email service is one such service that every business requires to keep in touch with the customers. In this paper, an email service management system is developed and deployed as a microservice. The microservice is implemented in Golang and Apache Kafka is used as the message streaming platform. The implementation mentioned in the paper is highly scalable and performance efficient. The evaluation metric for a microservice is the number of requests handled per given time duration. This paper deals with the advantages of microservice approach as opposed to the trivial monolithic design.**

IndexTerms **-Microservice, Monolith, Golang, Apache Kafka, Email Service**

I. INTRODUCTION

With the advent of digitization and the scale of consumer demands, many businesses have strived to adapt to the changes by increasing their product and service availability to the potential consumers. The advantage is that these businesses need not have expensive resources unlike that of the older methods and practices. The digital reachability is a scalable factor which can be achieved using the underlying principles of the Internet. A business which caters to a certain niche of consumer base can use the technology stack available as open source without caring about hard expenses.

A successful business is built on the trust of the consumer base, which is implicit when the business meets the ever growing demands. This basically implies that the business must be reliable to the customers in order to build the trust amongst them. The business needs to cater to the customers when required to make an impact on the trust that the consumers have.

Thus catering such a nearly flawless experience, a very stable and intuitive service infrastructure needs to be put in place irrespective of the niche or use cases. Satisfaction of the customer was the primary goal for decades, but with the increase in the quality demands, it is safe to say that the customer happiness needs to be focused on. The services need not just fit to the customer needs but it must enhance the experience of the customer so that they have a positive feedback in relation to other competitors. To have a reliable and easily available service environment , the prime factor of consideration dwells on scalability.

A scalable architecture needs to have a distributed approach to request handling and it needs to handle concurrency very efficiently. Cloud computing can be leveraged to meet for the needs and demands of data storage, computing power, ease of access. Concurrency can be met by utilizing the technology stack for a light-weight distributed architecture that handles each type service individually. This ensures isolation and hence, prevents a complete service hindrance on any incident of down time.

II. MOTIVATION

Over the last few years, it was observed that the request inflow for several large service providing businesses had proliferated exponentially due to the surge of Internet and affordable rates of Internet access. One in every three people in the country have access to the internet. To handle the potential service demand , it is important to consider a nearly flawless service architecture to cater to the same.
With the advent of open source tools and documentations available throughout the internet, it has been easier than ever to start a business with lucrative opportunities.

III. LITERATURE SURVEY

The application of a distributed cloud computing based architecture requires a set of very lightweight, powerful tools to leverage bandwidth requirements and concurrency while managing consumer requests.

In [1], a study was conducted on tech stack requirements for the intended performance requirements. It was observed that Golang had the optimized benefits when utilized in building a distributed architecture. Golang is a light weight, super-fast programming language developed by Google to fulfil the requirements of faster and concurrent execution in the distributed systems.

In [2], different techniques were studied on communication among different components of a service or different services. The message passing technique is utilized as it is more efficient than a shared memory, which defeats the purpose of a distributed architecture. It was inferred that platform consumers like Email Service, Channel Partners can be set as consumer group. We can then configure the Kafka broker clusters to pertain to the needs of different types of request pulls from each of many consumer groups. So the study implied the benefits of Apache Kafka as an ideal event streaming service.

The study [3] showed the advantages of the use of hosting services when dealing with distributed microservice architectures. Amazon AWS Elastic-cache two was found to be the most useful due to extensive documentation given by AWS and various configurable technologies that compatible with the same.

The study conducted in explored idea of microservices as opposed to the monolithic architectures. The was observed that microservices[4] can be dealt with easily for code refactoring , duplications and migrations. Microservices are used by major service oriented companies like Netflix, Uber, Airbnb etc.

IV. SYSTEM DESIGN

The problem[5][6] is addressed through three modules which need to be setup and integrated together for the functionality of the final product. The Kafka module addresses the issue of scalability of incoming requests from the producer instances Kafka is set up with the email service as its consumer. Kafka ensures scalability through concurrent functionality.
The Email Service module is the immediate successor in the flow, it handles the email information & reports from both Kafka and Amazon modules. The email service module is built for concurrency that can leverage the same from the prior module.
The final module that drives the system is the Amazon module which have sophisticated setup and maintainability standards to cater to the issue of scalability and concurrency. The Amazon module sends and receives and processes the email data directly from the email clients.
The three modules are setup and then integrated continuously via agile industrial practices.

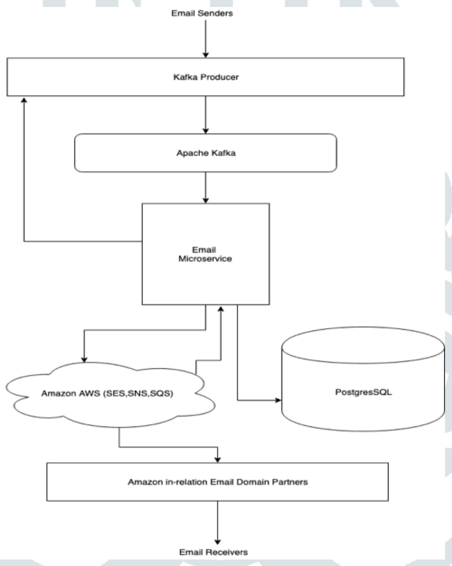

Figure 4.1 functional modules in the microservice

4.1 Kafka Module

This is the first module of the system that is used for a distributed approach for message streaming. The purpose of this module is to provide an event streaming service for sending emails to the microservice from the producer end. The email data created by the user at the Producer end. The email data received by the consumer group, the email microservice. This module is a large scale decentralized messaging system which is used to provide seamless, redundant messaging service.

4.2 Microservice Module

This is the second module of the system[7] is used as a Kafka consumer to facilitate a highly-scalable architecture that can provide up to thousands of incoming request at a time using the Kafka or RabbitMQ streaming service. The email service is deployed on an AWS EC2 cluster with a help of docker[8]. The purpose of this module is to provide a centralized email processing and management service. The email data represented as a JSON format from the Kafka module. The email data represented as a HTML/text MIME format. The email JSON values from the producer is parsed and processed using the inhouse APIs, open source software to create a valid email format to be sent.

4.3 Amazon Module

This is the last module of the system consists amazon software development kits with Golang interface used extensively to provide for a very fast and an efficient transaction rate due to the availability of concurrency in go in server side scripting. The

AWS module is used to send the emails to the respective clients. The MIME type text/HTML format emails. The JSON reports generated by Amazon SES based on the status of the email to be sent. The prime functionality of the AWS module is to send the emails to the partnered email clients like Gmail, Apple, Yahoo etc. The emails once sent, are tracked using notification service(SNS) implementation for corresponding emails for status information responses using the JSON format.

V. IMPLEMENTATION

The Kafka module consists of Kafka Brokers and Zookeeper network which is used as communication medium among the Kafka Brokers. The email data are stored under the 'emails' topic and have offset values representing each individual email data. The email data is pulled by the email service consumer group.

The Zookeeper network server is used to update and provide the offset details to the consumer groups upon each request pull. The email service module consists of several inhouse Golang packages for every amazon service leveraged.

The SES package is used for sending the email to the clients using the incoming the email data. The SQS package is used to update the email sent statuses and delivery report parsing and storing in the particular database. The SES package is used to store the data into the database.

The AWS module consists of SES, SNS, SQS module which are responsible for sending , handling and storing the incoming emails from the email microservice. The emails and attachments table is changed by the SES package while the recipients table is affected by the SQS package. The design flow is depicted in figure 5.1.

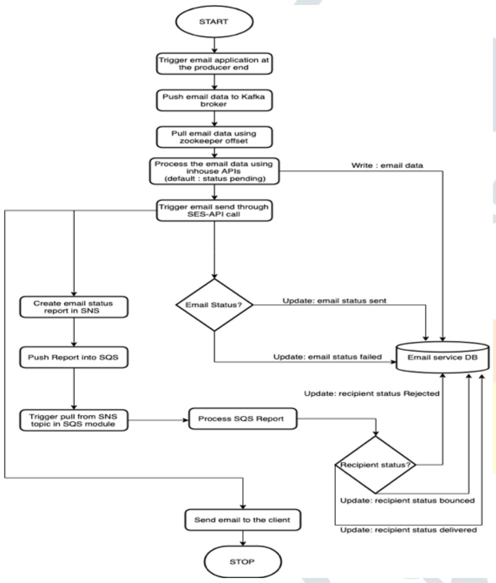
START
Trigger email application at the producer end
Push email data to Kafka broker
Pull email data using zookeeper offset
Process the email data using inhouse APIs (default : status pending)
Write : email data
Trigger email send through SES-API call
Create email status report in SNS
Push Report into SQS
Trigger pull from SNS topic in SQS module
Email Status?
Update: email status sent
Update: email status failed
Email service DB
Update: recipient status Rejected
Process SQS Report
Recipient status?
Update: recipient status bounced
Update: recipient status delivered
Send email to the client
STOP

Figure 5.1 flow chart for microservice implementation

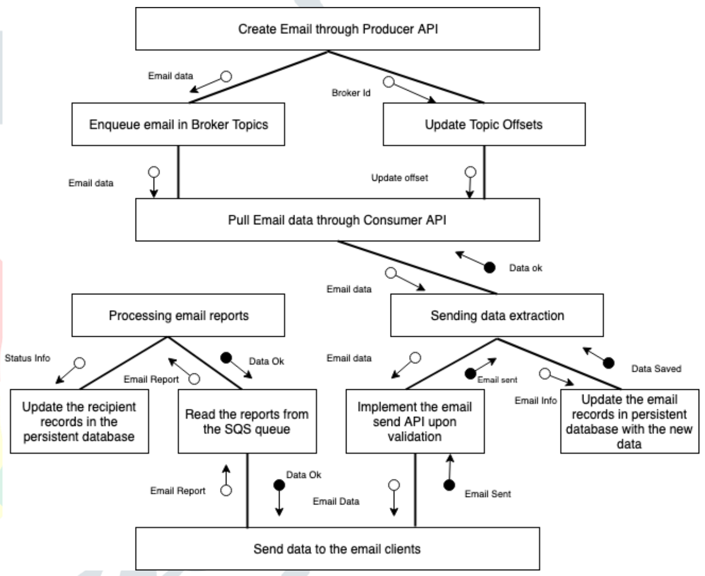
Create Email through Producer API
Email data
Broker Id
Enqueue email in Broker Topics
Update Topic Offsets
Email data
Update offset
Pull Email data through Consumer API
Data ok
Email data
Processing email reports
Sending data extraction
Status Info
Email Report
Data Ok
Email data
Email sent
Email Info
Data Saved
Update the recipient records in the persistent database
Read the reports from the SQS queue
Implement the email send API upon validation
Update the email records in persistent database with the new data
Email Report
Data Ok
Email Data
Email Sent
Send data to the email clients

Figure 5.2 structure chart for the email service

The structure chart depicted in the Figure 5.2, shows the end to end control flow of the email service[9] which starts from the creation of emails in the Kafka producer.

The email is sent or pushed into the Kafka message streaming platform. Kafka is often handled using Zookeeper, which is a networking system among the various Kafka Brokers. Zookeeper provides a very crucial functionality for a reliable data streaming functionality. The corresponding data in the brokers are identified and read from using the offset values. These offset values are updated and maintained by the Zookeeper.

The data is then pulled from the consumer group, which is the email microservice. The data is processed into the email format and sent to the Amazon SES module while being saved in the database module. The Amazon SES is responsible for sending the email to the respective customers. The module then creates reports for each email sent and returns them back to the email microservice.

These reports are categorized and processed and the corresponding recipient tables are updated based on the email sent status in the reports. The reports can be either delivered, bounced, rejected and pending based on the sent status. The module also tracks if the emails are opened by the receiver. This process is done to scale up to several thousand emails per day.

VI. ANALYSIS AND RESULTS

The performance of a microservice is analysed based on the amount of requests successfully tended to in a given particular amount of standard time duration. A microservice as a product or service is created to mitigate the process of handling complex functionality in a monolithic system. The microservice[10] thus is generally expected to reduce the build and deployment time for any developed functionality. Therefore, the build time is also considered as an evaluation metric.

1. The first evaluation metric is the number of requests handled by the microservice per a given duration of time. The graph depicted in Figure 6.1 provides the number of requests, email processing requests handled per day.

2. The second evaluation metric is the build time for the microservice and the improvement with respect to monolithic counterpart.

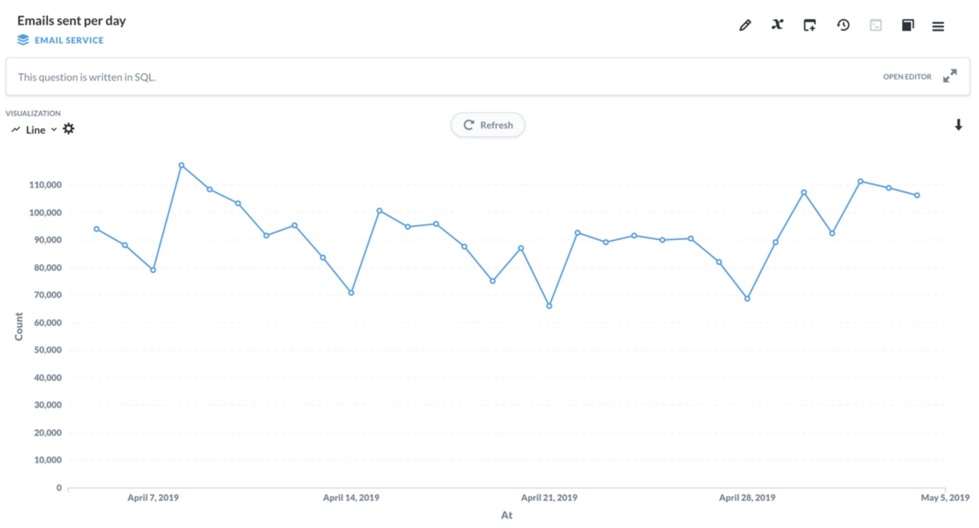

Figure 6.1 email requests handled per day

The graphical analysis shows that the microservice provides **1.15740 requests/second**.

Figure 6.2 build time for the microservice

The build time for microservice is very important from performance point of view. The microservice architecture is designed for reduction of test and build time for the system modules. The average build time of 'Email-service' is **2 minutes** (9 x faster) than monolithic design. The resources used are 4 GB RAM and Dual Core CPU. The Figure 6.2 shows the Build time for the system.

VII. CONCLUSION

The project to develop and deploy the email service management system as a microservice was intended to provide a seamless and efficient way to handle email services between the users and their customers. The project met its intended requirements successfully as more than 100,000 users avail this service to transact with their corresponding customers.

The functional requirements of having to handle thousands of requests gracefully was met due to the implementation of Golang routines and by the use of concurrency[11] tools like channel and select statements.

The system is currently put to production and is maintained by the operations team to scale up and down with Kafka integrations depending on the demands of the merchants and customers.

REFERENCES

[1] Westrup, Erik & Pettersson, Fredrik. (2014). Using the Go Programming Language in Practice. ResearchGate. Department of Computer Science, Lund University. 5(3): pp. 10–66.
[2] Philippe Dobbelaere & Kyumars Sheykh Esmaili.(2017). Industry Paper: Kafka versus RabbitMQ. ResearchGate, 10.11(3): pp. 227-238.
[3] Mahesh K ,Yogesh Kumar Sharma, (2019).A comparative study on google app engine amazon web services and microsoft windows azure. IAEME publication, computer science department, CMR engineering college. 3 (20): pp. 54-60.
[4] Zimmermann, Olaf. 2017. Microservices tenets: Agile approach to service development and deployment. Computer Science - Research and Development. Jagadish Prasad University, Rajasthan, pp. 60-64

[5] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In Present and Ulterior Software Engineering, (Springer International Publishing), pp. 195–216.
[6] Larrucea, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018). Microservices. IEEE Software *35*, pp. 96–100.
[7] Singh, V., and Peddoju, S.K. (2017). Container-based microservice architecture for cloud applications. In Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2017, (Institute of Electrical and Electronics Engineers Inc.), pp. 847–852.
[8] Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. IEEE Software *33*, pp. 42–52.
[9] Di Francesco, P., Malavolta, I., and Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017, (Institute of Electrical and Electronics Engineers Inc.), pp. 21–30.
[10] Sill, A. (2016). The Design and Architecture of Microservices. IEEE Cloud Computing *3*, pp. 76–80.
[11] Bavdys, M. (2018). Golang Multithreading. Proceedings of X International Scientific and Practical Conference "Electronics and Information Technologies.", pp. 59-64.