NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM
INFORMATION AND DATA MANAGEMENT

DISSERTATION THESIS

Micro-Service-Based Referrals on AWS

Kyriaki Aikaterini I. Ganoti

Advisor: **Alex Delis,** Professor

ATHENS

DECEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΔΙΑΧΕΙΡΙΣΗ ΠΛΗΡΟΦΟΡΙΑΣ ΚΑΙ ΔΕΔΟΜΕΝΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Micro-Service-Based Referrals on AWS

Κυριακή Αικατερίνη Ι. Γανωτή

Επιβλέπων: **Αλέξης Δελής,** Καθηγητής

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2020

DISSERTATION THESIS

Micro-Service-Based Referrals on AWS

Kyriaki Aikaterini I. Ganoti
R.N.: M1465

ADVISOR: **Alex Delis,** Professor

December 2020

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Micro-Service-Based Referrals on AWS

Κυριακή Αικατερίνη Ι. Γανωτή
Α.Μ.: M1465

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής

Δεκέμβριος 2020

ABSTRACT

On this project we developed a microservice which provides a structured way of collecting and organizing referrals. The main idea behind a referral is the ability of individuals or other business entities to refer customers to a business, in return for a kind of compensation, such as commissions on resulting sales. Typically, customers refer people who they believe will be benefited from a particular service. So, the customer tells their friends about the business and the business gains new customers. This project is implemented in Java 8 [1], uses the MySQL relational database management system managed by Amazon Aurora database engine, it integrates Amazon's tools like SQS (Simple Queue Service) and SNS (Simple Notification Service) and it is being fully deployed on Amazon's EC2 instances (Elastic Compute Cloud). The main implementation includes a single microservice called referrals, which is responsible for any referral related business logic, as for example, the tracking of all new referrals which happen on the ecosystem along with the progress of each of them. Each referral is being created under a specific referral scheme and for each referral scheme we define properties, like what are the conditions a referral needs to have met in order to be considered as completed, what the status of a referral is, what kind of reward a referral can get after completion etc. The service defines an API where it exposes CRUD endpoints and can accept requests by other services or clients over JSON-RPC protocol [2]. It also holds listeners on Amazon's queues for getting information about event-messages which have been published by other services or clients.

SUBJECT AREA: Software Development
KEYWORDS: AWS, database, entity, aurora, ec2, sqs, sns, datadog

ΠΕΡΙΛΗΨΗ

Στην παρούσα εργασία υλοποιήσαμε ένα microservice το οποίο παρέχει έναν δομημένο τρόπο συλλογής και οργάνωσης συστάσεων. Η κύρια ιδέα πίσω από μια σύσταση είναι η ικανότητα ατόμων ή άλλων επιχειρηματικών οντοτήτων να παραπέμπουν πελάτες σε μια επιχείρηση, σε αντάλλαγμα για ένα είδος αποζημίωσης, όπως προμήθειες για τις προκύπτουσες πωλήσεις. Συνήθως, οι πελάτες αναφέρονται σε άτομα που πιστεύουν ότι θα επωφεληθούν από μια συγκεκριμένη υπηρεσία. Έτσι, ο πελάτης λέει στους φίλους του για την επιχείρηση και η επιχείρηση κερδίζει νέους πελάτες. Αυτή η εργασία υλοποιήθηκε σε Java 8 [1], χρησιμοποιεί το σχετικό σύστημα διαχείρισης βάσεων δεδομένων MySQL το οποίο διαχειρίζεται η μηχανή βάσεων δεδομένων Amazon Aurora, ενσωματώνει τα εργαλεία της Amazon όπως SQS (Simple Queue Service) και SNS (Simple Notification Service) και εκτελείται σε EC2 της Amazon (Elastic Compute Cloud). Η κύρια εφαρμογή περιλαμβάνει ένα microservice, το οποίο ονομάζεται referrals και το οποίο είναι υπεύθυνο για οποιαδήποτε λογική που σχετίζεται με συστάσεις, όπως για παράδειγμα, η παρακολούθηση όλων των νέων συστάσεων που συμβαίνουν στο οικοσύστημα μαζί με την πρόοδο καθεμιάς από αυτές. Κάθε σύσταση δημιουργείται κάτω από ένα συγκεκριμένο σχήμα συστάσεων και για κάθε σχήμα συστάσεων ορίζουμε ιδιότητες, όπως ποιες είναι οι προϋποθέσεις που πρέπει να πληροί μια σύσταση για να θεωρηθεί ως ολοκληρωμένη, ποια είναι η κατάσταση μιας σύστασης, τι είδους ανταμοιβή μπορεί να λάβει ένας χρήστης για μια σύσταση η οποία ολοκληρώθηκε κ.λπ. Η υπηρεσία καθορίζει ένα API όπου εκθέτει λειτουργίες CRUD και μπορεί να δεχτεί αιτήματα από άλλες υπηρεσίες ή πελάτες μέσω του πρωτοκόλλου JSON-RPC [2]. Διατηρεί επίσης ακροατές σε Amazon queues για τη λήψη πληροφοριών σχετικά με μηνύματα εκδηλώσεων που έχουν δημοσιευτεί από άλλες υπηρεσίες ή πελάτες.

ΠΕΔΙΟ ΕΝΔΙΑΦΕΡΟΝΤΟΣ: Software Development
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: AWS, database, entity, aurora, ec2, sqs, sns, datadog

To mom and dad

ACKNOWLEDGMENTS

During the writing of this dissertation, I have received a great support, so I would like to thank my supervisor and professor Mr. Alex Delis whose knowledge and mentorship were invaluable, and who has always been a great motivation for me.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Referral marketing

People influence people. Referral marketing is known to be the best marketing approach applied by small and large businesses around the world. This type of marketing helps the increase of the number of new clients and is usually done by offering rewards encouraging customers to recommend the business's product and services to other people. Before the development of the service started, we proved that the referrals service would indeed help to bring new clients to the business. For achieving that, we made available the referral process on a specific percentage of our existing users and we monitored through statistical analysis their behaviour. From this analysis, known as AB Testing in digital marketing, we observed that the referral process helped indeed, to bring many new users into the business.

1.1.1 Referral process

During the referral process, an existing user has the ability to refer one or more friends, who have never signed up into the platform before. The referrer provides the email addresses of whoever they are willing to invite into the business and by just clicking a submit button, they send an invitation email to those addresses. The email for each of the addresses is unique. It contains a link, which will carry information about the referrer and the email address and once the recipient of the email clicks on this link, they will be led in the platform's sign up page. That way, when a new user signs up into the platform, the tracking of information about the referee becomes very easy and we should not forget that this kind of knowledge is required to be stored later on referrals service.

2. DATABASE DESIGN

2.1 Referrals database

Referrals microservice is written in Java 8. It uses Google Guice framework to automate the dependency injection and stores the required data in a mysql database. The basic concept needs to be defined first, is the referral scheme. A new referral is being created under an active referral scheme, which defines information related to how long a referral is valid for completion before to expire, what conditions need to be fulfilled in order for a referral to be considered as completed, what the reward for the referee will be if their referrals manage to complete their referral process, what the reward for a new referred user will be etc. The database design consists of six tables for storing all the actions related to what we described above and one more table, which is a task manager information table. We will explain more about the task manager information table later. The main six tables are the **reward, referral_scheme, referral, condition_table, referral_condition** and **referral_scheme_condition_table** and the relationships from one to another could be observed on Figure 1, where the ER diagram of the database is presented.

2.2 Database schema

In more details, the **referral_scheme** defines the required properties of a referral scheme such as its name, if it's active or historic, for how long it can be active etc and also holds two foreign keys on the **reward** table. The **condition_table** defines the available conditions which can be attached on one or more referral schemes (many to many relationship M:M). For this reason the table **referral_scheme_condition_table** defines which conditions have been attached on each, currently active or past active, referral scheme. The **referral_condition** table also holds a many to many relationship (M:M) between the **condition_table** and the **referral** table, because we need to know which conditions have been met for a referral. The database schema will look like the following:

```
CREATE DATABASE IF NOT EXISTS `referrals` DEFAULT CHARACTER SET
utf8mb4 COLLATE utf8mb4_unicode_ci;
USE `referrals`;


CREATE TABLE IF NOT EXISTS `reward` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(20) NOT NULL,
 `description` varchar(255) DEFAULT NULL,
 `amount` decimal(12,2) NOT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;


CREATE TABLE IF NOT EXISTS `referral_scheme` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `product` varchar(20) NOT NULL,
 `expiry_period_type` varchar(20) NOT NULL,
```

```
 `expiry_period_length` INTEGER NOT NULL,
 `limit_rewards`  INTEGER NOT NULL,
 `recruit_reward_id` bigint(20) unsigned NOT NULL,
 `referrer_reward_id` bigint(20) unsigned NOT NULL,
 `active` tinyint(1) NOT NULL,
 `activated_at` timestamp NOT NULL,
 PRIMARY KEY (`id`),
 CONSTRAINT `FK_referral_scheme_recruitreward` FOREIGN KEY
(`recruit_reward_id`) REFERENCES `reward` (`id`),
 CONSTRAINT `FK_referral_scheme_referrerreward` FOREIGN KEY
(`referrer_reward_id`) REFERENCES `reward` (`id`),
 INDEX `referral_scheme_active_idx` (`active`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;


CREATE TABLE IF NOT EXISTS `condition_table` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(20) NOT NULL,
 `description` varchar(255) DEFAULT NULL,
 `event_type` varchar(20) NOT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `name` (`name`),
 UNIQUE KEY `event_type` (`event_type`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;


CREATE TABLE IF NOT EXISTS `referral_scheme_condition_table` (
 `referral_scheme_id` bigint(20) unsigned NOT NULL,
 `condition_id` bigint(20) unsigned NOT NULL,
 `ordinal` int unsigned NOT NULL,
 `data` text NOT NULL,
 PRIMARY KEY (`referral_scheme_id`,`condition_id`),
 CONSTRAINT `FK_referralschemeconditiontable_referralscheme`
FOREIGN KEY (`referral_scheme_id`) REFERENCES `referral_scheme`
(`id`),
 CONSTRAINT `FK_referralschemeconditiontable_conditiontable`
FOREIGN KEY (`condition_id`) REFERENCES `condition_table` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;


CREATE TABLE IF NOT EXISTS `referral` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `referrer_user_id` bigint(20) unsigned NOT NULL,
 `user_id` bigint(20) unsigned NOT NULL,
 `created_at` timestamp NOT NULL,
 `referral_scheme_id` bigint(20) unsigned NOT NULL,
 PRIMARY KEY (`id`),
 CONSTRAINT ReferralScheme_User UNIQUE (`referral_scheme_id`,
`user_id`),
```

```
 CONSTRAINT `FK_referral_referralscheme` FOREIGN KEY
(`referral_scheme_id`) REFERENCES `referral_scheme` (`id`),
 INDEX `referrer_user_id_idx` (`referrer_user_id`),
 INDEX `user_id_idx` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

CREATE TABLE IF NOT EXISTS `referral_condition` (
 `referral_id` bigint(20) unsigned NOT NULL,
 `condition_id` bigint(20) unsigned NOT NULL,
 `created_at` timestamp NOT NULL,
 PRIMARY KEY (`referral_id`, `condition_id`),
 CONSTRAINT `FK_referralcondition_referral` FOREIGN KEY
(`referral_id`) REFERENCES `referral` (`id`),
 CONSTRAINT `FK_referralcondition_condition` FOREIGN KEY
(`condition_id`) REFERENCES `condition_table` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;
```

2.3 Entity Mapping

For being able to map Java classes to database tables and Java data types to SQL data types, we used hibernate framework[3]. Hibernate is a powerful object-relational mapping tool which also provides data query and retrieval facilities. In Java, the objects in a relational database context are defined as entities. In order to define an entity we need to create a class that is annotated with the @Entity annotation. The @Entity annotation is a marker annotation and is used to discover persistent entities. For our database design, we created six classes annotated with the @Entity annotation, for our main six tables and one more for the **task_manager** table. For example, the entity class for the referral scheme will look like the following:

```
@Entity
@Table(name = "referral_scheme")
public class ReferralSchemeEntity {
    /* properties, setters, getters and whatever else is needed
    it should live here */
}
```

It defines that the ReferralSchemeEntity will be mapped on the **referral_scheme** table, as the @Table annotation declares. The same logic applies for the rest of the entities.

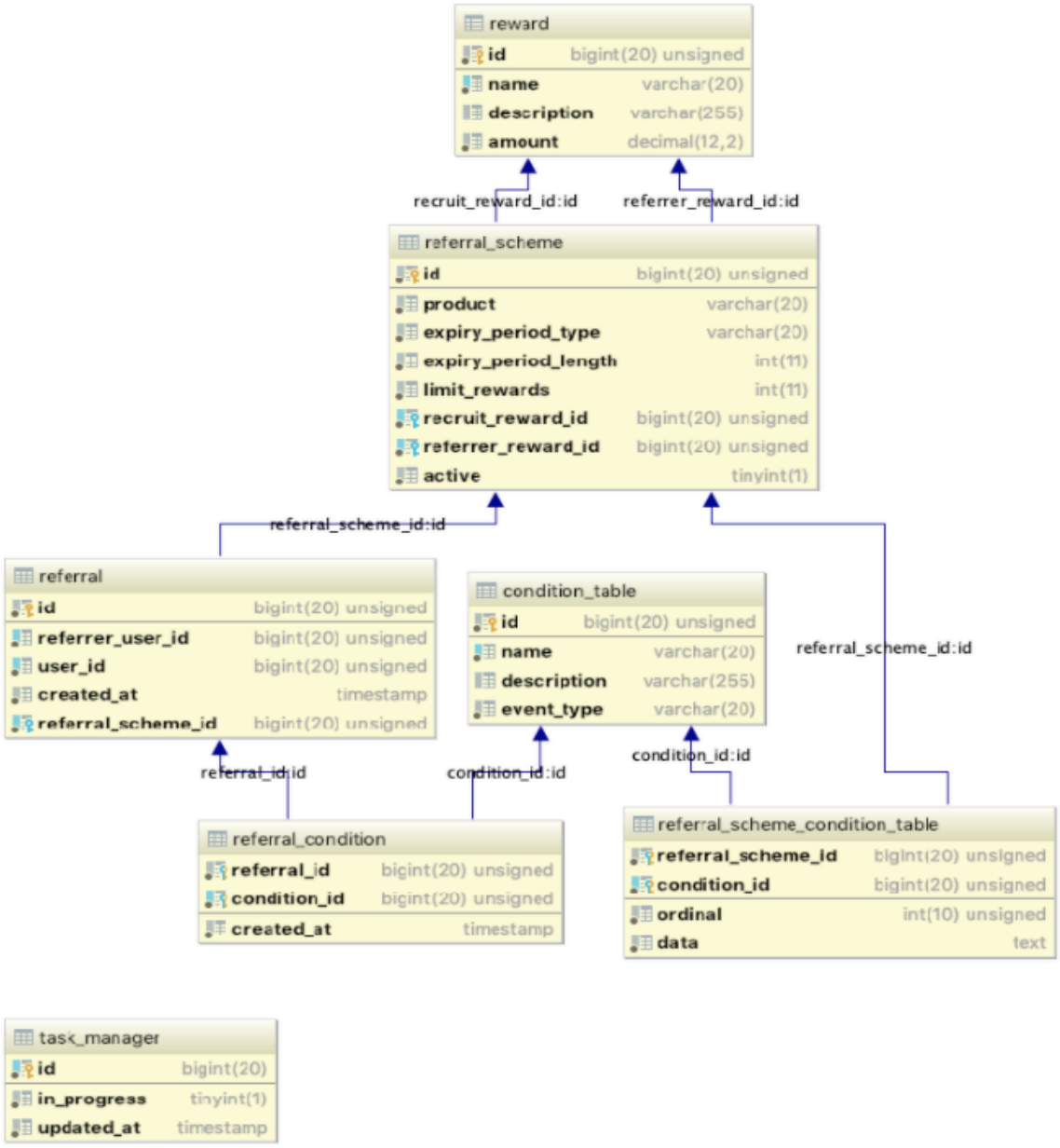

Figure 1: ER Diagram of referrals database design

2.4 Readers Writer Pattern

As it was mentioned earlier, we used Amazon's Aurora relational database engine in order to take advantage of its high-performance storage subsystem. The underlying storage can grow automatically as needed and also the database clustering and replication happens automatically as well. One or more DB instances are forming an Aurora DB cluster and its two main types are the *Primary DB instance* and the *Replica*. The Primary DB Instance or Master, supports reads and writes operations to the database and the Replica or Slave, supports only read operations. Replica instances can and should be more than one in order to support high read of rps (reads per second) but the Master instance should be always one. (Figure 2)

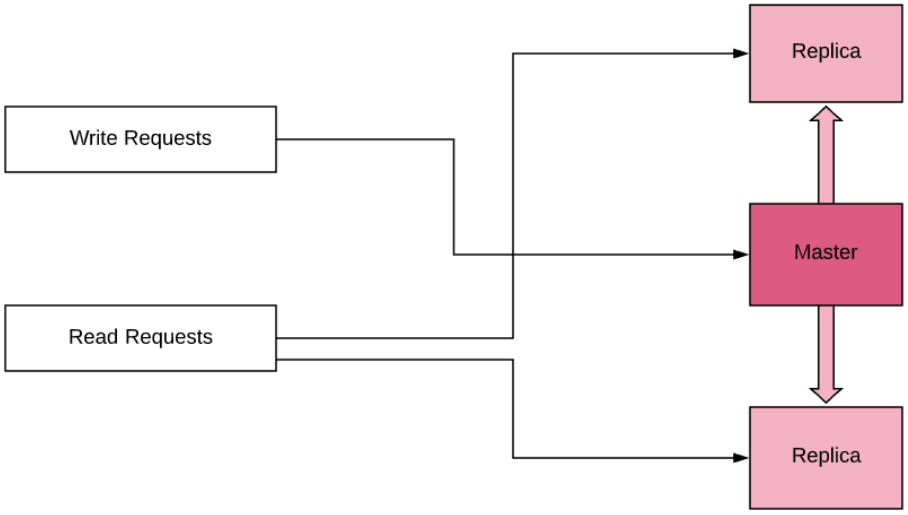

Figure 2: Readers Writer Scheme

3. AMAZON AURORA

For setting up an Amazon Aurora cluster we need to create an AWS account [4] and then login into the AWS console [5]. Once we login into the AWS cli we should see a page like Figure 3.

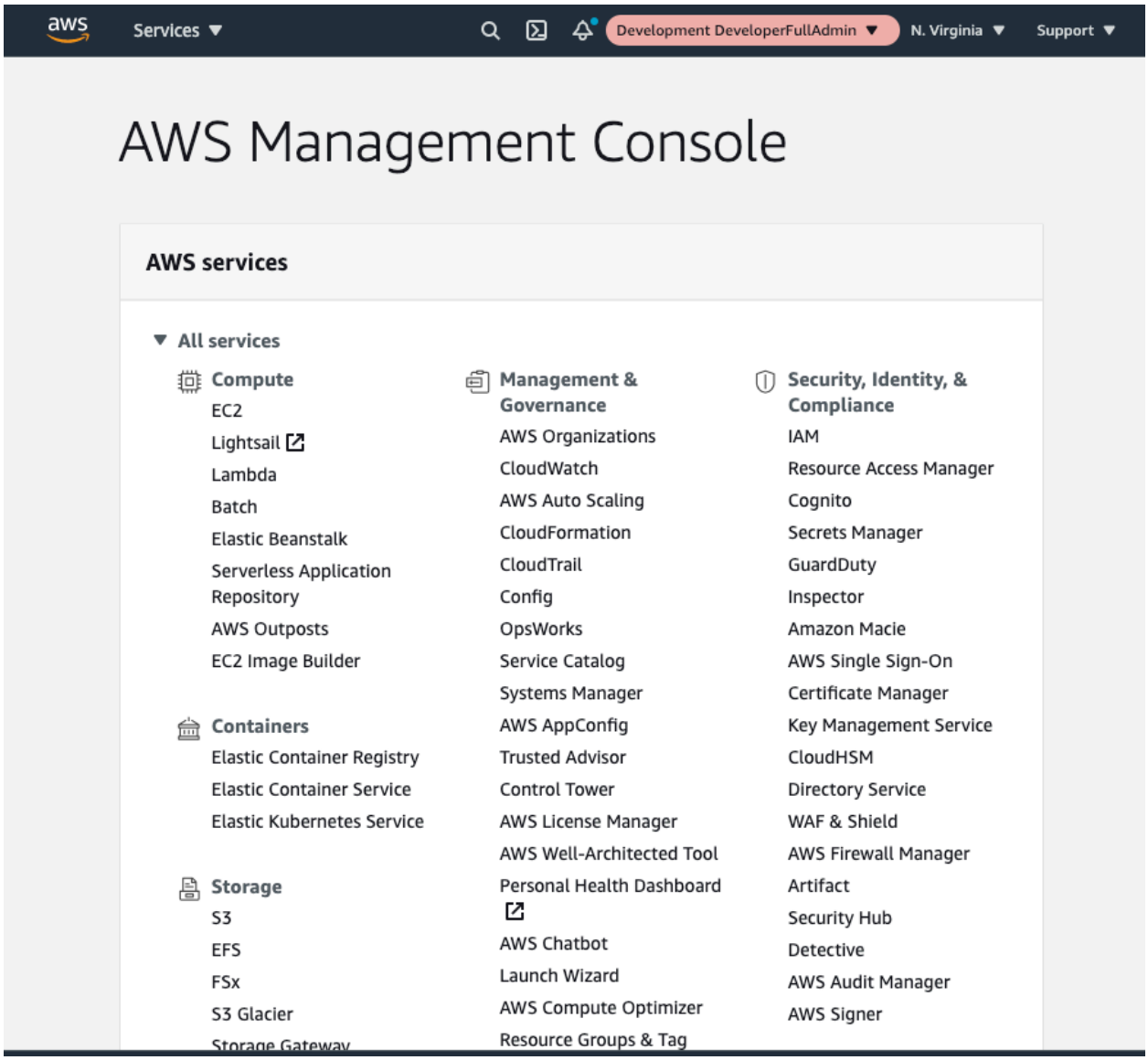

Figure 3: AWS Cli Login page

From the list of the available services, we will choose the Databases and we will filter the databases for referrals database. In Figure 4 we can see the referrals cluster we created. For creating a database, Amazon's documentation provides a good start [6].

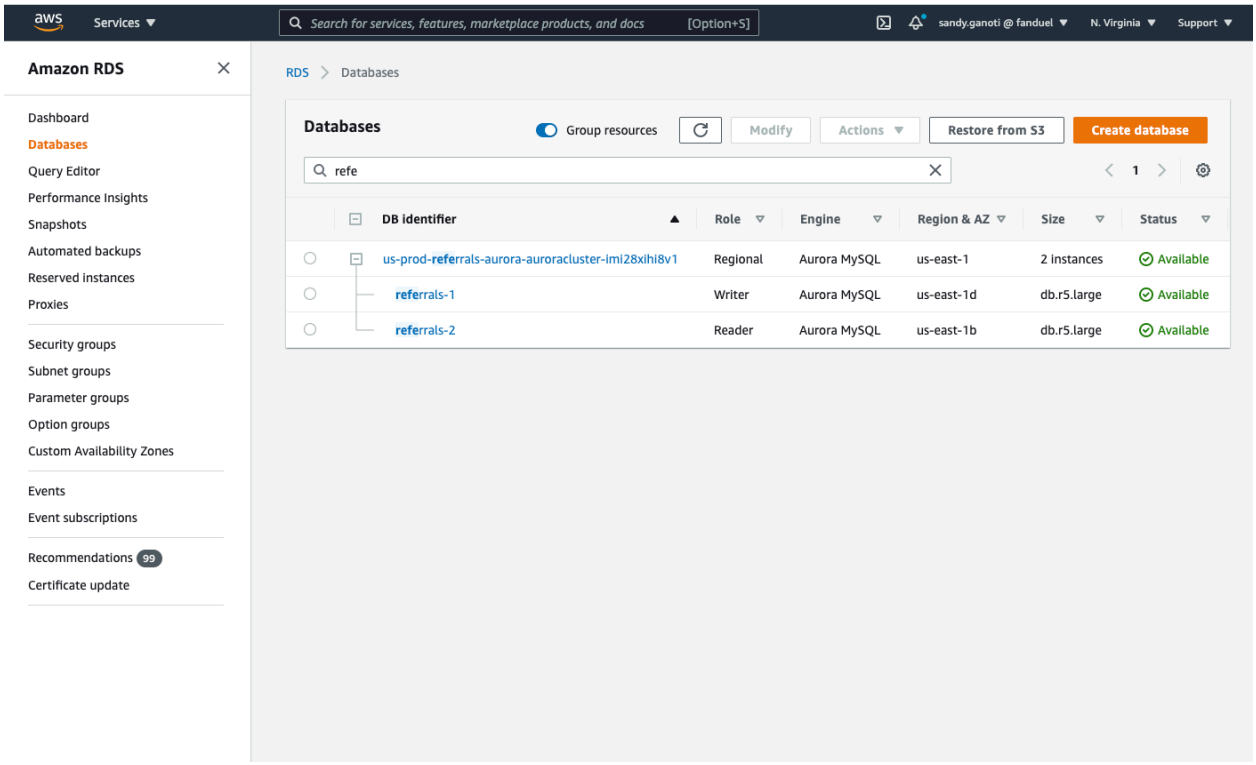

Figure 4: Database page

As we can see on the above figure, at the moment we got the screenshot, Aurora cluster for referrals maintains one Master and one Replica instance. In periods where the reads will be more often, for example on a period where a campaign runs and more and more referral actions are happening, the Replica instances might be increased.

4. OVERALL ARCHITECTURE

4.1 Referrals microservice

Referral microservice is responsible to define the active referral schemes, the conditions need to be fulfilled in order for a referral to be completed, any new referral and any old referral along with the status of each referral. The service exposes any required information through endpoints over json-rpc protocol. Those endpoints are mostly read operations, in the favour of exposing information like the status of the referral for a user, the current active referral scheme etc. Some write operations are exposed as well but are only related to admin actions, for example for setting up a new referral scheme, a new condition or a new reward. However, the data that the service needs for its main business logic around the track and the fulfilment of a referral, are not reaching the service through its endpoints. Data are collected and stored into the referrals database in two ways. The first is through event-message queues and the second is via polling another microservice. In more details, the referral process is designed currently as follows. There are three conditions attached to a current active referral scheme. Once these three conditions have been met, it's safe to be assumed that the referral of the user is completed.

4.1.1 Referral scheme and referral conditions

The first condition is related to the user signup event. Each time a user creation happens, there is a microservice which is safe to be called user service, which handles the creation of users. Subsequently, it publishes an event-message on an AWS SNS topic. This event-message contains valuable information about the user who just completed the signup process. On the other hand, referrals service creates an SQS Queue which subscribes on this SNS topic. Thus, each time an event-message is being published on the SNS topic, the referrals service gets notified and is able to check if the signup event was a referral. In the case that wasn't not a referral, it removes the message from the queue and does nothing more. Although, in the case that the new user was a referral, it creates a new entry on the **referral** table and one entry on the **referral_condition** table inside a transaction and then, it removes the message from the queue. Of course, the messages which are currently in processing, are being labeled as in progress, so we can prevent other instances of the application service to consume the same messages twice.

The second condition is related to a user KYC verification process [7]. We will not get into details on how a user is being verified but, what we need to know here is that there is a microservice, which is safe to be called verify service, which handles the verification process for each new user. Each time a user's verification process is being completed, an event-message it's being published from the verification microservice to an SNS topic. Of course, the user verification SNS topic it's quite different from the user signup SNS topic. Similarly with the signup event-message, the referrals service creates another SQS queue which subscribes on this verification topic and gets notified for each new event-message. After getting notified, the second condition for this user is being completed and a new entry on the **referral_condition** table is being added.

For the third condition, which is related with the amount of money the verified user has spent on internal games, we need to poll another microservice periodically (*the microservice which manages how much a user spent*). This microservice is a heavy load service which handles also basic components of the whole platform and thus it was

decided to not alter it at all, trying to avoid introducing latency on an already heavy process. In that case, a scheduled task sends a request on the corresponding service periodically. The referrals send the request in batches, in order to avoid adding a high database load on this service. More specifically, it collects all the user ids who haven't fulfilled the last condition yet and their referral is not expired and sends them to be checked on the corresponding microservice. For each referral it was found that the amount which was spent was enough on the corresponding microservice, a new entry is being created on the **referral_condition** table, which will declare that the third condition has been met. The overall architecture is displayed on Figure 5.

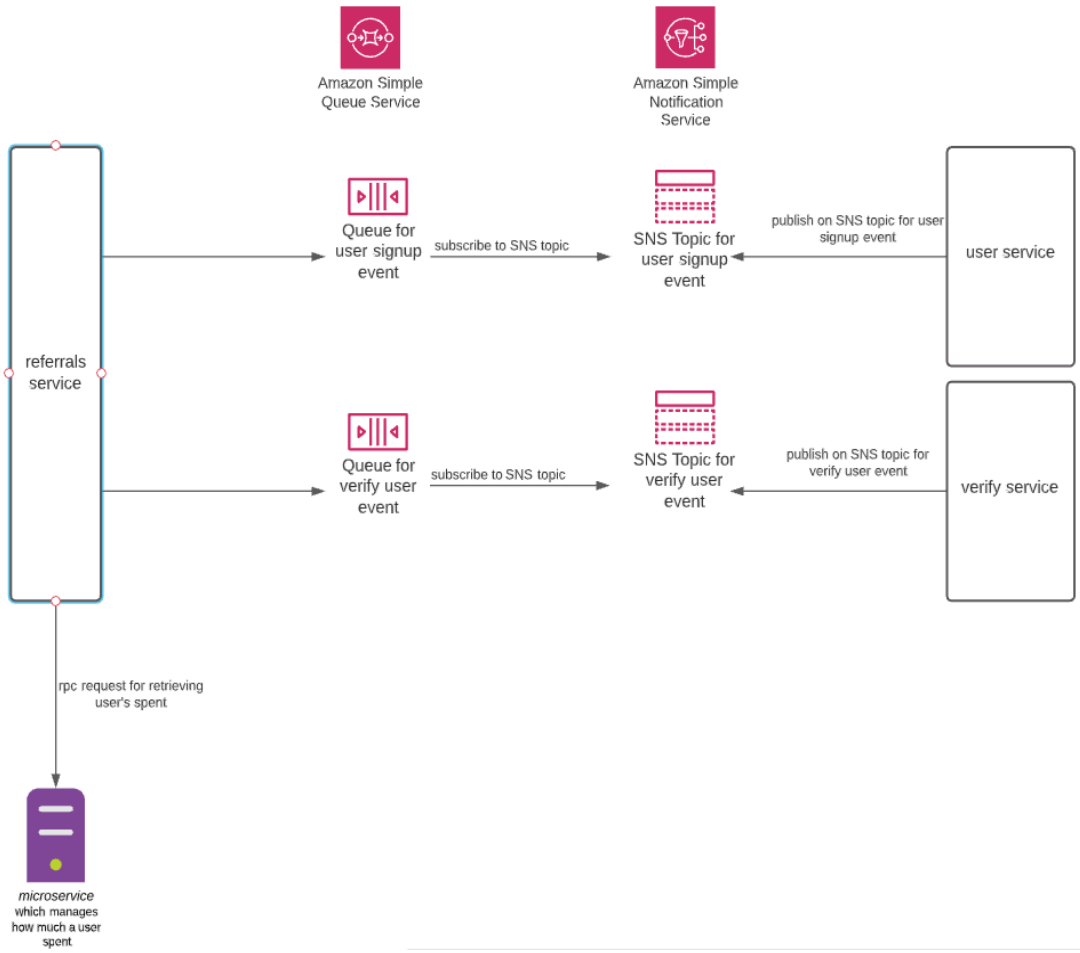

Figure 5: Overall architecture

4.2 Amazon EC2

As it was mentioned earlier, the service is deployed in Amazon's Elastic Computing (AWS EC2). EC2 is a part of Amazon's web services which allow users to rent virtual computers on which they can run their own applications, known as **instances**. Usually, we use more than one instance for each service in the production environment. Although, no matter how many instances we have deployed our software into, all of the instances connect to the same write database instance (Master). Because of that, the need arose to prevent more than one instance from making a request on the service that stores the information about the amount a user has spent. As we mentioned earlier, it should be avoided any extra load on this service. Also, in the case that two instances work on the same dataset, the risk of violation exceptions from the database would increase. For example, the database schema defines that only one user entry for a specific condition can be stored on the **referral_condition** table. In that case, a unique violation exception could be raised.

4.3 Task manager implementation

For addressing the above issue, the quickest solution would be to maintain another host, lets name it *referrals-management* which would be the only one responsible to run the periodic task. However, this solution would require extra configuration for making only the one host available to run the scheduled task which would have resulted in a dirtier codebase. It would also require duplicating any setup and configuration we needed for the primary host, but with extra changes in order to serve the management host, and last but not least, extra testing. That's why the following solution was one-way. Consequently, we chose to implement a task manager which would be able to manage the polling to the service. Particularly, the polling is being defined to run every 2 minutes. The task manager monitors when was the last time that the polling process ran and prevents any scheduled task to run again, if the time that passed from the last run is not higher or equal to 2 minutes. The run statistics are stored on the **task_manager** table and one entry is maintained each time. A database trigger was used for ensuring that only one entry is stored on the **task_manager** table each time. The **task_manager** table schema will be similar to the following:

```
CREATE TABLE IF NOT EXISTS `task_manager` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `in_progress` tinyint(1) NOT NULL,
 `updated_at` timestamp NOT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;
```

The database trigger will be similar to the following:

```
DELIMITER $$
CREATE TRIGGER accept_only_one_row
BEFORE INSERT
ON task_manager FOR EACH ROW
BEGIN
 DECLARE createdrows INT;

 SELECT COUNT(*) INTO createdrows FROM task_manager;

 IF createdrows > 0 THEN
   signal sqlstate '45000' set message_text = 'Only one task is
allowed';
 END IF;
END $$
DELIMITER;
```

The above trigger, on a new entry save action, it will check if the rows of the existing entries are more than 0 so it will not allow more than 1 entries to be stored on this table ever.

5. AWS SQS/SNS

5.1.1 Amazon SQS

Amazon SQS [8] (Simple Queue Service) is a managed message queue service offered by Amazon Web Services (AWS). It provides an API over http protocol, which is being used by applications for submitting items into and reading items out of a queue. The queue as a structure, is fully managed by AWS, which makes SQS an easy solution for passing messages between different parts of software systems that run in the cloud. Typically, one consumer subscribes to an SQS queue. There could be the case where more than one consumer could subscribe to one SQS queue, but this could cause issues given that all consumers would need to read the message at least once. Also, in case a message is being processed successfully, it is being deleted from the queue automatically.

5.1.2 Amazon SNS

Amazon SNS [9] (Simple Notification Service) is a publisher subscriber network, where subscribers subscribe to topics and are receiving the messages whenever a publisher publishes an event-message to that publisher. When an Amazon SQS subscribes to an Amazon SNS topic, a publisher, which in our case will be either the service which handles the user creation or the user verification, publishes an event-message to the topic and Amazon SNS sends an Amazon SQS message to the subscribed queue. The Amazon SQS message will contain the subject and the message which was published to the topic along with extra data (metadata) about the document, in a json format. This is called Fanout pattern [10] and it is the design we used for referrals.

5.1.3 Subscription of SQS to and SNS topic

We can create an SNS topic and an SQS queue and following, we can subscribe the SQS queue to the SNS topic, via AWS console. Firstly, we need to create an SNS topic [11]. The page should look like Figure 6.

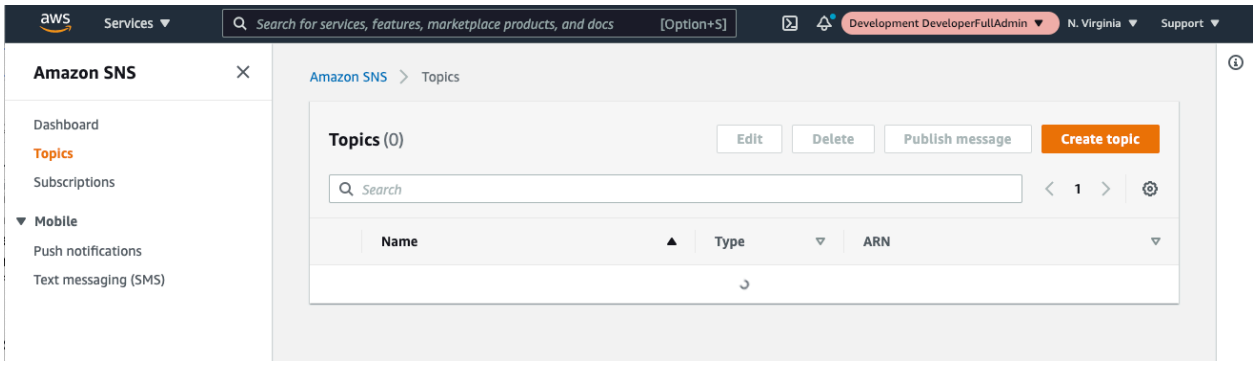

Figure 6: SNS page

By clicking on the *Create Topic* button, we will be transferred to the create topic page, where we can set a topic name, leave all the other settings as default and submit the request by clicking on the *create topic* button. Once we do that, the next page will inform us that our topic was created successfully (Figure 7).

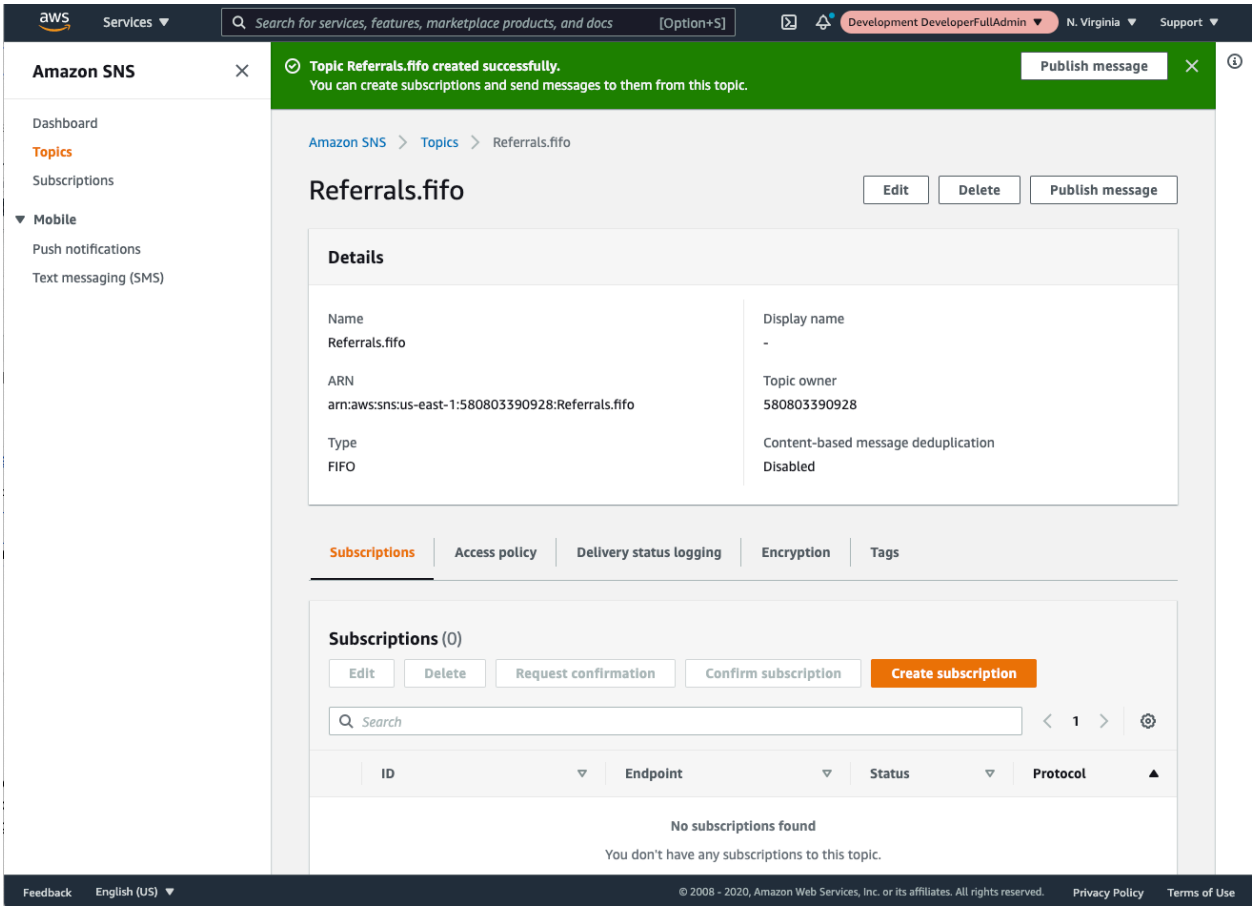

Figure 7: Newly created Topic page

The next step is to create the corresponding SQS Queue, which we will subscribe to the topic we just created. We need to visit the Amazon SQS cli [12], which will look like Figure 8 and click on the *Create queue* button.

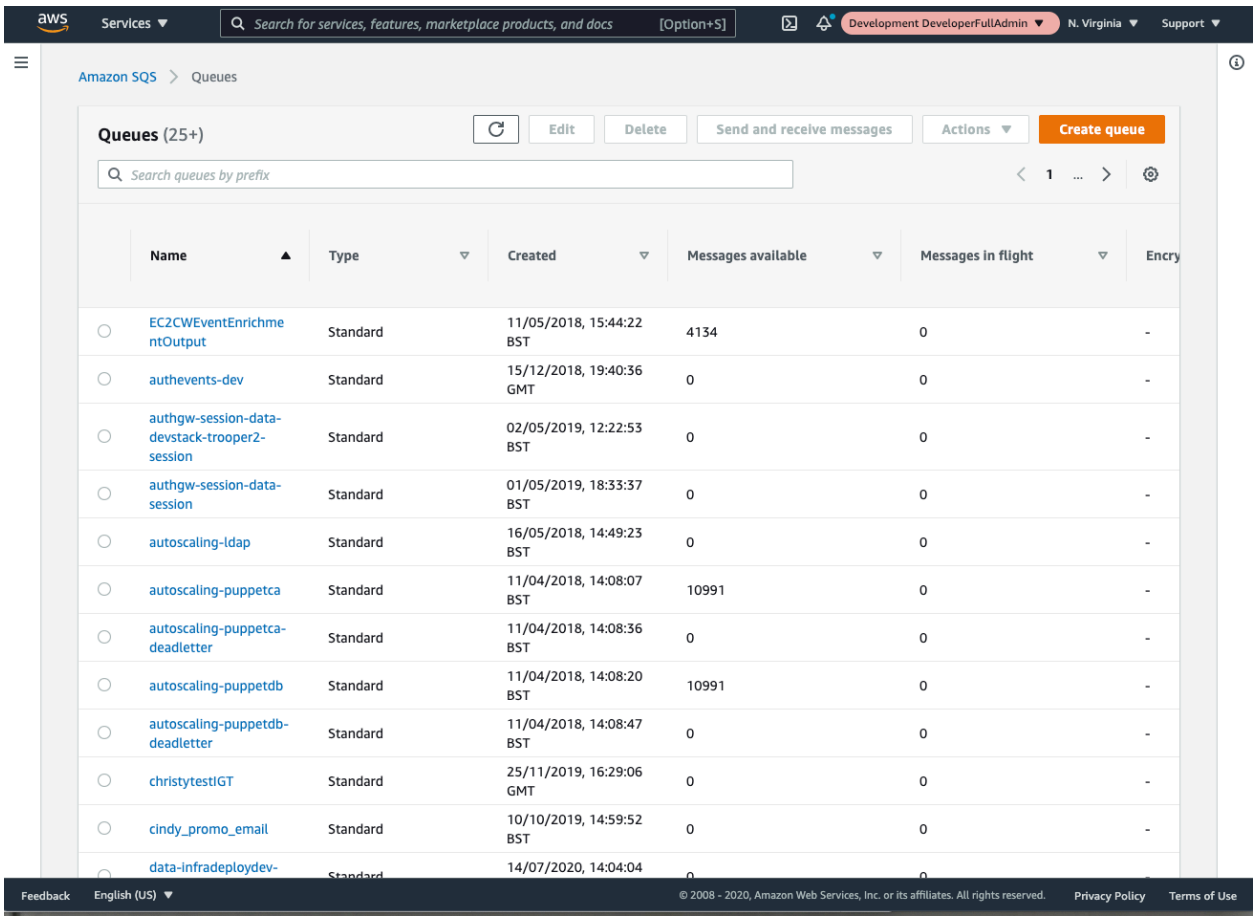

Figure 8: Amazon SQS page

On the *Create queue* page, we need to set a name for the queue, but we can leave the rest of the configuration as it is. In a production environment though, those default settings need to change. For example, on the access policy section the following are defaults (Figure 9) but if we don't change them, on a production environment our queue will be useless.

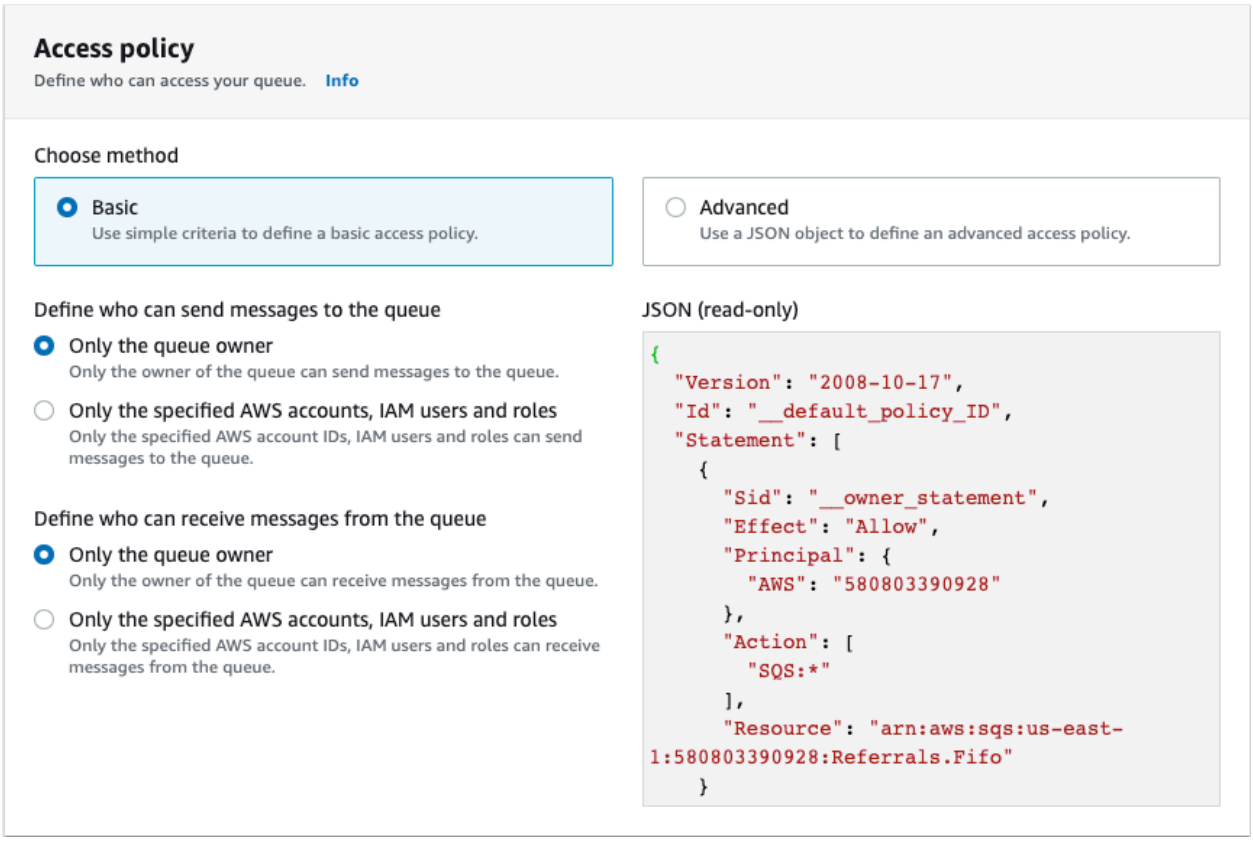

Figure 9: Default settings for a dev environment

Once we have created the SQS queue, we should get informed like Figure 10 and the next step will be to subscribe this SQS queue to our SNS topic.

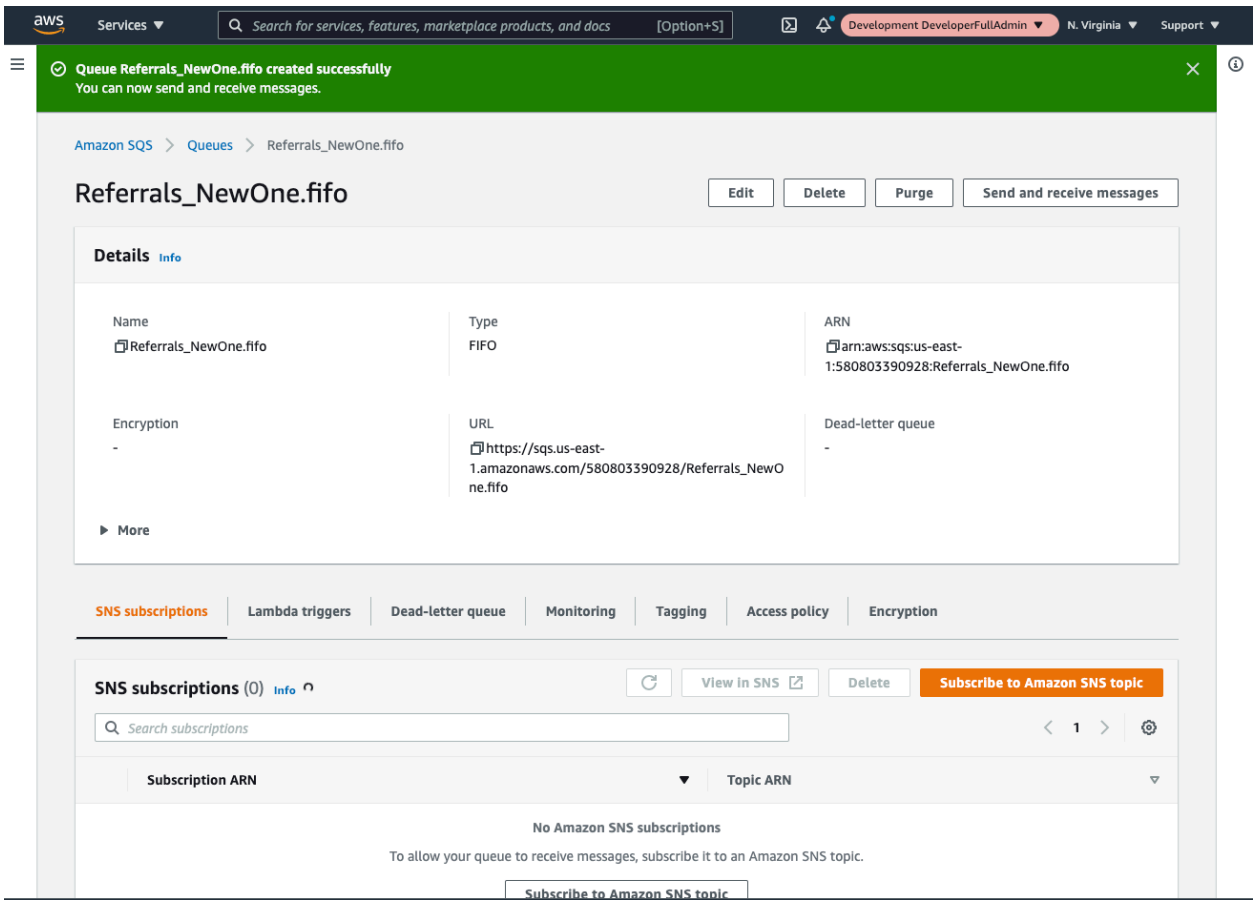

Figure 10: Newly created SQS Queue page

On the above page, we can click on the *Subscribe to Amazon SNS topic* button, which will lead us on the *Subscribe to Amazon SNS* topic page (Figure 11).

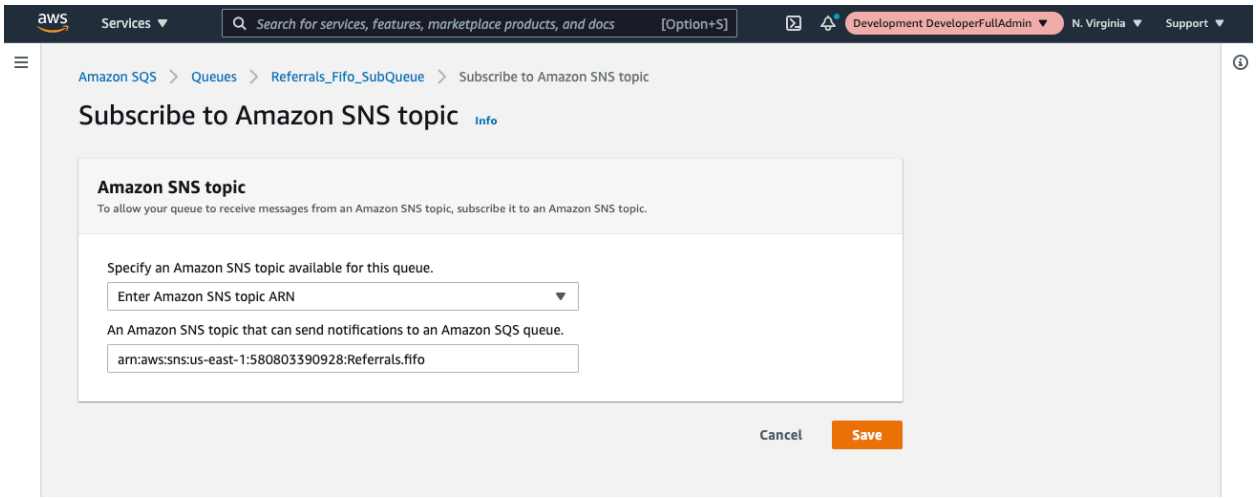

Figure 11: Subscribe to Amazon SNS topic page

On the *arn* input field, we need to paste the *arn* from Figure 7 and then to click the button *Save*. A new page like Figure 12 will be presented. After we have set up the SNS topic and the SQS queue which listens on that topic, we can listen for event-messages which are published on the SNS topic.

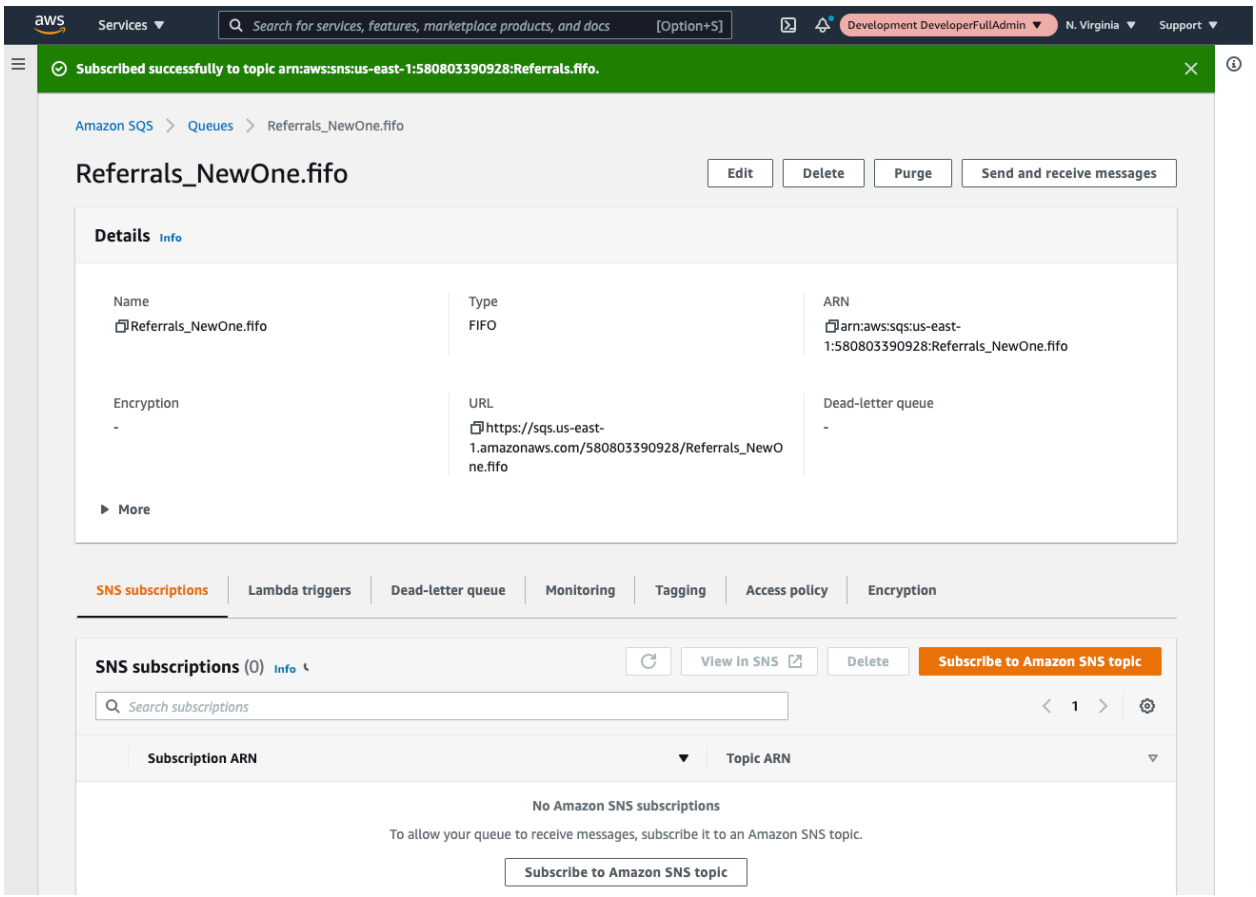

Figure 12: Successful SQS queue subscription on SNS topic

6. METRICS

For monitoring performance and issues that might come up about the new service, we used Datadog [13]. Datadog is a monitoring and analytics platform which gives a real-time insight of how a service performs in a development or a production environment. It is being used for diagnosing issues and for gaining “insight info” of the production code. There are Datadog Widgets available for monitoring the CPU level of an instance in production, the used memory, the api calls which take more time. Also, Datadog gives us the ability to create monitors which will trigger an alarm based on a condition, for example when the number of a specific error exceeds some threshold. Specifically, for referrals, we set various dashboards but some of the most important were the one which shows us which endpoints are heavily used, so we can monitor if there is anything which seems odd along with the CPU and memory load. Also, we set timeseries for the fulfilments of each condition. This is extremely important since it shows how the digestion of data performs, given that most of the data are entering referrals via message queues. Following figures show some widgets from the Datadog dashboard for referrals.

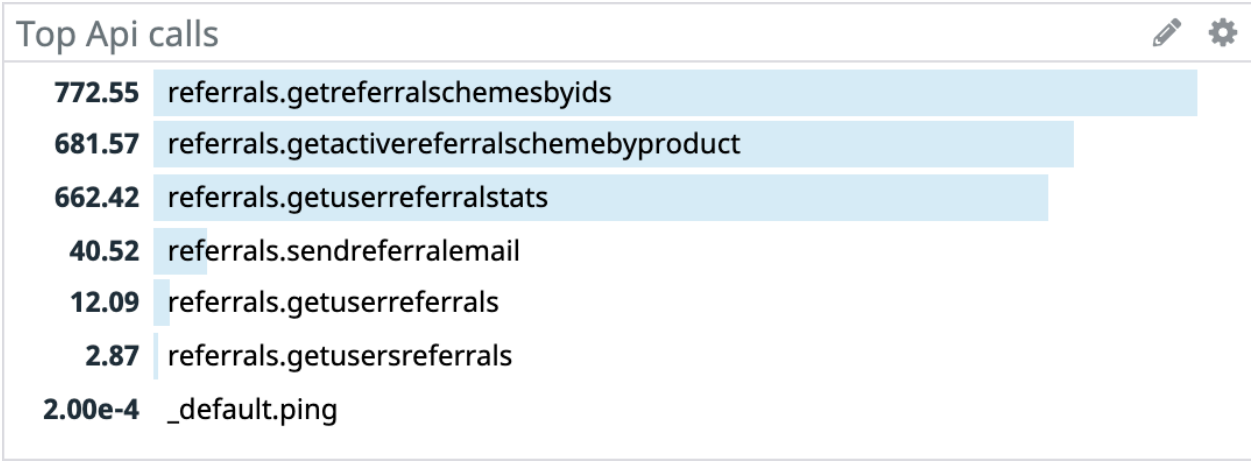

Figure 13: Datadog top API calls for referrals service

The above widget can present extremely different data of course, depending on the period we monitor the dashboard. For example in a period where a campaign promotion run and referrals are more active, the *referrals.sendreferralemail* could come in the first place of the most used endpoints.

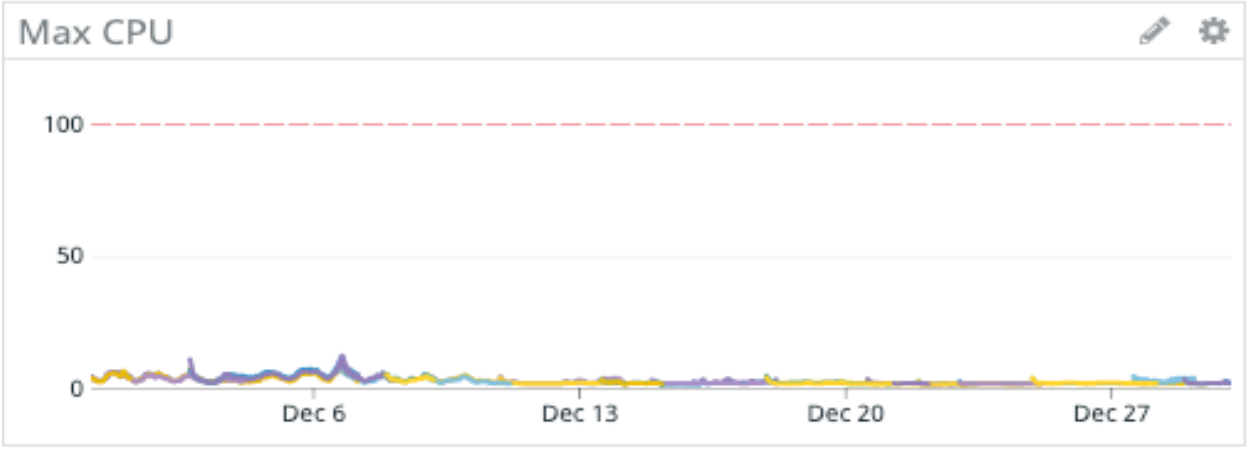

Figure 14: Max CPU rate

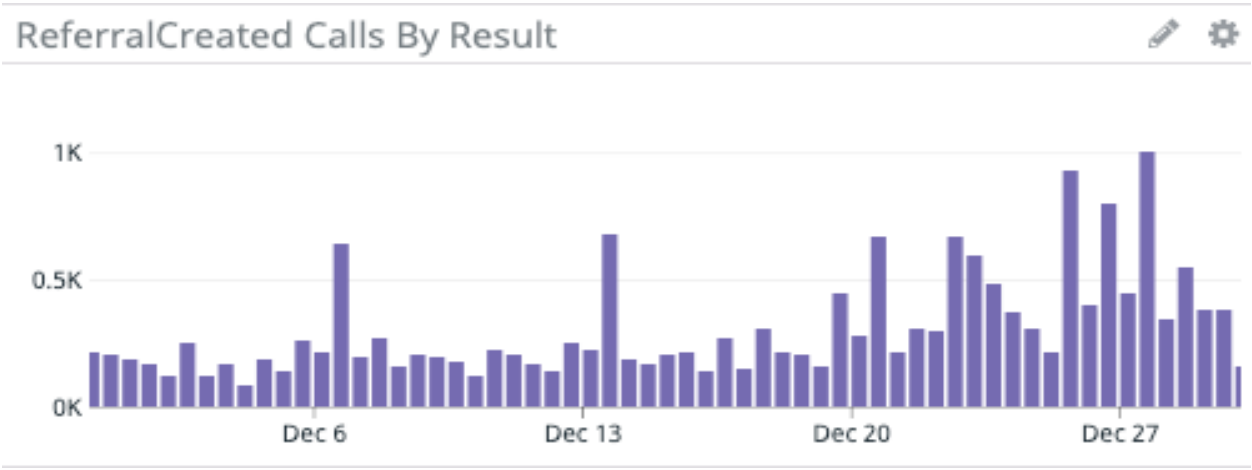

Figure 15: Referral created rate on referrals

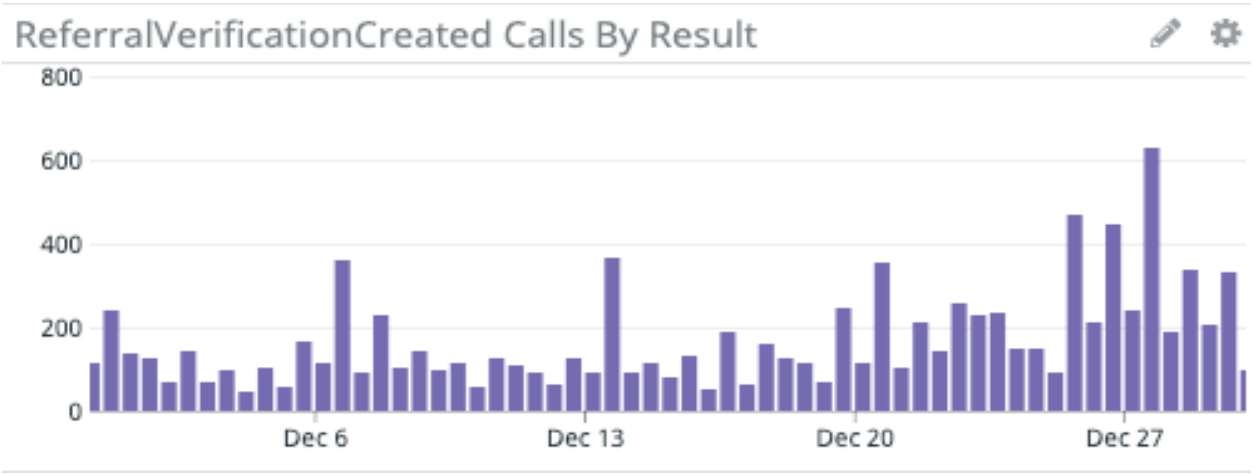

Figure 16: VerificationReferralCondition created rate on referrals

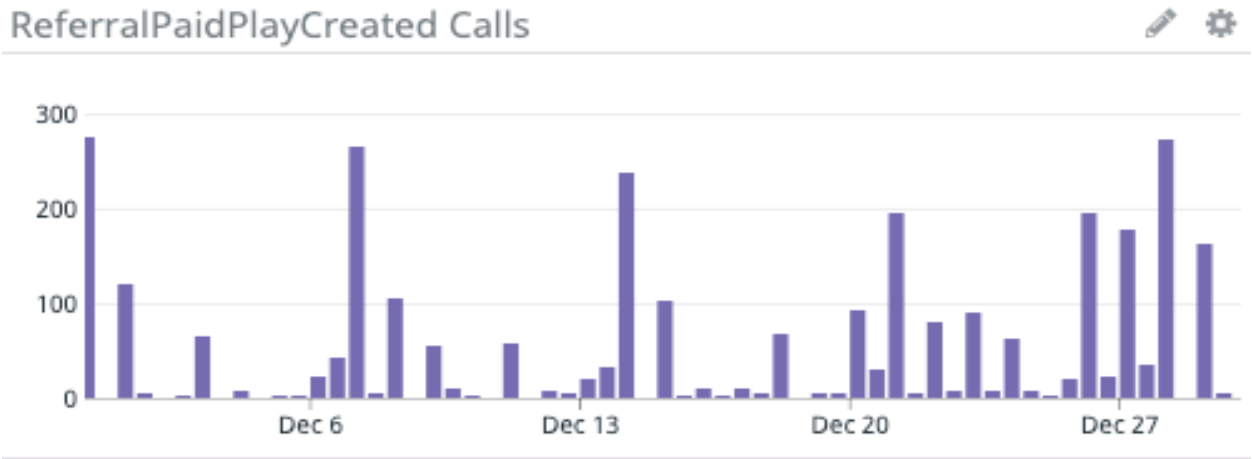

Figure 17: PaymentReferralCondition created rate on referrals

ACRONYMS

API	Application Programming Interface
AWS	Amazon Web Service
CRUD	Create Read Update Delete
EC2	Elastic Compute Cloud
JSON-RPC	Json remote procedure protocol
M:M	Many to Many Relationship
MySQL	My Structured Query Language
RPS	Reads per second
SNS	Simple Notification Service
SQS	Simple Queue Service

REFERENCES

[1] “Java 8 Central - Oracle.,” [Online]. Available: https://www.oracle.com/java/technologies/java8.html. [Accessed 29 December 2020].

[2] “SON-RPC 2.0 Specification,” [Online]. Available: https://www.jsonrpc.org/specification. [Accessed 29 December 2020].

[3] “Hibernate,” [Online]. Available: https://hibernate.org/. [Accessed 29 December 2020].

[4] “Create and activate an AWS account - Amazon AWS,” [Online]. Available: https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/. [Accessed 2020 December 29].

[5] “AWS Management Console - Amazon AWS,” [Online]. Available: https://aws.amazon.com/console/. [Accessed 29 December 2020].

[6] “Creating a DB cluster and connecting to a database on an ...,” [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_GettingStartedAurora.CreatingConnecting.Aurora.html. [Accessed 29 December 2020].

[7] “Know your customer - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Know_your_customer. [Accessed 29 December 2020].

[8] “Amazon Simple Queue Service - AWS Documentation,” [Online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html. [Accessed 29 December 2020].

[9] “What is Amazon SNS? - AWS Documentation,” [Online]. Available: https://docs.aws.amazon.com/sns/latest/dg/welcome.html. [Accessed 29 December 2020].

[10] “ Fanout to Amazon SQS queues - AWS Documentation,” [Online]. Available: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html. [Accessed 29 December 2020].

[11] “Amazon Simple Notification Service (SNS) - Amazon AWS,” [Online]. Available: https://aws.amazon.com/sns/. [Accessed 29 December 2020].

[12] “Amazon SQS | Message Queuing Service | AWS,” [Online]. Available: https://aws.amazon.com/sqs/. [Accessed 29 December 2020].

[13] “Datadogh,” [Online]. Available: https://www.datadoghq.com/. [Accessed 29 December 2020].