FoodShare

Eliminating Food Wastage using the Cloud

Mihail Calin Ionescu
Computer Science Department
University of Bristol
Bristol, UK
mi14828@my.bristol.ac.uk

Mudit Gupta
Computer Science Department
University of Bristol
Bristol, UK
mg14777@my.bristol.ac.uk

***Abstract*—FoodShare is an online platform that connects food contributors to consumers so that surplus food can be shared instead of being thrown away. This could be leftovers from last night, spare groceries in your fridge, food nearing its sell by date in large supermarkets and much more. Users can range from individuals to local stores to large food chains. Our service is built upon Google App Engine to provide a scalable solution to meet an increasing user base. This paper discusses the design and implementation details of the service and its evaluation via simulated user and load testing. FoodShare can be previewed at:**

https://opportune-study-185818.appspot.com/

***Keywords*—cloud, Google App Engine, food, share, scalability**

I. Introduction

A. Background

Food wastage is currently one of the biggest problems facing mankind. It has led to widespread hunger especially in poor and underdeveloped countries. According to a recent study done by the World Food Programme, 815 million people - one in nine - still go to bed on an empty stomach each night. Even more, one in three people suffer from some kind of malnutrition [1].
Majority of food wastage is avoidable if we produce and share food in an organized fashion. An average UK household dumps £470 worth of food in the bin just because there's no other alternative to its consumption [2]. This made us realize that having a platform where users can share excessive food with the rest of the community will be an easy and convenient way to solve this problem.

B. FoodShare

FoodShare is a website which provides people a means of sharing surplus food instead of throwing it away. Using FoodShare is simple. Whenever a user has a food item which he wants to share, he simply needs to open FoodShare website, upload a picture of the item, add a description along with pick-up information and expiry time. When another user wants to consume a shared item, he can click on consume on the item details page and arrange a pick-up via private messaging.

II. Motivation for using Cloud Computing

Cloud Computing is a service which provides users with on demand access to computing resources over a network. We opted for a cloud based solution due to three key benefits which are listed below:

1) Low Cost of using compute resources:
   Most cloud providers rent compute power on a flexible pay-as-you-go basis which means users only pay for the resources they use. This is useful for a startup like us with a limited budget.

2) Elasticity and ease of scalability:
   Using a cloud provider allows us to adapt to sudden changes in workloads by scaling instances up and down as required. This is a possible scenario in case the service is suddenly bombarded due to a natural disaster or calamity (famine etc.) which will lead to an abnormal increase in traffic.

3) Reduced Time-To-Market:
   This is a strong factor behind using cloud computing. It eliminates the hassle of buying and setting up required hardware which implies a reduced time-to-market for our product. Additionally, it also eliminates lots of security risks which we would otherwise need to take care of.

III. IaaS VS PaaS

Infrastructure as a Service (IaaS) is a cloud computing model which provides users with direct access to the underlying infrastructure such as servers, storage and networks. Users can spin up virtual machines and customize it however they want. It provides high level of control but also puts burden on the user to manage a lot of components.

Platform as a Service (PaaS) is a service model which provides users with a platform to develop and deploy their apps. Users are free to focus on development without having to worry about managing the underlying infrastructure.

FoodShare doesn't require any special control of the underlying infrastructure such as a specific OS or memory capacity. Thus, we chose PaaS since it allowed us to develop and deploy our application with a much reduced complexity. Besides, PaaS provides options for easy integration of resources (such as datastores, message brokers etc.) which is highly desirable.

IV. GAE vs AWS

Amazon and Google are the two major cloud players today with a lot of developer support so we decided to choose one of them. Amazon Web Services (AWS) is primarily known for IaaS but in recent years they have come up with a PaaS offering in the form of Elastic Beanstalk. On the other hand, Google's first cloud offering was PaaS in the form of Google App Engine (GAE) which is much more mature now in comparison to Elastic Beanstalk. GAE is also much easier to use with a better learning curve for beginners. GAE integrates with Google's CDN out of the box and distributes an application's assets through that increasing loading speed considerably. Due to all the above reasons we ultimately decided to go with GAE.

V. System Functionality

In this section, we will look at different use cases of our system and how a user can interact with it to perform different tasks.

A. User Login

On visiting FoodShare, the user is greeted with a landing page that shows a list of top contributions that are currently active. Before the user can start contributing items or consume existing items, he must login first. On clicking the Login button on the top right corner of the page (as shown in Figure 1) the user is directed to the Google Accounts sign in page.

Fig. 1. Landing page with login button (top right)

B. Viewing and Consuming active Contributions

A user can browse through the list of contributions on the main page. On clicking the details button, the user is taken to a new page with all the relevant details of that contribution such as contributor's username, picture of item, pick-up address and description (Figure 2). Additionally, user has the option of consuming the item and getting directions from Google Maps to the pick-up address.

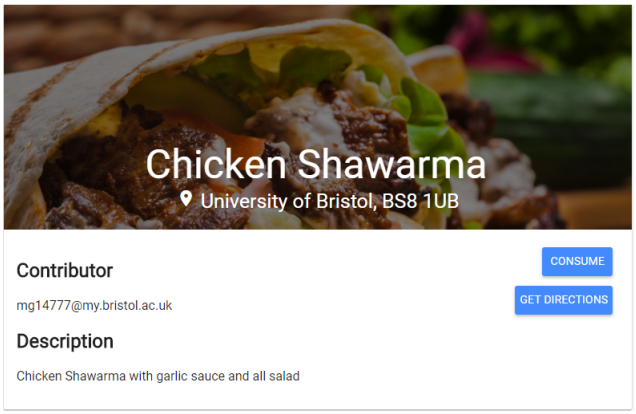

Fig. 2. Details of an active contribution can be viewed individually

C. Making a contribution

To contribute a food item, the user can simply click on the *Contribute* button on the navigation bar. He will be presented with a popup form (Figure 3) asking for the relevant details including name of item, picture, pickup address, expiry and short description. On clicking submit the server validates the input fields and stores them in the datastore.

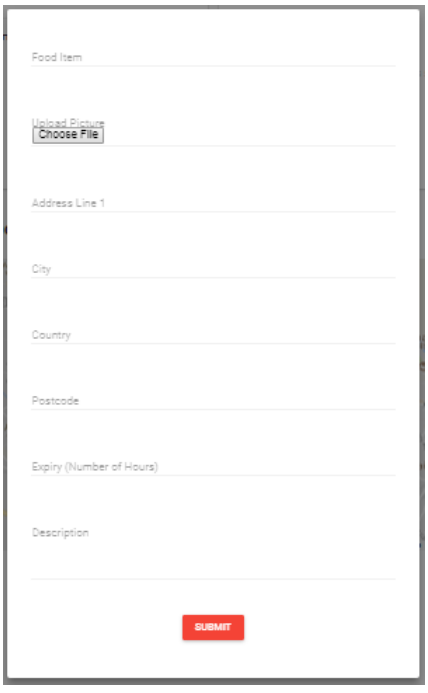

Fig. 3. Form for making a contribution

D. User History and Locality Search

On logging in, a user can access his dashboard by clicking on the *Dashboard* button on the navigation bar. The dashboard provides the user with a history of all his previous contributions and any consumptions he may have made (Figure 4).

Additionally, the dashboard contains a locality search feature

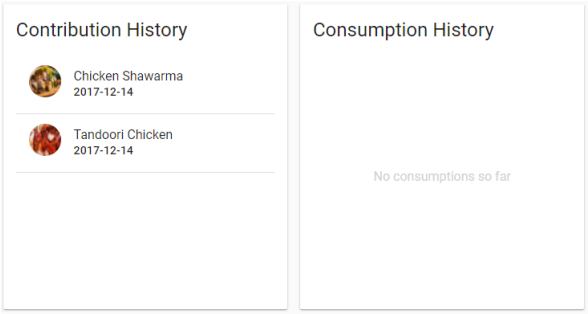

Fig. 4. Contribution and consumption history of a user can be viewed from the dashboard

that informs the user about all contributions which are within a 5km radius of his current location. These are displayed on a map with the user's location marked as "You" (Figure 5).

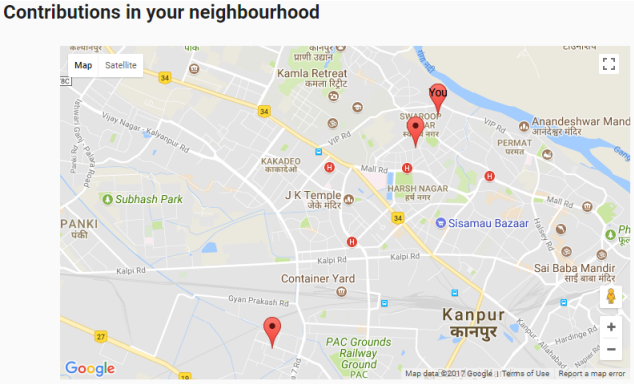

Fig. 5. Locality search allows a user to view contributions near him

VI. System Architecture

The architecture of the system can be divided into two main components. A *frontend web application (FoodShareUI)* which implements the user interface and several *microservices* running independently in the background which are all part of the backend. The architecture is shown in Figure 6.

When a client visits the website for the first time, Google routes the request to our frontend application which creates a template with all the static and dynamic content. In order to generate the dynamic content, the frontend app makes multiple REST API calls to various microservices for getting appropriate data. The requests to different microservices are routed via the App Engine dispatcher (*dispatch.yaml*) which also acts as our API Gateway. Each microservice runs independently to deliver a specific purpose as we will discuss shortly. Certain microservices also communicate with External APIs to accomplish different tasks. Once the content is ready, the template is sent back to the client as response.

The different microservices and their functions are described below:

1) **Share service**: This service is responsible for handling creation of new contributions and retrieving details of existing contributions. It is also responsible for keeping track of contributions which have been consumed and ones which have expired. Share service interacts with Gmail API to send an email to a user whenever his contribution is consumed by someone.
2) **Image service**: As the name suggests, image service is responsible for retrieving images of contributions for display on the UI.
3) **Locality Search service**: It is the implementation of the locality search feature we discussed in the System functionality section. User's location is passed in the form of geolocation coordinates. The service first does a reverse geocode lookup to determine the address of user, in particular the city. It then uses this information as a filtering mechanism to only consider contributions in the same city. We have made the assumption that contributions within 5km radius are very likely to be in the same city. The database is queried to get all such contributions. The service uses Google Maps Distance Matrix API to find the walking distance to all candidate contributions and returns the ones which are within the 5km radius.

A. Motivation for using Microservices

Microservices or Microservice architecture is an architectural style that structures an application as a collection of loosely coupled services. It is a variant of Service Oriented Architecture (SOA) architectural style. Each microservice is independent and is responsible for a small number of closely related tasks. If needed, microservices generally communicate with each other using APIs or well defined interfaces. The key benefits of using microservices are listed below:

1) It enables independent development and deployment of each microservice which streamlines project execution
2) Different microservices can be scaled independently based on demand
3) Improves fault isolation since only services which fail will be unavailable while others will still function properly. In other words there is no single point of failure as in a monolithic application

Initially, we started off with a single monolithic architecture. However, we soon realized that some parts of our architecture perform a lot more heavy-lifting than others. For example, we noticed that loading images takes a lot more time than retrieving the theoretical details of a contribution. When

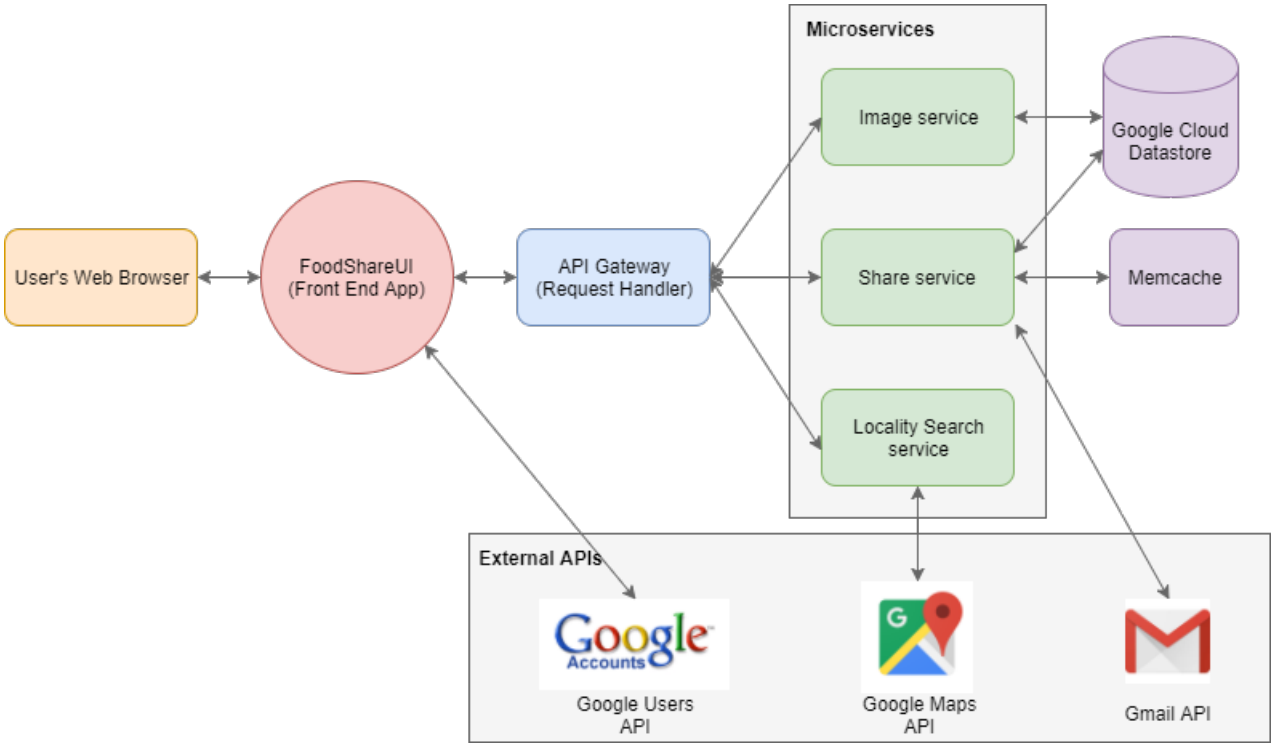

Fig. 6. System Architecture of FoodShare

we came up with the locality search feature we faced the same issue again since calculating contributions in close proximity to a user takes a lot more processing than retrieving contribution details from the datastore. Thus, we ultimately decided to move on to a microservices approach.

B. Decomposing a Monolith into Microservices

Decomposing a monolith into microservices is generally a challenge which we also faced in the beginning. In order to ease the process, we followed GAE's guide on "Migrating to Microservices From a Monolithic App" [3] and "Best Practices for Microservice Performance" [4]. Based on these guides we attempted to split our monolith into services that have different scaling needs. This led to the creation of *Image service* and *Locality search service*. Another point we took into consideration was having a separate service for entities accessed with CRUD pattern which led to *Share service*.

VII. Datastore

To store our data we went for Google's Datastore option for a few reasons. NoSQL databases have significantly better scalability than relational databases. The simpler model of data storage in NoSQL means aggregation operations are either computationally expensive or impossible to do at times. Since we mainly perform transaction operations (reads and writes), the performance provided by NoSQL databases is far better than SQL. Costs were another reason, Google's Datastore is free up to some daily limit, while Google Cloud Storage and Google Cloud SQL are a bit more expensive and do not have a free tier[11].
The non relational data base was useful throughout the development process as we could make minor changes without having to reinitialize the Datastore.
Our main data types are the contributions and consumptions which we realised are quite similar and only have a few different properties, so we decided to make use of the inheritance model offered by the NDB Datastore library and they both inherit from the parent model, "Share"[fig.7]. Some of the properties of the contributions are:

- contributor id - StringProperty - Links the contribution back to the user who created it
- timestamp - DateTimeProperty - The creation time of the contribution
- expiry - DateTimeProperty - The date when the contribution expires
- description - StringProperty
- address - StringProperty - Post code and location is extracted from the address
- picture - BlobProperty

Consumptions have two additional properties:

- consumer id - StringProperty - Links the consumption back to the user who consumed it
- consumption date - DateTimeProperty

We are storing the contribution images as a BlobProperty alongside the other model properties. As we let the used

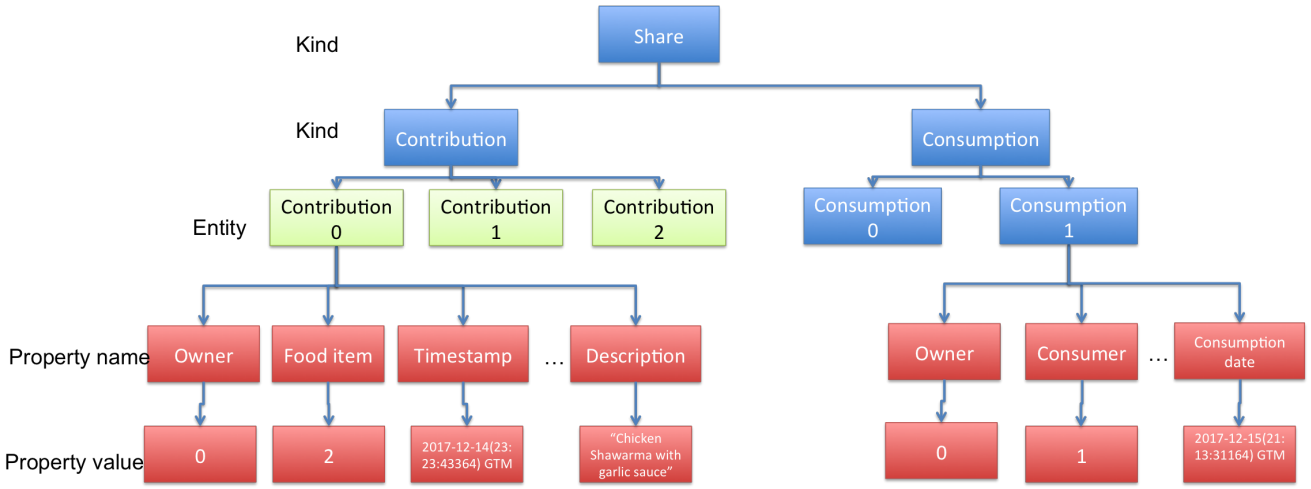

Fig. 7. Datastore model for contributions and consumptions showing the inheritance from the base "Share" class

upload and image size they want, the Datastore retrieval might be slowed down considerably. We have tried to solve this issue by saving all images using the Google Blobstore service, but we had some problems relating to the fact that Google does not officially support Blobstore anymore and they recommend to migrate to Google Cloud Storage, which, unfortunately, does not have a free tier for datastorage.

We used a few more Datastore models, one for the food items to remove data redundancy, multiple contributions can point to the same food item, this is useful because it allows us to add additional information for each food item without having to replicate it everywhere, eg. nutritional values.

There is also another model to extend the user object that we receive from the Google Users API. The user extension model has the user id so it can be linked back to a Google User and the additional properties:

- User photo
- Reputation
- Number of consumptions

The Memcache[10] was implicitly used to speed up database accesses. We are using the free, shared (default) service level which tries to provide cache capacity for the application on a best-effort basis[12]. This is helpful especially when we get lots of requests from people in the same city. The consumptions they receive on the main page are the same, so they will most likely be found in the Memcache.

VIII. Scalability

Scalability can be defined as the ability of a system to continue to perform well under an increased or expanding workload [5]. Using a PaaS platform like GAE conveniently handles most of the scalability issues for us in the background. However, we had to configure the scaling settings for each of our modules depending on their workloads. This is discussed later in this section.

Our approach to designing a scalable system was based on the scale cube shown in Figure 8.

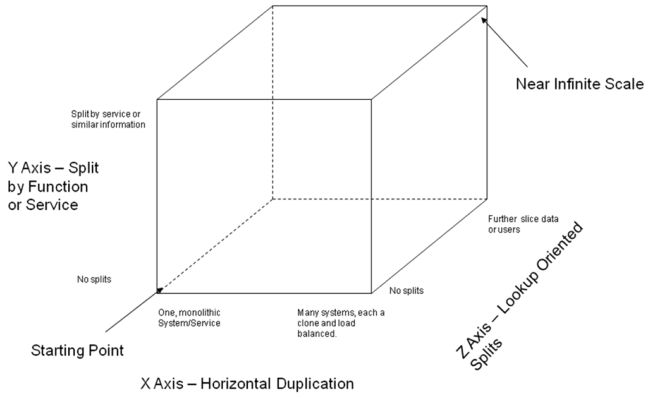

Fig. 8. The scale cube [6]

1) **X-Axis Scaling**: It consists of running multiple instances of an application and then balancing the load using a load balancer. The idea is that if we run K instances then each instance only needs to handle 1/K of the load. In FoodShare, X-Axis scaling is performed by GAE by replicating each module independently depending on workload to serve an increasing user base. GAE also provides vertical scaling options in the form of different types of instances with different compute power. A list of different instance classes provided by GAE along with their details is shown in Figure 9.

2) **Y-Axis Scaling**: In this scaling type we decompose a monolith application logic layer (based on function) into multiple independent services. Each service is responsible for one or more closely related tasks. The services are self-contained and can communicate with each other via REST interfaces.

The main reason for using microservices in FoodShare

Instance Class	Memory Limit	CPU Limit	Supported Scaling Types
B1	128 MB	600 MHz	manual, basic
B2	256 MB	1.2 GHz	manual, basic
B4	512 MB	2.4 GHz	manual, basic
B4_1G	1024 MB	2.4 GHz	manual, basic
B8	1024 MB	4.8 GHz	manual, basic
F1	128 MB	600 MHz	automatic
F2	256 MB	1.2 GHz	automatic
F4	512 MB	2.4 GHz	automatic
F4_1G	1024 MB	2.4 GHz	automatic

Fig. 9. GAE Instance classes along with supported scaling types [7]

was to perform Y-axis scaling. In this way we can ensure that when traffic increases, we only scale those logic components that require scaling. If we hadn't done so the whole logic layer would unnecessarily scale together at once which would have been a wastage of resources.

3) **Z-Axis Scaling**: In Z-axis scaling we follow a similar approach to X-axis scaling i.e. running multiple copies of an application across multiple servers. The difference lies in the fact that each copy only deals with a subset of data. The partitioning (sharding) of the data is done based on a common attribute. This is mainly used for scaling databases and leads to improved cache utilisation, transaction scalability and fault isolation.
We don't perform any Z-axis scaling. This is signified by the fact that each of our microservices accesses the same datastore (as given in our architecture diagram).

App engine offers different scaling options to control the creation of instances. A brief description of each of these is given below:

1) **Basic**: A service with basic scaling will create an instance when the application receives a request. The instance will be turned down when the app becomes idle. Basic scaling is ideal for work that is intermittent or driven by user activity [8].

2) **Automatic**: Automatic scaling is based on request rate, response latencies, and other application metrics [8].

3) **Manual**: A service with manual scaling runs continuously, allowing you to perform complex initialization and rely on the state of its memory over time [8].

We decided to use automatic scaling over the other two methods. Automatic scaling takes into account more information than basic scaling when spawning/destroying instances and thus it is able to waste fewer resources. Manual scaling could be useful if the amount of resources needed does not fluctuate and it also comes with the option to create a multithreaded application, which is banned when using automatic scaling, but in our case, the resources required for each microservice depend on how many concurrent users there are, so dynamically adding or removing instances is more useful than hard-coding them manually. Another advantage of using automatic scaling is that GAE allows up to 28 instance hours per day on their free accounts, while basic and manual can only have a maximum of 8 instance hours per day.
We start with a single instance and, depending on the load, can go up to 20 instances for each microservice, on the free tier of the AppEngine, going for the paid version would allow up to 100 instances and more microservices. We are also using the CPU utilization as a heuristic to know when to reduce or increase the number of instances, a threshold of 50 percent is used, if the CPU usage is more than this, GAE will attempt to increase the number of instances, if less than the threshold for a brief period of time, then GAE will try to close the instance.

IX. Load testing

We have attempted to do a load test even though most people advice against it saying that "GAE just scales"[9]. We were interested to see what exactly happens because, while AppEngine and the Datastore will be scaled automatically, this does not mean that our app is designed to scale properly or in a cost efficient manner. For example, even though the Datastore scales, unncessary reads and writes are quite expensive and should be avoided.
We managed to simulate close to 100 requests per second [Fig. 10], this might not be a huge amount, but it was enough for us to be able to gain some information.
The front end app (FoodShareUI) which is fairly light weight only needed 5 instances in order to cope with the requests.

The localitysearch microservice, which is more computationally intensive had to scale up to 8 instances in order to serve all of the users. This was with around 15 contributions in the database. Unlike the other microservices, which do not depend on the Datastore that heavily, with a huge number of contributions nearby, the time taken to do the search will increase which will lead to an increase in the number of localitysearch instances.
The response time fluctuates[fig.11] there is usually a slight increase flowed by a slight decrease representing the point in time when a new instance was spawned and work was split. In this particular test we set an upper bound of 5 locality search instances and after the limit was reached, a relatively big increase in response time can be noticed (there is also a small decrease at the end from the fact that the test was gradually stopped).

X. Future Work

Currently, there is lots of scope of improving FoodShare by adding new features and increasing efficiency of existing functionality. Due to time constraints we weren't able to incorporate the following components and have left it as future work.

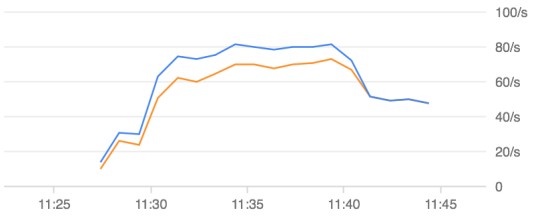

Fig. 10. Number of requests per second

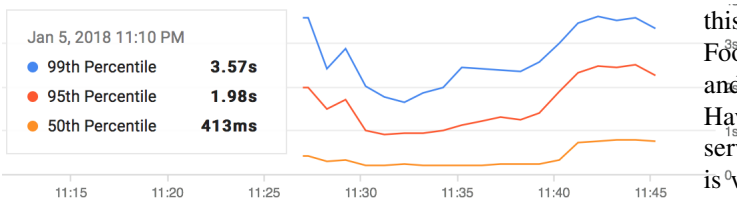

Fig. 11. Response time for the locality search microservice

A. User Reputation (Incentive Compatibility)

This feature is currently in partially developed stage. The idea is that each user should have a reputation score to signify the quality of his contributions. This serves two purposes. First, it improves the popularity of genuine users providing them an incentive to contribute more. Second, it helps in suppressing and eventually eliminating selfish users who contribute poor quality of food just to get rid of it.

B. In built messaging system

It would be great to have our own messaging system so that when an item is consumed the contributor and consumer can chat directly to arrange a pickup. This can also be used to gain more information regarding a contribution if needed. Currently, we notify the relevant user when his item is consumed using email which is not so efficient and convenient.

C. Improving efficiency of locality search

As demonstrated in load testing, locality search is computationally expensive which to a certain extent is reasonable to expect. However, we believe we can optimise our searching algorithm to reduce the latency.

XI. Conclusion

In this report, we have presented FoodShare, a user driven online platform that helps in eliminating food wastage by connecting food contributors to consumers. We started off by discussing our motivation to utilize cloud computing and the different service models (IaaS and PaaS) available. We then talked about our decision to use Google App Engine platform for building our service and why we chose it over Amazon Web Services (AWS). In section V we outlined the functionality of our system and the different features it provides. We discussed how the user can interact with the system to perform different tasks. Based on this functionality and requirements, we derived an architecture for our system which was discussed in section VI. We saw the motivation behind using microservices and its related pros and cons.

In section VIII we discussed in detail how the system is designed to scale based on increasing demand and how the different scaling options provided by GAE help us in achieving the same. We also analyzed the scaling settings used by us to certain depth. Finally, in section IX we demonstrated the amount of load our system is able to handle and how this affected our scaling settings for different microservices. FoodShare has been designed and built to provide a robust and scalable solution to the problem under consideration. Having said that, there is still a lot of scope to improve the service and provide a more immersive user experience which is what we intend to do in the future.

References

[1] Unknown, *Zero Hunger: World Food Programme* http://www1.wfp.org/zero-hunger

[2] Rebecca Smithers, *UK throwing away 13bn of food each year, latest figures show*. Jan 10 2017. https://www.theguardian.com/environment/2017/jan/10/uk-throwing-away-13bn-of-food-each-year-latest-figures-show

[3] Unknown, *Migrating to Microservices from a Monolithic App*. App Engine Docs, 2017. https://cloud.google.com/appengine/docs/standard/php/microservice-migration

[4] Unknown, *Best Practices for Microservice Performance*. App Engine Docs, 2017. https://cloud.google.com/appengine/docs/standard/php/microservice-performance

[5] Bondi, Andr B. (2000). *Characteristics of scalability and their impact on performance*. Proceedings of the second international workshop on Software and performance WOSP '00. p. 195.

[6] M. Abbot, M. Fisher, *The Art of Scalability*. 2nd ed. Addison-Wesley Professional, 2015

[7] Unknown, *The App Engine Standard Environment*. App Engine Docs, 2017. https://cloud.google.com/appengine/docs/standard/#instance_classes

[8] Unknown, *An Overview of App Engine*. App Engine Docs, 2017. https://cloud.google.com/appengine/docs/standard/python/an-overview-of-app-engine

[9] AppEngine load testing https://stackoverflow.com/questions/17621977/app-engine-load-testing

[10] John Lowry Designing for Scale. App Engine Docs, 2017. https://cloud.google.com/appengine/articles/scalability

[11] - Choosing a storage option. Google Cloud Platform Docs, 2017. https://cloud.google.com/storage-options/

[12] Unknown, Memcache Overview. Google App Engine Docs, 2017. https://cloud.google.com/appengine/docs/standard/python/memcache/#service_levels