

# GitHub Repository Analysis

**CS 441: Engineering Distributed Objects  
for Cloud Computing**

**Course Project Report**

**Abhijay Patne (UIN: 663324999)  
Sandeep Singh (UIN: 662153937)**



**COMPUTER  
SCIENCE  
COLLEGE OF  
ENGINEERING**



**Table of Pointers**

## **Introduction**

## **Highlights and Features**

## **Technology Stack**

Akka Actors with Scala  
Google Cloud Platform  
MongoDB  
MySQL  
RapidMiner  
Akka HTTP

## **Application Details**

## **How to Run**

## **Results Analysis**

## **Application Testing**

## **Learnings**

## **Limitations**

## **Future Work**

## **Known Issues**

## **File Description**

## **References**

## Introduction

The goal of the course project was to get hands-on experience with programming distributed systems which will be used in large scale distributed systems. This was very different from other projects where we did not have specified goal (given some requirements and expected outcome), so we had to first brainstorm about what are possible ways to do analysis, do machine learning. Some of the major technologies, concepts involved, but not limited to, are Akka Actor Framework, Akka HTTP framework, NoSQL databases(MongoDB), Relational databases(MySQL), Google Cloud Platform.

Initially, the application scans GitHub repositories using GitHub APIs where we specify different parameters such as language, timeline, size etc. These repositories are cloned locally to obtain some metadata from them. Additionally, information about the owners of these repositories is also obtained through GitHub API calls. Metadata about the repositories and users is stored in MongoDB as JSON strings of key value pairs. Once MongoDB has been populated with all the information mentioned above, some of it is filtered and stored in MySQL database.

Using RapidMiner, we analyze the data in MySQL database to find some interesting patterns in the gathered data.

A web service has also been created to provide some analytical results including recommended projects/repositories. The web service can be accessed using REST calls.

## Highlights and features

- Use of multiple Google Cloud tools
- Recommender system for Repositories
- Web Service hosted in cloud (<http://104.197.28.49:8080/>)

Note: The WebService fetches the data from a Google Cloud SQL instance. We have configured the database instance to go to standby mode after 12 hours of inactivity. So, if you don't get any response that means the database is in standby mode, in this case please contact us to start the database instance. We can't afford to keep the instance running 24\*7\*365

- Implemented in Scala for extra bonus :)

## Technology Stack (Software Requirements)

### Akka Actors with Scala

Akka is a toolkit for designing distributed and concurrent applications. Akka is available for many different programming languages including Scala, which is our primary choice for this project. Actors are the “fundamental units of computation that embody processing, storage and communication”. This allows the developer to focus on designing and implementing the functionality and leaving the other implementation of threads, their lifecycle etc. on the framework. Actors allows developers to create actors which can communicate asynchronously, have their own state and perform processing by handling messages or behaviors. The Akka Actor Model provides an abstraction for implementing concurrent programming. Akka actors can be considered as objects which implement their own state and behavior and communicate with each other through messages.

In our application, we have used the Actor Model for different purposes, varying from downloading GitHub Repositories to loading tables into MySQL.

### Google Cloud Platform

- Google Compute Engine: Initially we had created a Debian VM where we had installed MySQL and were planning to access it remotely from our driver program, but we were not able to connect it remotely. So we decided to use Google Cloud SQL
- Google Cloud SQL: This instance stores all our relational databases of repositories and users
- Google Cloud Storage: To store MySQL database dumps and other and store downloaded repository files
- Google Cloud Networking: To add firewall exception to access MongoDB VM, Cloud SQL instance from our program
- Google Cloud Launcher and Google Cloud Deployment Manager: To deploy MongoDB and MySQL solutions provided by Bitnami and other vendors
- Google Cloud IAM and Admin: Used this to give full 'Owner' access for google cloud project to my teammate, Sandeep

### MongoDB

MongoDB vs Neo4J: Why did we choose MongoDB

MongoDB is one of the highly ranked NoSQL database in terms of popularity and is a document based data store which accepts JSON as input records. As the data we obtained from GitHub APIs was in the JSON format, we decided to use MongoDB. Neo4J is a graph database and treats records as nodes which are connected to each other with some relation. For one of our ideas of building a ranking of users based on PageRank algorithm, we were planning to use Neo4j but because of lack of time, we did not implement it.

## MySQL

MySQL is an open source relational database, and one of the most popular ones. We needed to filter out some information from the MongoDB database and store it in a relational database, and MySQL fit our requirements perfectly.

We are using the Google Cloud SQL for storing our relational database, which can be accessed through remote API calls.

## Akka HTTP

The Akka toolkit also contains Akka HTTP, which is a toolkit intrinsically based on Akka Actor for creating REST APIs. It exposes Akka Actors to the web, and can be used for creating REST APIs, both client and server side.

We have used Akka HTTP to create a web service that connects to the MySQL database in the back end, and can be utilized through REST calls.

## BLACKDUCK | Open Hub

Black Duck Open Hub allows users to browse through open source projects using REST API calls. The response is in XML format allowing an easy parse and to retrieve project details including the home page, download page URLs for repositories. Black duck allows users to filter the results by specifying keyword in query which can be used to filter the projects with some keywords. But the major issues in using Black Duck | Open Hub REST API are:

1. The number of requests per day limit is 1000 which is very less.
2. The filters allow the user to filter the projects with keywords matched only from name, description, or URLs. There are no other filters like filtering the results on the basis of the created date.
3. Many projects do not have direct download URLs or have blank URLs which requires extra line of code to download the project repositories.

## Application Details

### Project Flow

As a first step, In **GithubProcessor.scala** we create actors to download GitHub repository details in the form of JSONs using some API search queries. The username of the owners of these GitHub repositories are also extracted, and JSONs containing their information are also obtained using the GitHub API. These different JSON strings are written to MongoDB (hosted in Google Cloud) using API calls. All the different tasks mentioned here are executed using different actors which communicate with each other using messages. **MongoDBOperationAPIs.scala** contains different methods which make these API calls to MongoDB, and these methods have been utilized by the actors mentioned previously.

After MongoDB has been filled with information about GitHub repositories and users, in **MongoDbToMySQL.scala** we create an actor to clone GitHub repositories (whose names and other details are stored in MongoDB) to extract some extra metadata about them. We clone these repositories using the JGIT library. Another actor is created to insert some filtered metadata about users and repositories into MySQL, which is also hosted in Google Cloud. **MySQLOperationAPIs.scala** contains methods that interact with the MySQL through API calls, and these methods are utilized by different actors to interact with MySQL.

Lastly, once the MySQL storage has been set up, we can run **WebService.scala** which creates and starts a web service using Akka HTTP. This web service takes different parameters from the user, which can be used for getting some analytical results. The web service intercepts user's requests and interacts with **MySQLOperationAPIs.scala** to retrieve information from MySQL. This information contains details of the kinds of repositories or users which the user is interested in. The web service provides a response to the user in the form of JSONs.

**SLF4J** has been used to log information from the different class mentioned above, and the log created can be utilized to analyze the entire application's execution.

### RapidMiner

We have used RapidMiner to find the patterns from downloaded data. RapidMiner can import data from flat files or database. We have imported data from our MySQL data store and stored in RapidMiner's Local Repository [Fig 1]. RapidMiner has reach set of APIs to process this data, build Machine Learning models on them. We created processes for different analysis steps we performed. Snippet of one of them is mentioned below [Fig 2].

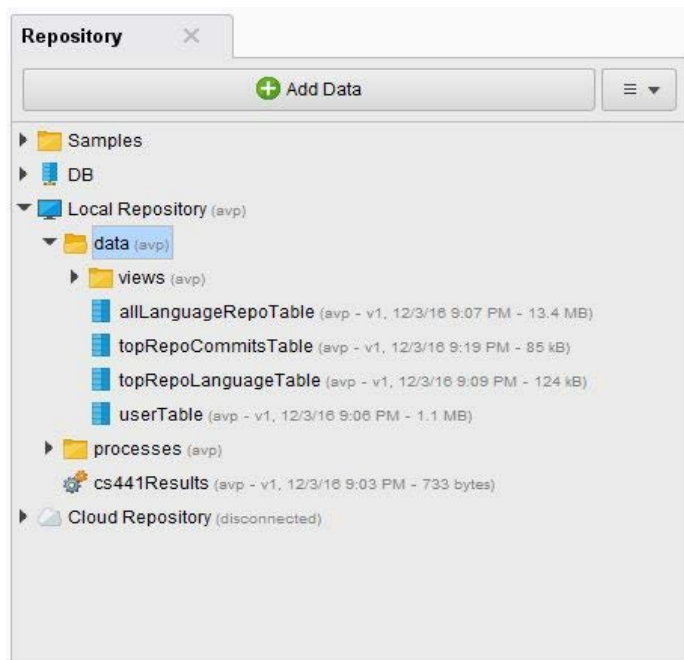


Fig. 1 RapidMiner Local Repository

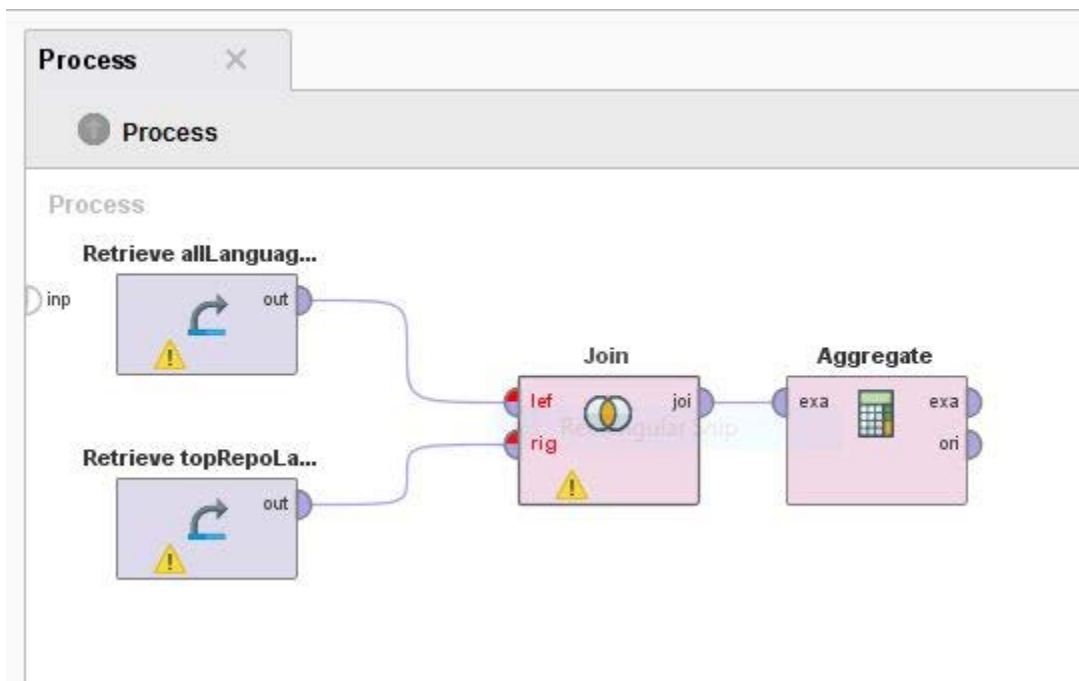


Fig. 2 Sample RapidMiner process

## Project Structure

1. **GithubProcessor.scala**: This is the first class file that should be run when trying to setup the entire project and databases. (If databases have already been loaded, WebService.scala can be run directly)

```

GithubProcessor class ==> downloaderActor ==> jsonParser Actor =====
      ^ ^ (passes downloaded || ||
      || repo/user JSON) || ||
      || || ||
      || (passes back user name) || ||
      ===== ||
                                     ||
                                     ||
                                     ||
mongoDbConnector Actor <=====
(gets all user/repo JSON and
writes them to MongoDB)

```

An object named **GithubProcessor** has been created which instantiates our main class, named **"initializerClass"**. We create 3 actors here: **downloaderActor**, **jsonParser** and **mongoDbConnector**.

Firstly, this classes passed a message to the actor `downloaderActor` which downloads repositories metadata and sends it for further processing.

## Actors

- **downloaderActor** This actor downloads the repo details for GitHub repositories specified by language, created date, and lower limit for size. All GitHub queries that we have used have been mentioned later.

The repo details are retrieved in the form of JSON strings which are stored in files locally. Instead of downloading these locally, we can also process them on the fly. Since GitHub has a restriction on the number of API calls we can make in one minute (for some API calls these limits are also per hour), our actor goes to sleep until the next minute start so that we do not encounter a “limit exceeded” response from the GitHub API call.

Next, a message is passed to the second actor “**jsonParser**” which parses the downloaded JSON strings.

After processing of the JSON strings this actor `jsonParser` gives a message back to `downloadActor`, which downloads the user details through GitHub API calls. These user names are extracted by the `jsonParser` actor described below.

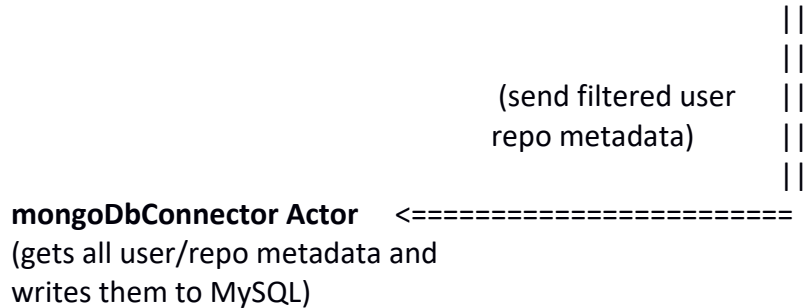


- **jsonParser:** This actor reads the JSON string which had been downloaded and saved locally as mentioned before. It processes the JSON to extract the username for the owner of the repository. Later, it sends various messages to the third actor, “**mongoDbConnector**” providing it the entire repo JSONs.
  - **mongoDbConnector:** This is the third and the final actor in this file. This actor gets the user/repo JSON from the second actor, and uses an API call to MongoDB to write the JSON there, in the appropriate collection. This API call is made by calling a method in **MongoDBOperationAPIs.scala** (described later).
  - **HttpBasicAuth:** This class is used by other actors to make the GitHub API calls with authentication. GitHub allows a higher limit for authenticated API calls, so this comes in very handy whenever we make multiple API calls to GitHub.
2. **MongoDBOperationAPIs.scala**  
This file contains MongoDB Scala driver connector. It connects to remote MongoDB virtual machine hosted in Google Cloud and contains APIs to perform various MongoDB operations such as insert, add, query data from database.
  3. **MySQLOperationAPIs.scala**  
This file contains MySQL Scala driver connector. It connects to Google Cloud SQL and contains APIs to perform various MySQL operations such as insert, add, query data to database.
  4. **ParameterConstants.scala**  
Contains various constant parameters used in all programs such database hostname, database name, driver name, collection names in case of MongoDB etc.
  5. **MongoDbToMySQL.scala:** This class uses the Akka actor system to read data from MongoDB, download some extra metadata, and write the filtered information to MySQL. This class should be executed after MongoDB has been populated by **GithubProcessor.scala**.

```

MongoDbToMySQL.scala =====> mongoDbReaderActor =====> getMetadataJgit Actor
                                (pass repo/user JSON                      ||
                                retrieved from MongoDB)                    ||
                                ||                                         ||
                                ||                                         ||
                                ||                                         ||

```



Different behaviors in the actor “mongoDbReaderActor” are sent messages, which initiate the process of extracting information from MongoDB.

- **mongoDbReaderActor:** After getting JSONs for different repos from MongoDB, we extract their URLs and passes these as messages to the actor **getMetadataJgit**.

This actor also gets the entire user JSONs from MongoDB, extracts the user subscription URL (is used to get some extra information about the user) and makes API calls to GitHub to get this user metadata. Again, authentication is used for GitHub API calls to get a higher limit. This user metadata and repo metadata is passed onto another actor.

- **getMetadataJgit:** This actor clones the repositories using Jgit library to obtain some extra relevant information like total lines of code, total number of files, different languages used, number of commits etc.  
Subsequently, all the filtered details of users and repos and passed onto the actor **mySqlWriterActor**.
- **mySqlWriterActor:** This actor performs one basic function: write all the data it receives to MySQL into specified tables.  
Different cases have been created in the actor to receive different kinds of data, and write them to different tables in MySQL.  
There are separate methods defined in MySQLOperationAPIs.scala which make API calls to MySQL to insert rows into different tables. These methods are used by this actor to write into the database.

## 6. WebService.scala

The web service (created using Akka HTTP) listens for REST calls to produce different types of analytical results based on the information in the MySQL database.

WebService Class =====> Request received =====> Interacts with MySQL to get response

(starts web service,  
waits for rest calls)

||  
||  
||  
||  
\\

Output JSON response to REST call

Once a request is received, the request parameters are obtained, and accordingly a SQL query made to the MySQL database, which provides a response which can be given as output to the rest call. The output of the web service is formatted in the form of a JSON response.

Various web service parameters are described in the **How to Run** section and their analytics results are described in the **Results Analysis** section.

## Database schema

### MongoDB (NoSQL Database):

We have used MongoDB to store repositories downloaded from GitHub REST API. These repositories are downloaded in the JSON form and can be directly inserted in MongoDB. We have created a separate collection for each languages and one collection for user data which we download from GitHub. Here is the list of collections created

Language collections: (cCollection, cppCollection, cs441test, csharpCollection, goCollection, htmlCollection, javaCollection, javascriptCollection, phpCollection, pythonCollection, scalaCollection)

Users' collection: usersCollection

Keys retrieved and stored in a language document are:

> **Object.keys(db.javaCollection.findOne())**

```
[ "_id", "id", "name", "full_name", "owner", "private", "html_url", "description", "fork", "url",  
"forks_url", "keys_url", "collaborators_url", "teams_url", "hooks_url", "issue_events_url",  
"events_url", "assignees_url", "branches_url", "tags_url", "blobs_url", "git_tags_url",  
"git_refs_url", "trees_url", "statuses_url", "languages_url", "stargazers_url", "contributors_url",  
"subscribers_url", "subscription_url", "commits_url", "git_commits_url", "comments_url",
```

```
"issue_comment_url", "contents_url", "compare_url", "merges_url", "archive_url",
"downloads_url", "issues_url", "pulls_url", "milestones_url", "notifications_url", "labels_url",
"releases_url", "deployments_url", "created_at", "updated_at", "pushed_at", "git_url",
"ssh_url", "clone_url", "svn_url", "homepage", "size", "stargazers_count", "watchers_count",
"language", "has_issues", "has_downloads", "has_wiki", "has_pages", "forks_count",
"mirror_url", "open_issues_count", "forks", "open_issues", "watchers", "default_branch",
"score" ]
```

Keys for users' collection:

> **Object.keys(db.usersCollection.findOne())**

```
[ "_id", "login", "id", "avatar_url", "gravatar_id", "url", "html_url", "followers_url",
"following_url", "gists_url", "starred_url", "subscriptions_url", "organizations_url", "repos_url",
"events_url", "received_events_url", "type", "site_admin", "name", "company", "blog",
"location", "email", "hireable", "bio", "public_repos", "public_gists", "followers", "following",
"created_at", "updated_at"]
```

### MySQL (Relational Database):

We have used Google Cloud SQL for storing our relational databases. After studying the structure of our NoSQL database, we decided to do different analysis based on some parameters which we are storing in our database. Following is our schema and details of tables. Primary keys are marked in a rectangle.

- allLanguageRepoTable: This table stores details about all 174K repositories we downloaded from GitHub. These details include repository name, id, username, repository creation date, updation date, forks count, size of the repository, open issues and watchers count of the repository
- topRepoLanguageTable: This table stores details about 2392 popular repositories, having forks count greater than 4 and repository size greater than 10 MB. These details include different languages used, number of lines and number of files of each language
- userTable: This table contains information about all the users whose repositories were studied. This information includes number of public repos, user's followers count, number of subscriptions and number of other users this user is following to
- topRepoCommitsTable: This table stores total number of commits and number of files in a repository

cs441project toprepolanguageable	
🔑	repoName : varchar(500)
🔑	repoID : bigint(20)
🔑	language : varchar(20)
#	numberOfLines : bigint(20)
#	numberOfFiles : int(11)

cs441project usertable	
🔑	userName : varchar(500)
#	userID : bigint(20)
#	publicReposCount : int(11)
#	followersCount : int(11)
#	followingCount : int(11)
#	subscriptionsCount : int(11)

cs441project alllanguagerepotable	
🔑	repoName : varchar(500)
🔑	repoID : bigint(20)
👤	ownerUserName : varchar(500)
#	ownerID : bigint(20)
📅	createdAt : date
📅	updatedAt : date
#	watchersCount : int(11)
#	forksCount : int(11)
#	openIssue : int(11)
#	repoSize : bigint(20)

cs441project toprepopcommitstable	
🔑	repoName : varchar(500)
🔑	repoID : bigint(20)
#	numberOfCommits : int(11)
#	numberOfFiles : int(11)

### GitHub API queries:

These are the various GitHub API calls we used (with authentication). The response received was in JSON.

- To get repo details for 100 repos with language, created data range, size lower limit specified:

<https://api.github.com/search/repositories?q=language:csharp+created:2016-01-01..2016-01-05+size:%3E10000>

- To get user details:

<https://api.github.com/users/username>

- To get user subscriptions:

<https://api.github.com/users/username/subscriptions>

## How to run

- **OPTION 1(Run everything locally):**

1. Clone the repo and import the project into IntelliJ using SBT.
2. All classes can be run individually, GithubProcessor.scala and WebService.scala, as they are independent.

If starting from scratch, GithubProcessor.scala is the first class file that should be run when trying to setup the entire project and databases. MongoDbToMySQL.scala should be executed after MongoDB has been populated by GithubProcessor.scala.

Now run WebService.scala -> Right click and run WebService to run it locally and then use the web service url in your browser (<http://localhost:8080/>). The web service can be run directly if MySQL has already been setup.

**Note:** While running the scala programs for the first time, IntelliJ might show the error, "Module not defined". You can go to Run->Edit Configurations->Use classpath of module and specify the module there. And then rerun the program.

- **OPTION 2(Run web service and other programs in the cloud):**

1. Copy build.sbt, and /src/main/ to a folder in your google cloud VM. Run using SBT (From within the folder):
2. Run the commands: **sbt compile**

**sbt run**

**Note:** In our build.sbt we have specified the mainClass as WebService.scala. To run other classes specify them here in build.sbt and replace WebService.scala

After the web service is created, the URL to access it is <http://localhost:8080> (if web service is run locally OR use your google cloud external IP)

- **Different rest calls that can be made to the web service.** Instructions to use the web service created are given below.(These instructions can also be found when you browse to <http://localhost:8080> using a browser)

**Note:** If simply clicking the URL doesn't work, copy it and paste in your browser.

### Examples:

Below queries fetch the top repository owners for GitHub in our database, according to different sorting criteria. (Number of users to be displayed can be specified)

<http://localhost:8080/?topUsers=5&sortBy=followersCount>  
<http://localhost:8080/?topUsers=5&sortBy=followingCount>  
<http://localhost:8080/?topUsers=5&sortBy=publicReposCount>  
<http://localhost:8080/?topUsers=5&sortBy=subscriptionsCount>

To get the top repositories according to popularity (defined as a function of watchers and forks counts) (number of users to be displayed can be specified):

<http://localhost:8080/?topRepo=10>

To get average lines of code per language (number of languages to be displayed can be specified):

<http://localhost:8080/avgLocPerLanguage>

To get top languages, selected by number of repositories in each language (number of languages to be displayed can be specified):

<http://localhost:8080/?topLanguages=5>

### **Examples for Repository Recommendation:**

The user enters a repository name, and we try to recommend some similar repositories.

<http://localhost:8080/?getRecommendation=sprintnba>  
[http://localhost:8080/?getRecommendation=deep\\_recommend\\_system](http://localhost:8080/?getRecommendation=deep_recommend_system)  
<http://localhost:8080/?getRecommendation=emoji-mart>

## Results Analysis

Now that the data was downloaded and stored in the database, next goal was to find some interesting patterns from the collected Big Data. We have downloaded metadata of 174K repositories in different languages and user data for 27K users from GitHub APIs.

Our analysis was divided majorly into two sections:

- RapidMiner based
- WebService based

### RapidMiner based

Different patterns we found after analysis are as following:

1. Language vs number of repositories

After studying distribution of languages across 174K repositories, we found out that Java, HTML, JavaScript were the most popular languages whereas Scala and Go were the least contributed languages.

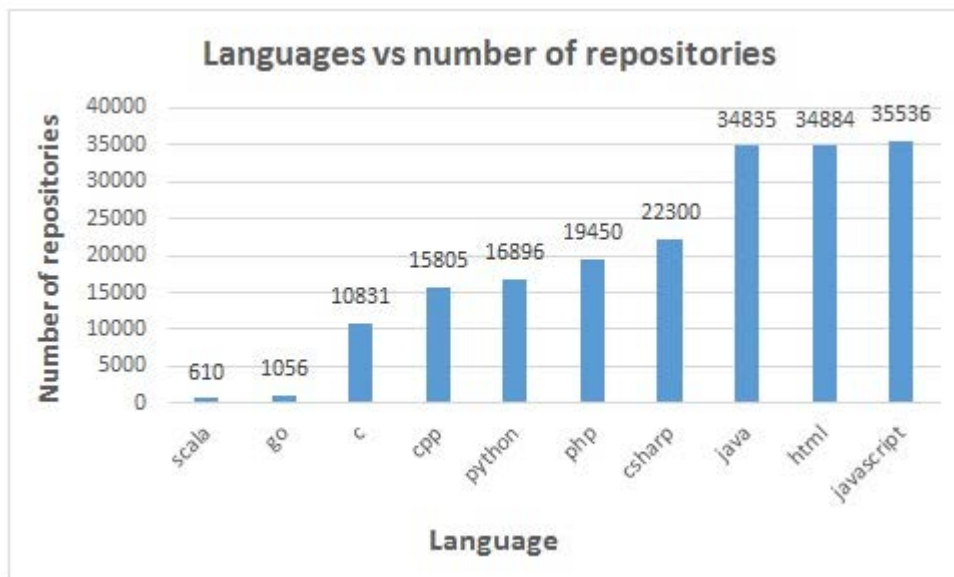


Fig. 3

After observing majority of the repositories, we found that some of the repositories were just sample repositories or with very few programs written as toy programs. So, to study more useful repositories, we decided to study popular repositories. We then selected the repositories whose fork count is greater than 4 and size is greater than 10 MB. This gave us some important repositories to study for. Further analysis of these repositories is shown below.



2. Distribution of languages across popular repositories  
Some projects are written completely in one language whereas some projects have combination of different languages. Fig 4 shows distributions of languages used in different projects.

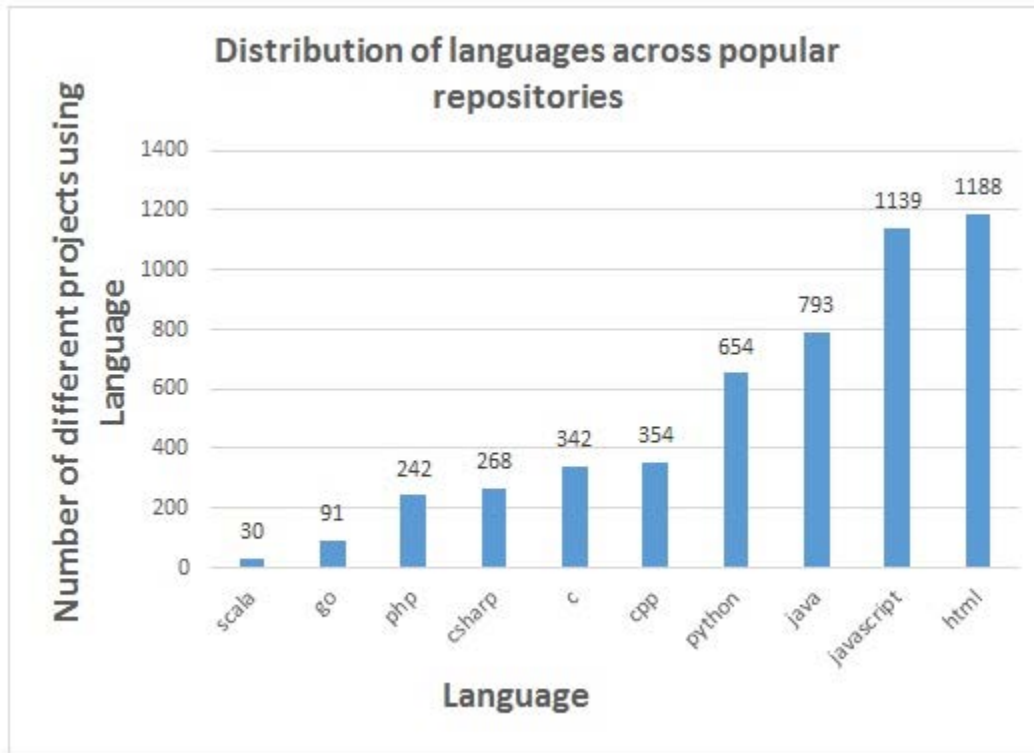


Fig. 4

3. Number of files in each language across popular repositories  
Fig. 5 shows number of files, scaled down by 100 in each language. We can see that Java, JavaScript continue to lead the distributions and Scala being on the lower end.

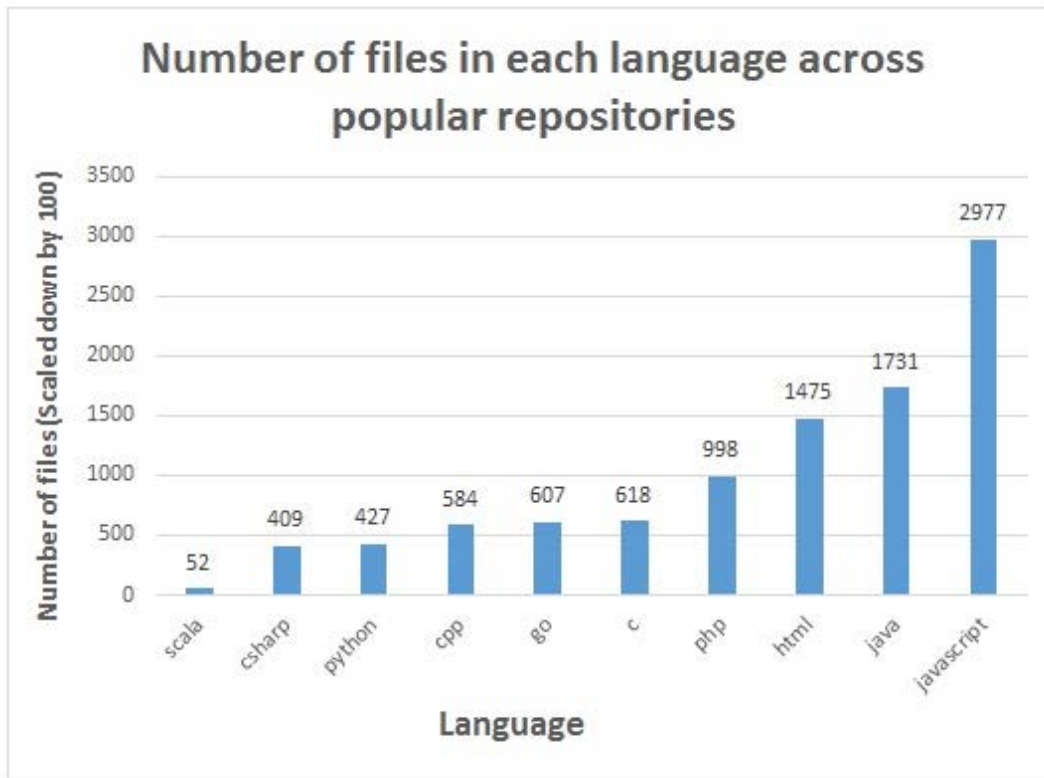


Fig. 5

4. Number of lines of code written in each language across popular repositories  
Next interesting graph in Fig. 6 tells a lot about the compact nature of programs across languages. Scala, Python, C# languages have very small lines of code versus traditional JavaScript code is very huge.

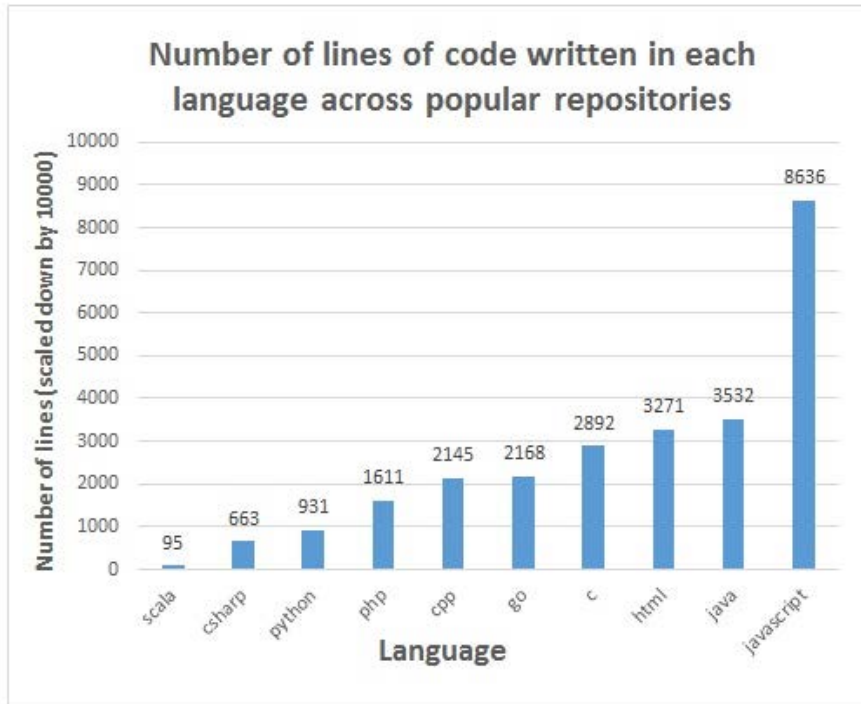


Fig. 6

5. Total number of open issues across popular repositories

As we have been observing from the previous graphs that Java, JavaScript, HTML being most popular languages, having large codebase and huge number of files, no wonder we expect a lot of open issues in these languages. Fig. 7 shows number of open issues per language across popular repositories.

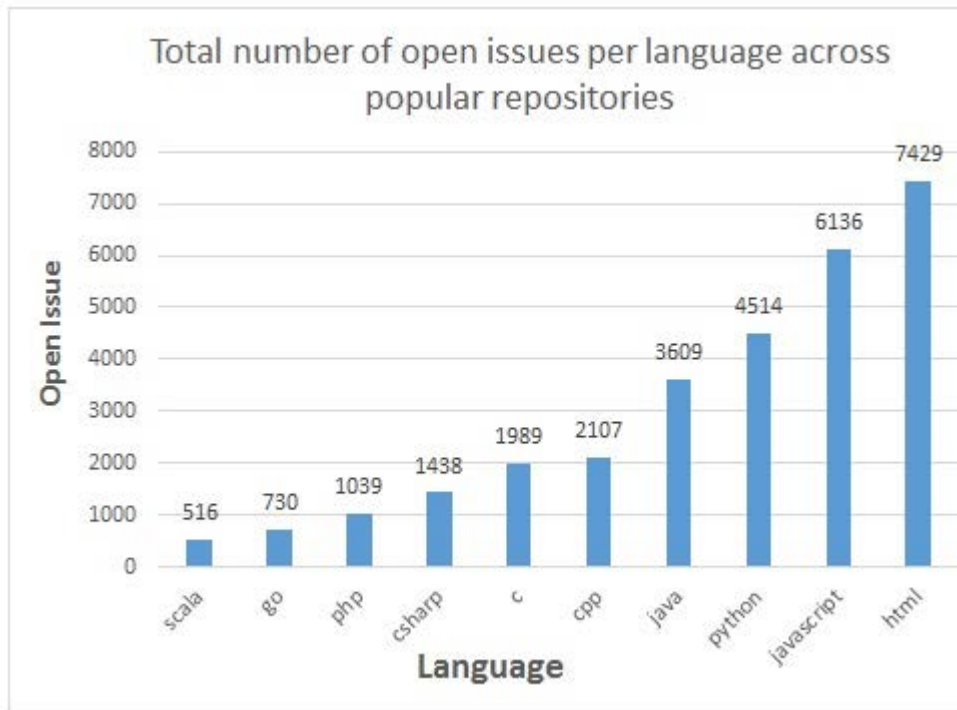


Fig. 7

#### 6. Popularity vs open issues

Apart from studying distribution of language and finding related patterns, we were trying to relate popularity of a project with number of open issues. We thought as a project becomes more popular, more people start using it and find bugs in them and report issues, so we plotted a graph of popularity of a project versus open issues in it. Here popularity is a function of followers count and fork count of the repository.

Unfortunately we didn't find any pattern in them. Fig. 8 shows the graph

Graph of Popularity vs Open Issues

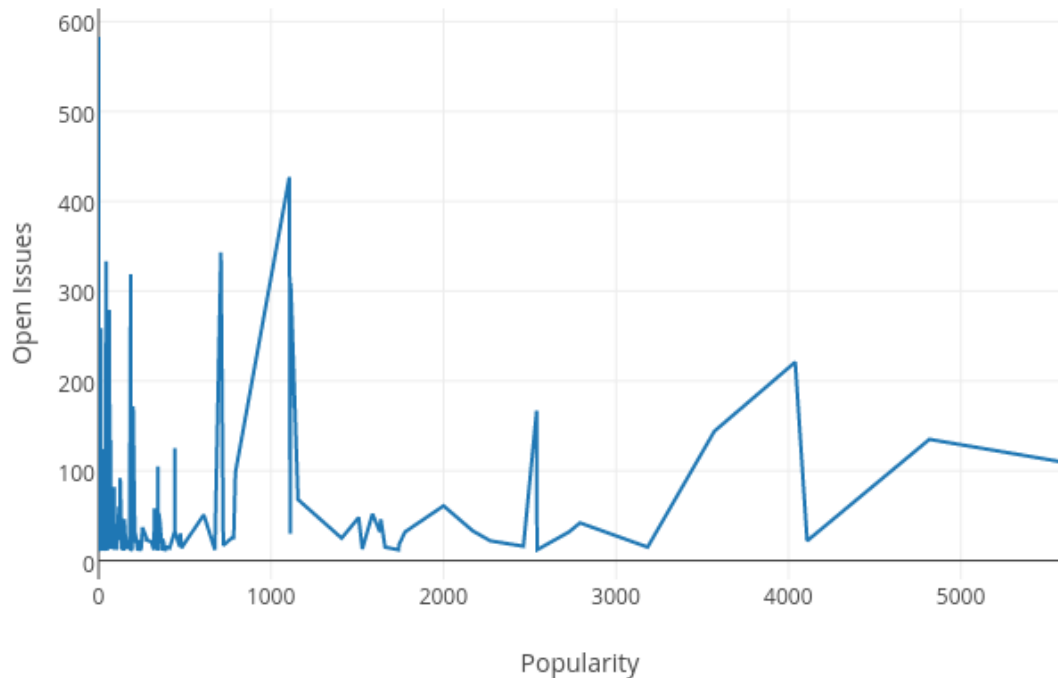


Fig.

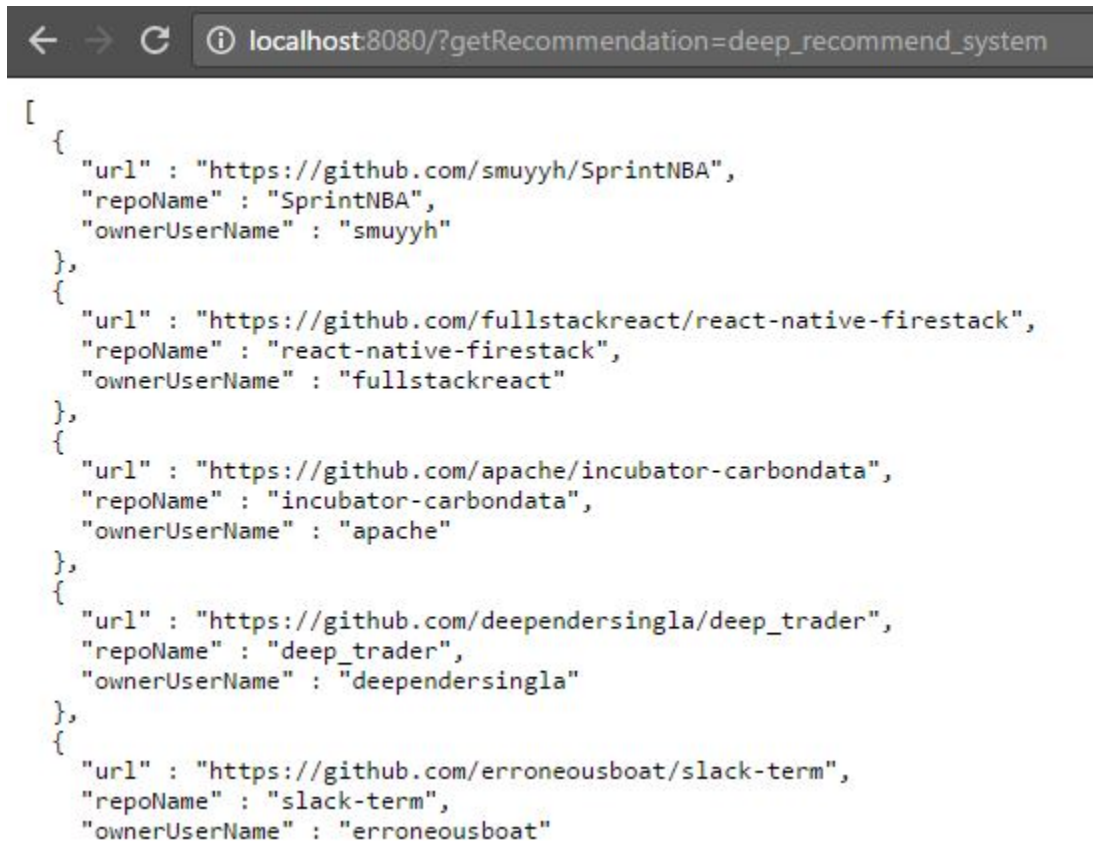
## Web Service based

We have created a WebService which any user can access and retrieve results for some interesting queries. This WebService queries to our MySQL database and fetches the results in JSON format so users can view the results online as well as they can access the results from command line or scripts too.

Following are some of the queries which we have implemented.

### 1. Recommend repositories

This REST API call returns recommended repositories for a given repository. The recommendation is based on language of repository, its popularity (function of fork count and followers count) and size of the repository. Fig. 9 shows sample recommendations for repository 'deep\_recommend\_system'



```
[
  {
    "url" : "https://github.com/smuyyh/SprintNBA",
    "repoName" : "SprintNBA",
    "ownerUserName" : "smuyyh"
  },
  {
    "url" : "https://github.com/fullstackreact/react-native-firestack",
    "repoName" : "react-native-firestack",
    "ownerUserName" : "fullstackreact"
  },
  {
    "url" : "https://github.com/apache/incubator-carbondata",
    "repoName" : "incubator-carbondata",
    "ownerUserName" : "apache"
  },
  {
    "url" : "https://github.com/deependersingla/deep_trader",
    "repoName" : "deep_trader",
    "ownerUserName" : "deependersingla"
  },
  {
    "url" : "https://github.com/erroneousboat/slack-term",
    "repoName" : "slack-term",
    "ownerUserName" : "erroneousboat"
  }
]
```

Fig. 9

## 2. Top n users

This REST API call returns top n popular users in the system based on their followers count, number of public repositories. User also has an option to sort the results by following count, followers count. Fig 10 shows a sample API call which returns top 5 users and sorts it by following count.



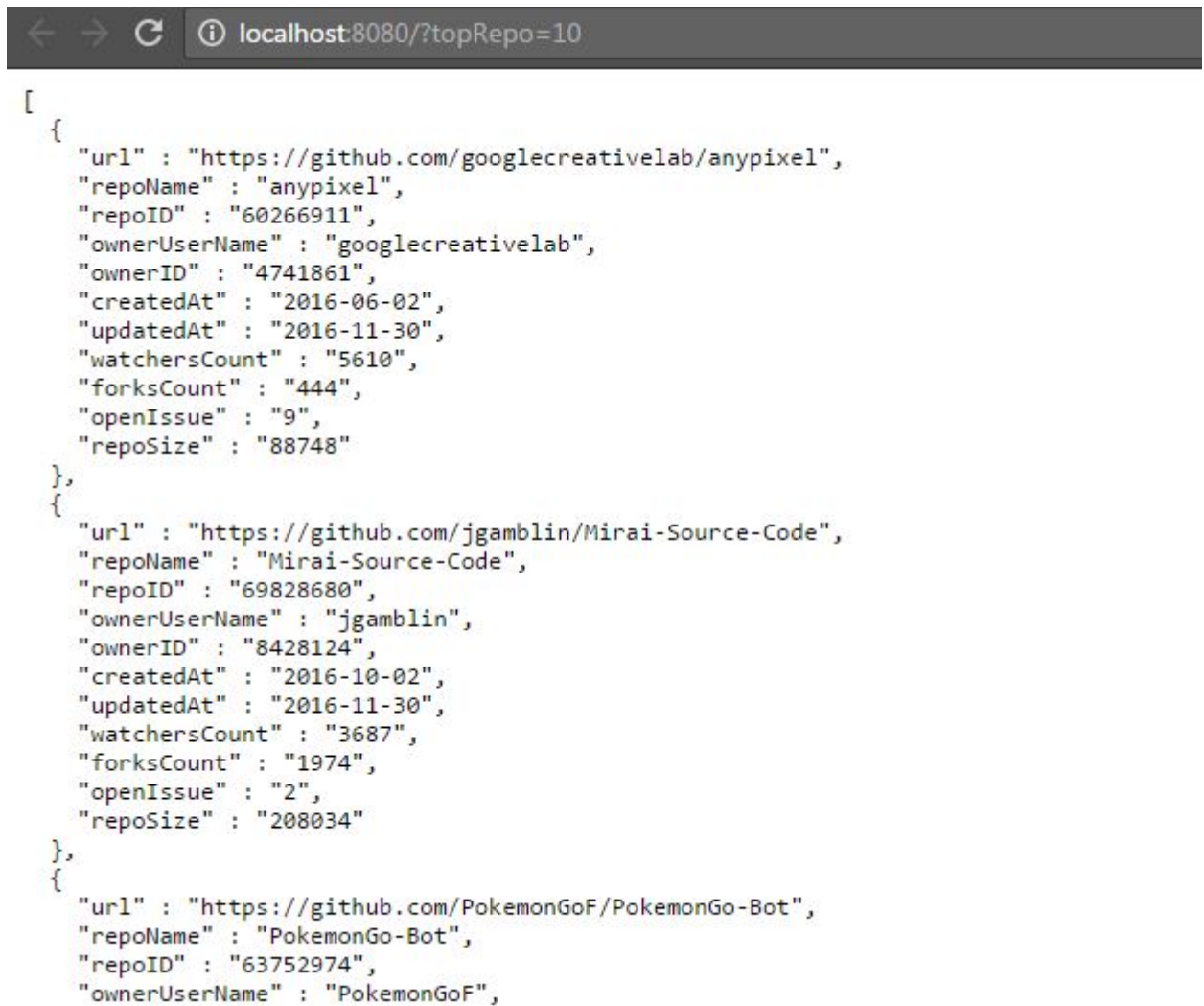
The screenshot shows a web browser address bar with the URL `localhost:8080/?topUsers=5&sortBy=followingCount`. Below the address bar, a JSON array of four user objects is displayed. Each object contains the following fields: `url`, `userName`, `userID`, `publicReposCount`, `followersCount`, `followingCount`, and `subscriptionsCount`.

```
[
  {
    "url" : "https://github.com/happyqq",
    "userName" : "happyqq",
    "userID" : "534200",
    "publicReposCount" : "2854",
    "followersCount" : "54",
    "followingCount" : "5462",
    "subscriptionsCount" : "30"
  },
  {
    "url" : "https://github.com/sirinath",
    "userName" : "sirinath",
    "userID" : "637415",
    "publicReposCount" : "117",
    "followersCount" : "83",
    "followingCount" : "4149",
    "subscriptionsCount" : "30"
  },
  {
    "url" : "https://github.com/caomw",
    "userName" : "caomw",
    "userID" : "8454286",
    "publicReposCount" : "4875",
    "followersCount" : "44",
    "followingCount" : "1553",
    "subscriptionsCount" : "30"
  },
  {
    "url" : "https://github.com/weimingtom",
    "userName" : "weimingtom",
    "userID" : "1041542",
    "publicReposCount" : "1118",
    "followersCount" : "109",
    "followingCount" : "1252",
    "subscriptionsCount" : "30"
  }
]
```

Fig. 10

### 3. Top n repositories

This REST API call returns top n popular repositories in the system. Popularity of a repository is a function of number of forks and followers count of a repository. Fig. 11 shows sample call which returns top 10 repositories.



```
[
  {
    "url" : "https://github.com/googlecreativelab/anypixel",
    "repoName" : "anypixel",
    "repoID" : "60266911",
    "ownerUserName" : "googlecreativelab",
    "ownerID" : "4741861",
    "createdAt" : "2016-06-02",
    "updatedAt" : "2016-11-30",
    "watchersCount" : "5610",
    "forksCount" : "444",
    "openIssue" : "9",
    "repoSize" : "88748"
  },
  {
    "url" : "https://github.com/jgamblin/Mirai-Source-Code",
    "repoName" : "Mirai-Source-Code",
    "repoID" : "69828680",
    "ownerUserName" : "jgamblin",
    "ownerID" : "8428124",
    "createdAt" : "2016-10-02",
    "updatedAt" : "2016-11-30",
    "watchersCount" : "3687",
    "forksCount" : "1974",
    "openIssue" : "2",
    "repoSize" : "208034"
  },
  {
    "url" : "https://github.com/PokemonGoF/PokemonGo-Bot",
    "repoName" : "PokemonGo-Bot",
    "repoID" : "63752974",
    "ownerUserName" : "PokemonGoF",

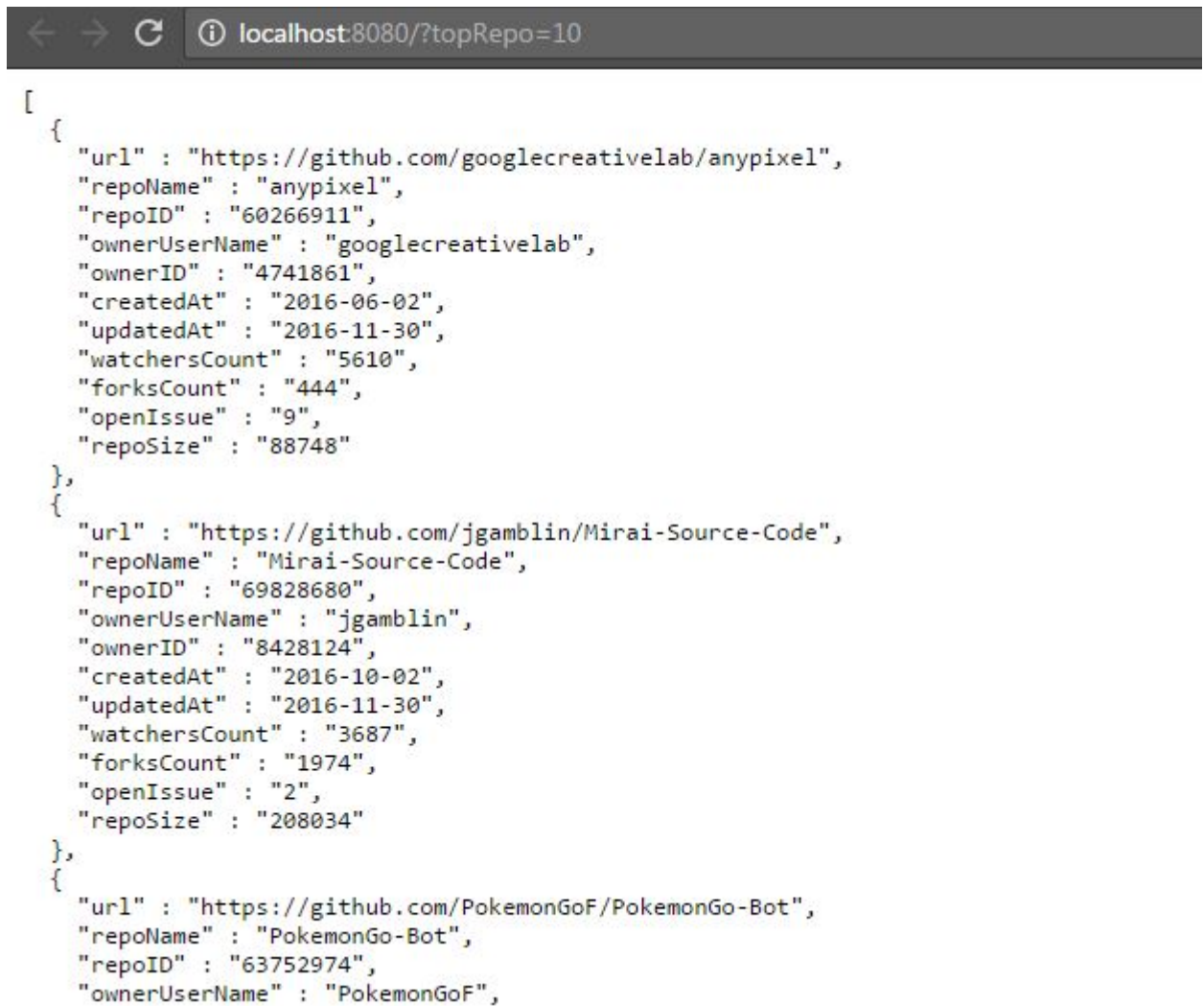
```

Fig. 11

#### 4. Top n languages

This REST API call returns top n languages in the system based on number of repositories of that language. Fig. 12 shows a sample result of top 3 languages.



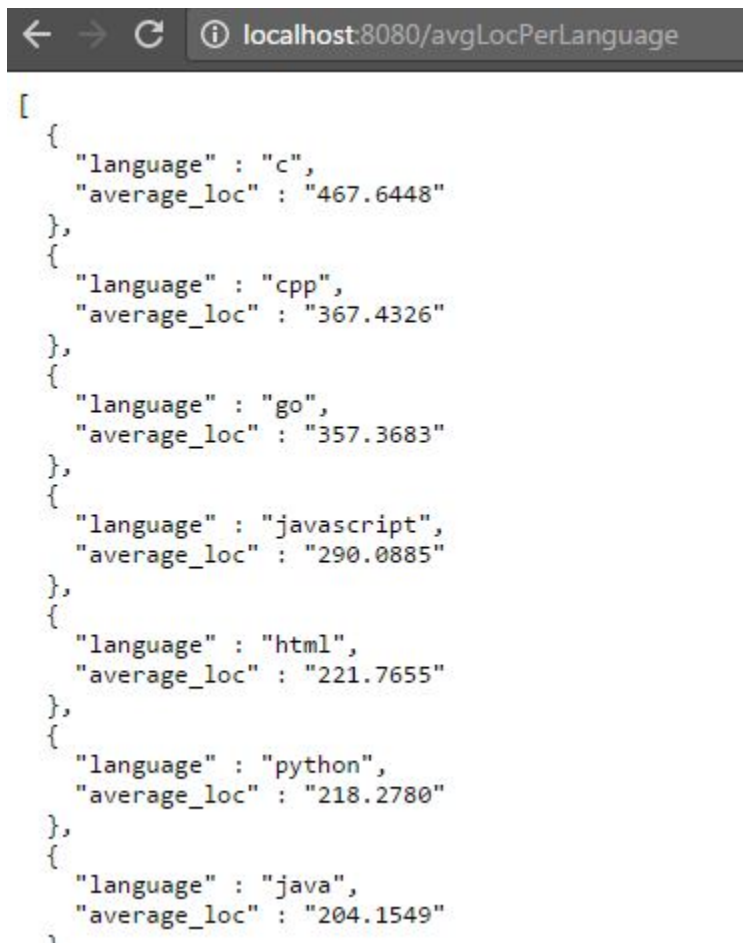


```
[
  {
    "url" : "https://github.com/googlecreativelab/anypixel",
    "repoName" : "anypixel",
    "repoID" : "60266911",
    "ownerUserName" : "googlecreativelab",
    "ownerID" : "4741861",
    "createdAt" : "2016-06-02",
    "updatedAt" : "2016-11-30",
    "watchersCount" : "5610",
    "forksCount" : "444",
    "openIssue" : "9",
    "repoSize" : "88748"
  },
  {
    "url" : "https://github.com/jgamblin/Mirai-Source-Code",
    "repoName" : "Mirai-Source-Code",
    "repoID" : "69828680",
    "ownerUserName" : "jgamblin",
    "ownerID" : "8428124",
    "createdAt" : "2016-10-02",
    "updatedAt" : "2016-11-30",
    "watchersCount" : "3687",
    "forksCount" : "1974",
    "openIssue" : "2",
    "repoSize" : "208034"
  },
  {
    "url" : "https://github.com/PokemonGoF/PokemonGo-Bot",
    "repoName" : "PokemonGo-Bot",
    "repoID" : "63752974",
    "ownerUserName" : "PokemonGoF",
  }
```

Fig. 12

## 5. Average LOC per language

This REST API call returns average lines of code per language present in the system. Fig. 13 shows a snippet of result for this query.



The image shows a web browser window with the address bar displaying 'localhost:8080/avgLocPerLanguage'. The main content area shows a JSON array of objects, each representing a programming language and its average lines of code (LOC). The languages listed are C, C++, Go, JavaScript, HTML, Python, and Java, ordered from highest to lowest average LOC.

```
[
  {
    "language" : "c",
    "average_loc" : "467.6448"
  },
  {
    "language" : "cpp",
    "average_loc" : "367.4326"
  },
  {
    "language" : "go",
    "average_loc" : "357.3683"
  },
  {
    "language" : "javascript",
    "average_loc" : "290.0885"
  },
  {
    "language" : "html",
    "average_loc" : "221.7655"
  },
  {
    "language" : "python",
    "average_loc" : "218.2780"
  },
  {
    "language" : "java",
    "average_loc" : "204.1549"
  }
]
```

Fig. 13

# Application testing

## 1. Unit Testing (Using ScalaTest)

- MongoDBOperationAPIsTest.scala: This test is a unit test for testing methods in MongoDBOperationAPIs.scala. MongoDBOperationAPIs.scala is a class used to stream JSON data to MongoDB. So, this tests checks whether a random incorrect name entered for a collection exists in MongoDB. The response is then checked for passing this test case.
- MySQLOperationAPIsTest.scala: This test is a unit test for testing methods in MySQLOperationAPIs.scala. MySQLOperationAPIs.scala is a class used to stream write data to MySQL. In this test case, we use methods in MySQLOperationAPIs class to insert some random data into a table which stores repository details, and later check whether the table returns results for the inserted data. We also check using another method in MySQLOperationAPIs, whether a random incorrect table name is found in the same table in our MySQL database. Accordingly, responses are matched against predefined values to pass these cases.

## 2. Integration Testing (Using ScalaTest)

- WebServiceTest.scala: This test case initiates the web service and makes rest calls to the web service and check its response. If the response matches, these test cases pass. Apart from checking the response for calling the web service response, we also check the response when the service is called with some specific parameters. When such a rest call is made, the web service queries the MySQL database and fetches the response. This response is received by the test case which matches it against predefined values to pass this test case.

**Note:** While running the scala test programs for the first time, IntelliJ might show the error, "Module not defined". You can go to Run->Edit Configurations->Use classpath and SDK of module and specify the module there. And then rerun the test program.

**Note:** Sometimes IntelliJ automatically changes the test library for running the test cases. That might cause syntactical errors in our test programs. You can specify the test library again by removing the scala test library specified in build.sbt, and then putting it back again. The following ScalaTest library has been used:  
**libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" % Test**

- **Performance Testing: Load Tests (Using SoapUI)**

The load testing is performed using LoadUI NG of SOAPUI with varying virtual users ranging from 5 to 25. Each load test contains 5 different REST API calls:

1. <http://104.197.28.49:9200>
2. <http://104.197.28.49:9200/?topUsers=15&sortBy=followersCount>

3. <http://104.197.28.49:9200/?topRepo=12>
4. <http://104.197.28.49:9200/avgLocPerLanguage>
5. <http://104.197.28.49:9200/?topLanguages=8>

Each load test was performed for 3 minutes and the number of virtual users (load) varying as per the maximum number of virtual users for that load test.

The cumulative result comparing different response time vs different load (virtual users):

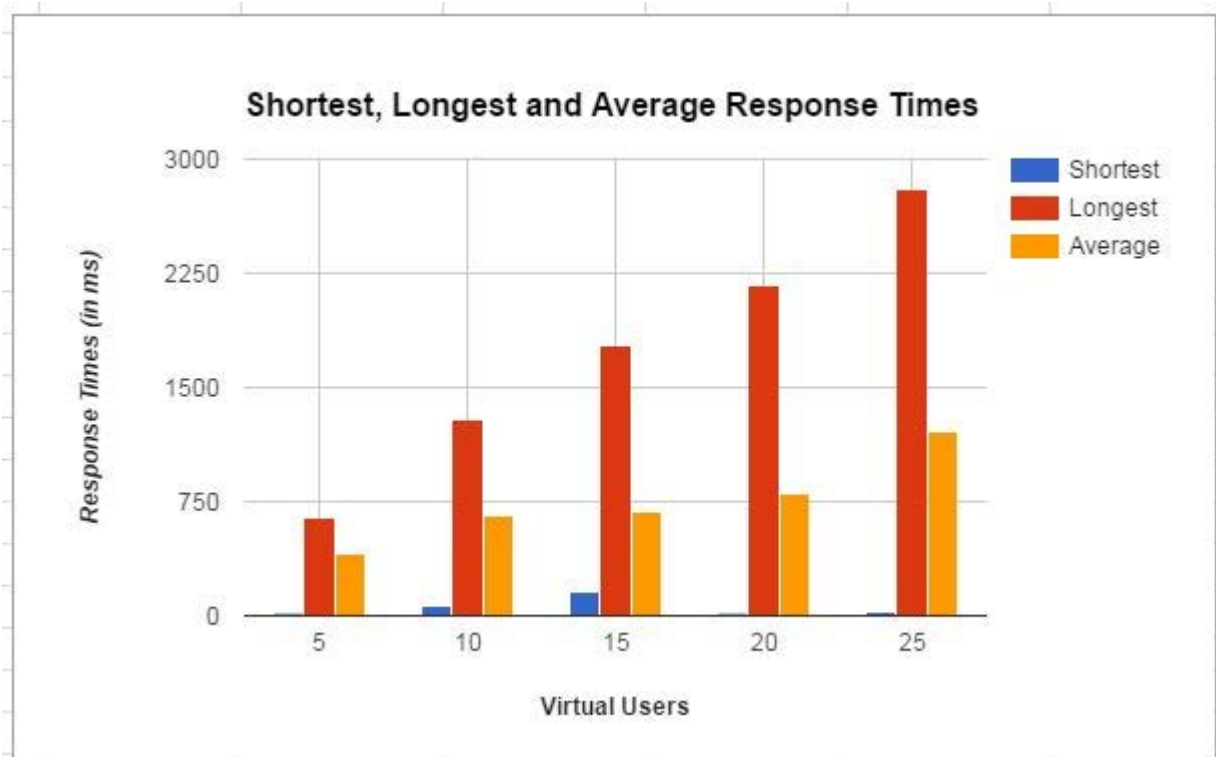


Fig. 14: The above graph compares the Shortest, Average and Longest Response Times for different loads (virtual users). The longest and average response time plays an important role where the graph shows a constant increase in response time with the increasing load. Each test was performed with the same set of 5 requests and varying load.

## Individual Test Results and Statistics

### 1. Load test with 5 virtual users:

#### Scenarios Summary

Scenario	Duration	Requests	Assertions	Failed
New Scenario	00:03:15	623	0	0

#### Overview Chart (All Scenarios)

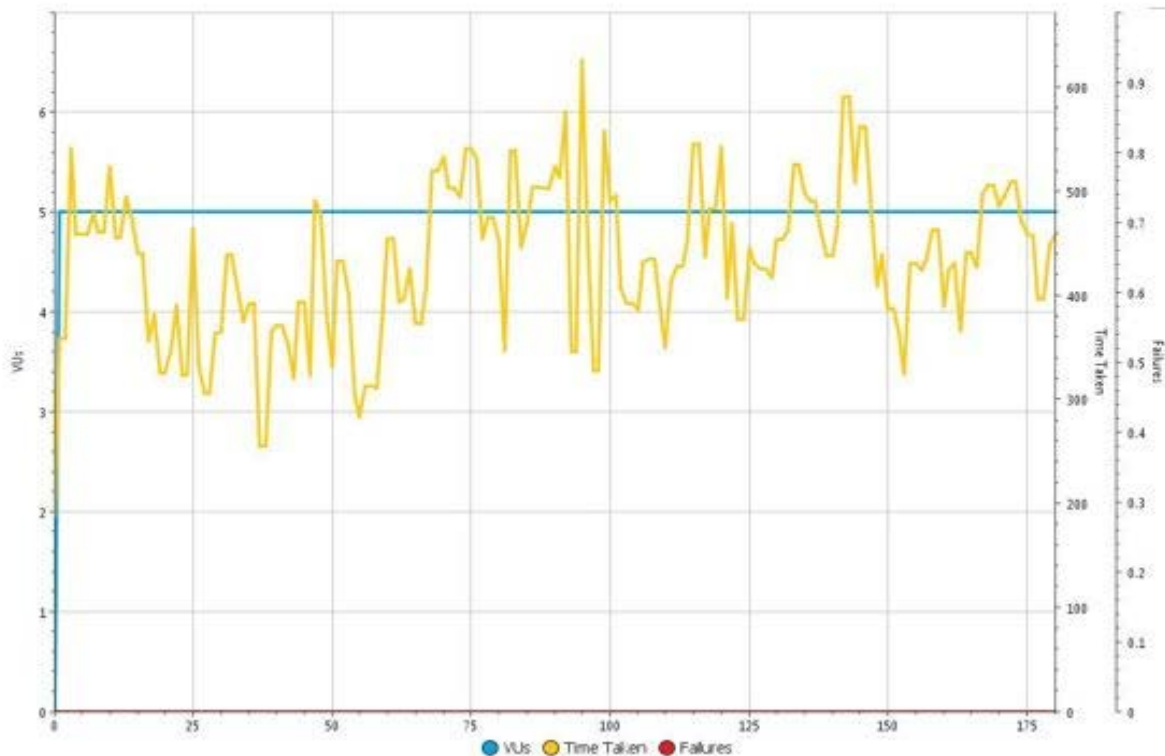


Fig. 15: The load test starts with number of users as 1 to 5 for 5 types of REST API calls and total number of requests = 623 and the total duration of simulation = 3 minutes (15 seconds being the warm up time for the test cases to load).

Longest response time (milliseconds)	649
Average response time (milliseconds)	408
Shortest response time (milliseconds)	16

## 2. Load test with 10 virtual users:

### Scenarios Summary

Scenario	Duration	Requests	Assertions	Failed
New Scenario	00:03:15	1059	0	0

### Overview Chart (All Scenarios)

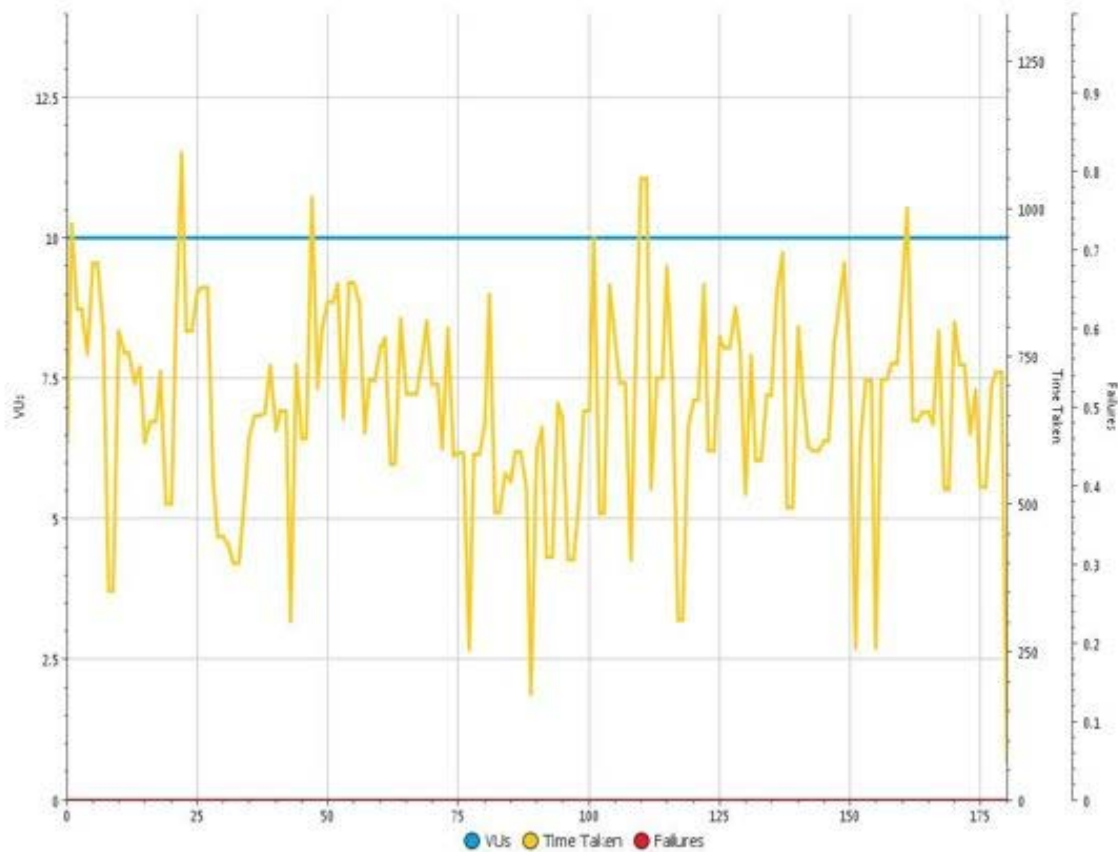


Fig. 16: The load test starts with number of users as 1 to 10 for 5 types of REST API calls and total number of requests = 1059 and the total duration of simulation = 3 minutes (15 seconds being the warm up time for the test cases to load).

Longest response time (milliseconds)	1287
Average response time (milliseconds)	661
Shortest response time (milliseconds)	64

### 3. Load test with 15 virtual users:

## Scenarios Summary

Scenario	Duration	Requests	Assertions	Failed
New Scenario	00:03:15	1572	0	0

## Overview Chart (All Scenarios)

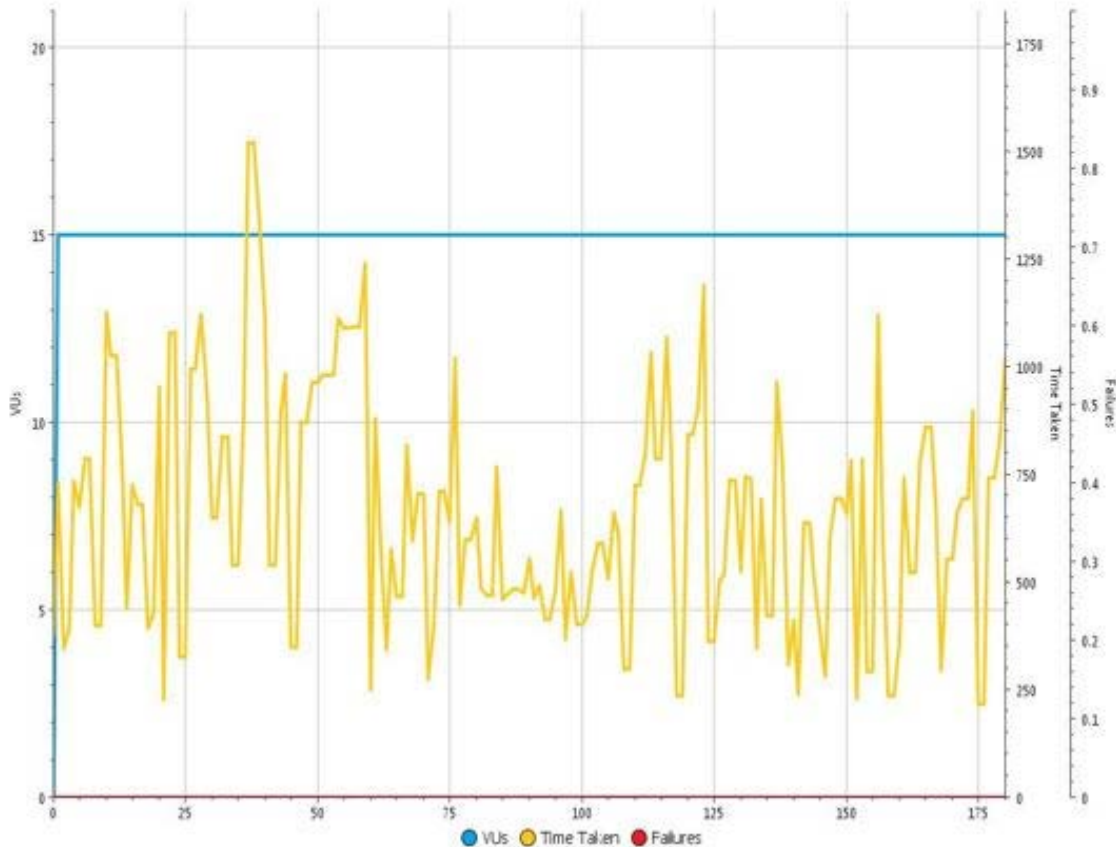


Fig. 17: The load test starts with number of users as 1 to 15 for 5 types of REST API calls and total number of requests = 1572 and the total duration of simulation = 3 minutes (15 seconds being the warm up time for the test cases to load).

Longest response time (milliseconds)	1770
Average response time (milliseconds)	685
Shortest response time (milliseconds)	153



#### 4. Load test with 20 virtual users:

### Scenarios Summary

Scenario	Duration	Requests	Assertions	Failed
New Scenario	00:03:15	1965	0	0

### Overview Chart (All Scenarios)

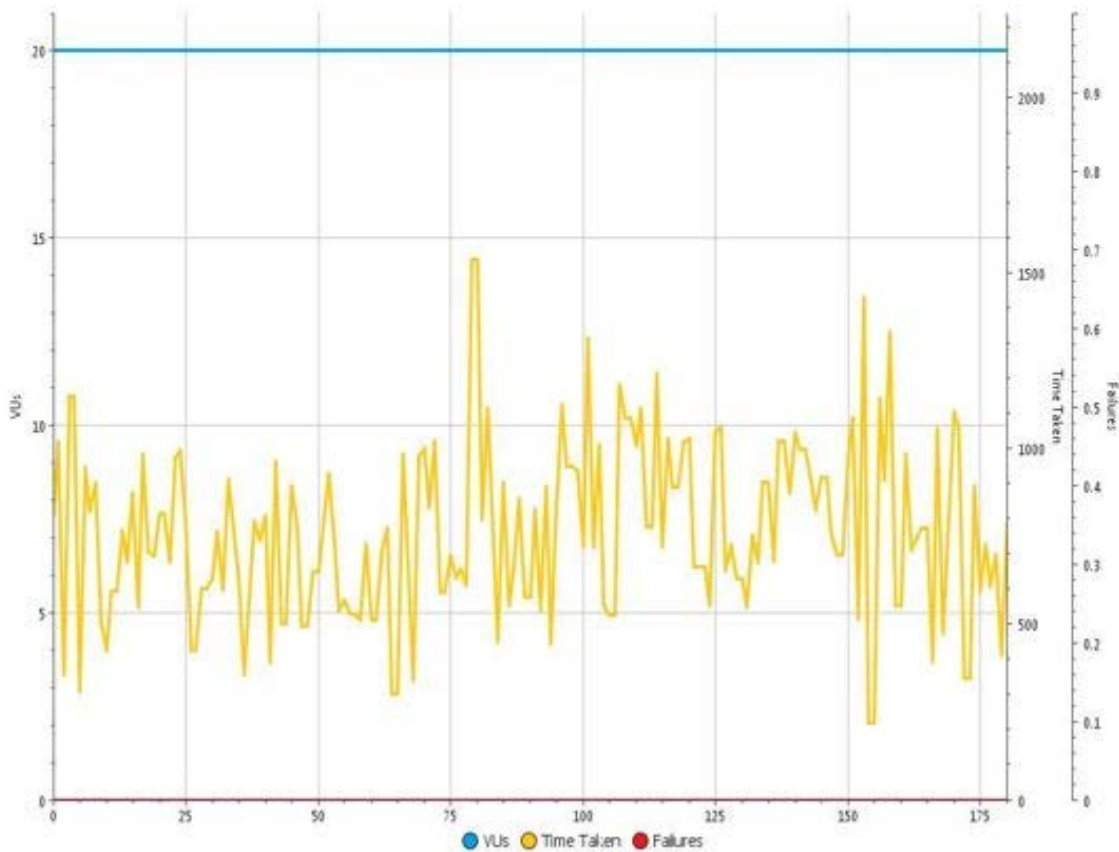


Fig. 18: The load test starts with number of users as 1 to 15 for 5 types of REST API calls and total number of requests = 1965 and the total duration of simulation = 3 minutes (15 seconds being the warm up time for the test cases to load).

Longest response time (milliseconds)	2170
Average response time (milliseconds)	803
Shortest response time (milliseconds)	21



## 5. Load test with 25 virtual users:

### Scenarios Summary

Scenario	Duration	Requests	Assertions	Failed
New Scenario	00:03:15	2010	0	0

### Overview Chart (All Scenarios)

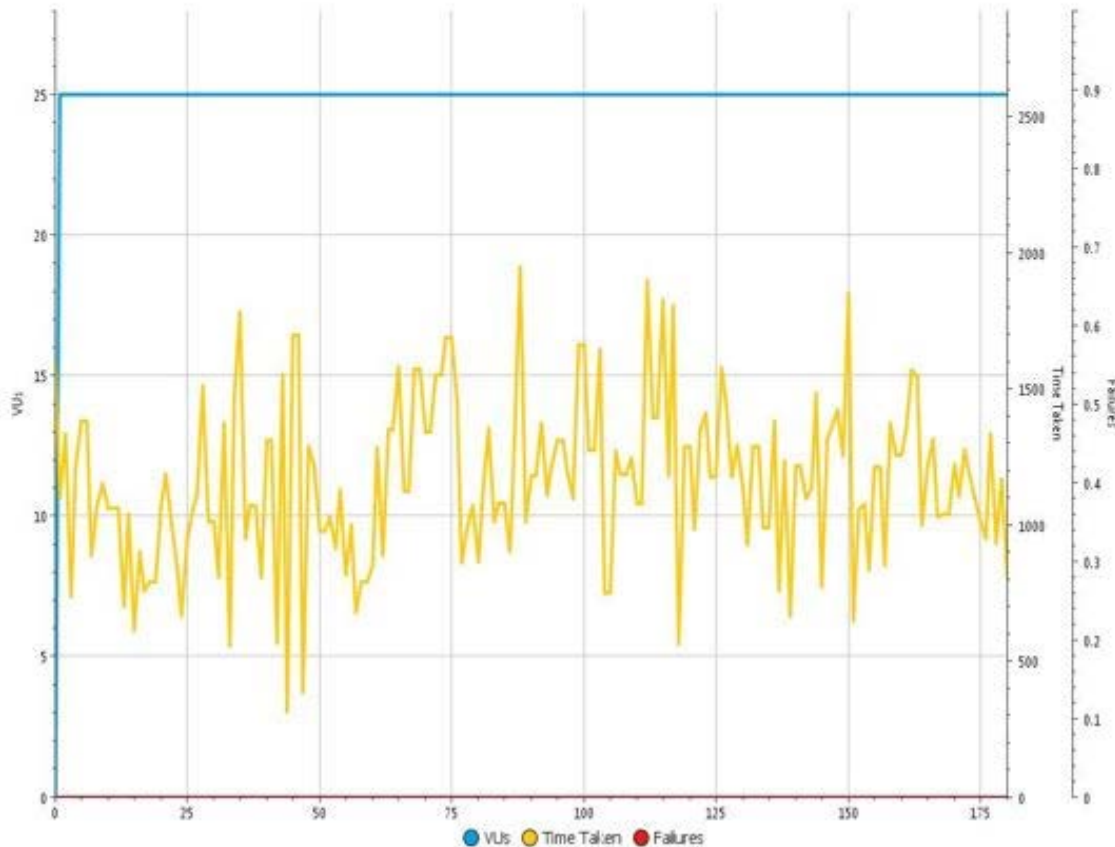


Fig. 19: The load test starts with number of users as 1 to 15 for 5 types of REST API calls and total number of requests = 2010 and the total duration of simulation = 3 minutes (15 seconds being the warm up time for the test cases to load).

Longest response time (milliseconds)	2801
Average response time (milliseconds)	1205
Shortest response time (milliseconds)	30

## Limitations

While we had planned many things in the brainstorming sessions initially, as we started working on the implementation we came across some limitations. This section sheds some light on them and how did we overcome them.

1. Limit on number of searches per minute  
With every API key GitHub allows only 30 search queries per minute are allowed so we limited our repository study to 10 languages and 174K repositories
2. Limited information and cloning large repositories  
Out of 174K repositories which we downloaded from GitHub API, we had very limited information. For more information like number of different languages, number of commits, and updation time etc. we had to clone the entire repository using JGit. As this was consuming a lot of bandwidth (average 1MB for 174K repositories), we decided to do this in-depth analysis for some popular repositories. We kept fork count greater than 4 and minimum repository size equal to 10MB as a criterion for popularity and cloned close to 2400 popular repositories.
3. Limit on number of user information per hour  
GitHub APIs limit number of user information requests to 5000 per hour so we decided to limit our user information by selecting users of only one language. We downloaded user information for 27K users.
4. Scala test  
Scala test runs only from some specified IPs, which we have added in the list of Google Cloud SQL. So if you plan to run it from your local IP, please provide that IP to us and we will add it to the trusted IPs

## Future Work

1. Because of the GitHub API limits, we were only able to get user related data for about 25K repository owners. Given more time, we can download data for a lot more users, and do better analysis.
2. Because all the repository metadata that could be relevant is not available through GitHub API calls, we used JGIT to get this information. We had to locally clone these repositories to get info like number of files, lines of code, lines of code per language used in one repo etc. Due to restriction of bandwidth, we were only able to download and clone about ~3K repositories (which we considered relevant) and get their extra metadata. In the future, we can download this info for a lot more repos, and conduct better pattern recognition.
3. In the future, we can implement a lot more machine learning if our user and repo data increases. Currently we have used RapidMiner mostly. We can work with Weka as well as use the popular ML libraries in Python.

## Known issues

1. If you get the exception *"org.bson.codecs.configuration.CodecConfigurationException: Can't find a codec for class org.bson.BsonDecimal128"* while running the program, try commenting/uncommenting the line `"libraryDependencies += "commons-codec" % "commons-codec" % "1.9"` in build.sbt.

We don't have concrete reasoning for this at this point, but we know the workaround: toggle the state of the line in build.sbt (comment -> uncomment or vice versa)

## Learnings

1. Right data source makes the difference

At the beginning, we were trying to get the data from BLACKDUCK APIs. With BLACKDUCK API, there are many limitations [Technology Stack -> BLACKDUCK section], so we decided to study GitHub API and that was one of the best decisions we have taken because we could get a lot of data to call it Big Data and the information, attributes were very uniformly structured and present across all the repositories.

2. Continuous feedback on work is important

We were doing a lot of brainstorming at every step and were trying to implement, try many

different things as the project goal was not clear. To confirm if we are on the right track, we met TA couple of times, discussed with professor and posted our queries on

### 3. Importance of Documentation at Given Time

You should document your work immediately after you complete it. If you have better idea of what the outcome will be, better write it (or at least take notes of them) before you do it, so that at last moment you need not worry about how you did that and if you are missing anything. In summary, basic design should be prepared before actually starting the work.

From this beginning of this project, we always used to keep a note of all the brainstorming sessions, To-Do items, our doubts and limitations. This helped to prepare our report very well, track the progress and not spend time doing random trials as we had very small amount of time to finish this project.

## **Attached Files Description**

1. Documents directory
  - a. Load\_Test\_Reports\_Screenshots  
Screenshots taken during SoapUI load tests
  - b. WebService\_Screenshots  
Screenshots of different REST API calls to our WebService

Other files include schema for MongoDB language, user collection, MySQL schema and sample JSON response from GitHub API

2. Graphs  
Different graphs generated from RapidMiner analysis
3. Report  
Final copy of our report
4. Src  
source directory
5. GithubRepoAnalysis.log  
Log of a sample run using SLF4J
6. CourseProjectCS441.pdf  
Course project description

## References

1. UIC Logo:  
<http://logos.uic.edu/DOWNLOAD.CGI?document=COL.ENG.CSCI.LOCKB.SM.RED.PNG>
2. OpenHub repository API  
[https://www.openhub.net/projects/45001.xml?api\\_key=c3943bda503b24b9ed76ba00add525ecd330720aa437f82fa3bc4cbeab330b7b](https://www.openhub.net/projects/45001.xml?api_key=c3943bda503b24b9ed76ba00add525ecd330720aa437f82fa3bc4cbeab330b7b)
3. GitHub API  
<https://api.github.com>