1TD075 Data Engineering II

UPPSALA
UNIVERSITET

Github Analytic System using a Streaming Framework

Group 17

Hanna Ericson Kim Kuruvilla Mathews Kirill Pavlov Adam Ross

December 9, 2024

Contents

1 Introduction

In this project, we create a GitHub analytic streaming framework to answer the following questions that would otherwise not be possible when directly fetching from the GitHub API interface:

1. What are the top N most frequently updated GitHub projects based on commit count?
2. What are the top N programming languages based on the number of projects developed?
3. What are the top N programming languages used by repositories following the test-driven development (TDD) approach?
4. What are the top N programming languages used by repositories following the TDD and DevOps approaches?

We propose a graphical web app for dynamically answering each question implementing a scalable distributed infrastructure algorithm for parallelized continuous integration/continuous deployment (CI/CD) fetching and analysing of repository data from the GitHub API within a given date range.

In the following sections, we discuss related work, provide a comprehensive description of the system, perform scalability experiments, and present and analyse results with graphical representations.

2 Related Work

GitHut is a previous work that provides statistics about language popularity in GitHub [1]. The statistics that can be queried are top languages, top licenses and number of pull requests per language in a given time span. These statistics can provide answers to the first two questions, however they do not provide statistics related to development approaches for the remaining two questions.

The data used in GitHut is collected from the GitHub Archive dataset via Google BigQuery. The GitHub Archive is a project that uses the GitHub API to collect and store data over time [2]. Google BigQuery is a PaaS that is used for handling and analyzing big data, which has the GitHub Archive available as a dataset [3].

No prior studies have been found that specifically addressed the question answered in this project. However, there exist plenty analysis of git hub commits and git hub data in aspect to their contents in other ways, such as the project successes. One such example is *A large scale study of programming languages and code quality in github* by Baishakhi Ray et. al.(2014) [4]. In this study, a large dataset comprising 728 projects and 17 programming languages - not including CSS, Shell script, and Vim script - was collected from GitHub to investigate the impact of programming languages on software quality. The results indicate that not allowing type confusion yields better results than allowing it, and functional languages generally outperform procedural languages. Furthermore, within functional languages, static typing is slightly superior to dynamic typing.

3 Methodology

The GitHub analytics streaming infrastructure consists of three main modules, one for a scalable data streaming framework where the data is fetched from the GitHub API and preprocessed for statistical analysis, one for a scalable CI/CD cluster deployment where the statistical analysis is generated with web UI graphical representation of the user-requested results and a third client module for orchestrating dynamic contextualization of the two prior modules.

For the data streaming framework, Apache Pulsar is implemented like in the first lab, but the most recent version is experimented with for adaptability with the Python 3.11 source code used for implementing the infrastructure algorithm. The producer server iteratively publishes instances of a class to each consumer server at a given scale for fetching data from the GitHub API with a different, consecutive date range, GitHub token, and associated account username for parallel streaming. When finished API data fetching, each consumer inserts the preprocessed fetched data as a document to a MongoDB database collection.

For the CI/CD cluster deployment, the deployment server retrieves all preprocessed GitHub analytics document data inserted by each consumer server, validates, merges, and saves all the document data to a single .json file for git hook pushing to the production server. The production server computes statistical analysis algorithms for each question on the merged data uploaded from the .json file and displays this to a web UI for user-interactive graphical displaying of the top N results, where N is any value between 1 and 100 as instructed by the user.

3.1 Analytics Strategy

To answer each GitHub analytic question concisely and accurately without bias in as few API requests as possible with the timeliest responses was not a simple task considering the available API documentation is not designed to support answering such problems. The adopted approach we used consisted of initially searching for up to 1000 repositories for each day in a given date range that had at least one commit pushed on that day to allow for counting commits per day. To remove the likelihood of duplicate and irrelevant repositories in the sample dataset, the API search query excludes fork, mirror, archive, and private repositories, and although borderline biased, furthermore excludes repositories with sizes below 30 KB and less than 2 stars as anything below these are considered highly likely irrelevant for the prime research focus, and sorts the results by commit date because push events don't just consist of commit updates.

The first two questions are fairly straightforward and just require counting the relevant metrics within a given date range, which are the repository count per language and commit count for the repositories. Two metrics have been selected for each of the four questions, as the latter can be used as the decider in the case of draws between the former metric.

To answer the last two questions, we must determine what is required to classify a repository as following a test-driven or a DevOps approach. Our implementation uses keywords found in commit messages, topics, and workflow files. If a repository contains a keyword in any of the searched data it is classified as following the approach related to the keyword. This is a vague approach and will cause a lot of false positives and negatives. A more thorough approach would take too much time and resources requiring analysis of an entire source code commit history to determine if test cases are being created before each associated development, and relies on the assumption that developers would leave a commit history with enough information specifically for this.

3.2 UPPMAX Cloud Outage

In light of the ongoing UPPMAX cloud outage spanning some weeks, on and off, it has not been possible to implement the methodology in practice using multiple virtual machine servers. Instead, as a second resort, for both debugging the source code and verifying that the data streaming algorithm is theoretically achievable, a test suite has been run using GitHub Actions workflows to simulate the parallel GitHub analytics data fetching and preprocessing for statistical analysis from the GitHub API up to four processes acting as separate consumer servers with different sequential date ranges, GitHub tokens, and associated account usernames. However, unfortunately, once all debugging was completed, the allocated 3,000 minutes per month had expired, but not before starting a new four-consumer simulation workflow, resulting in the database being wiped without any simulation consumer completing their workflow and inserting new document data leaving us with just one .json file with two days of preprocessed data fetched from the GitHub API.

4 System Architecture

In order to address the four questions regarding this project, Git Hub data need to be collected, distributed and analysed to then be compiled and presented to the user.

The system architecture consists of a client machine that acts as an orchestrator. It coordinates the producer, which can create one to four consumer instances. These consumer instances are responsible for fetching data from the Git Hub API.

Once the consumers receives the data it send the results to a MongoDB database. The MongoDB is responsible for retrieving the data as needed and converting it to a JSON file. The JSON file is then sent to both the backend and frontend components of the system. The deployment machine orchestrates this process using Ansible, that deploys a production and development server. The production server is the MongoDB that retrieves the JSON file. The development server is the backend and frontend.

The backend and frontend components are coordinated by a production environment. The backend utilizes Flask, a popular web framework, for deploying the results. It processes the data received from the deployment machine and prepares it for presentation. The frontend component is responsible for displaying the processed data on a website, which provides a user-friendly interface for accessing and interacting with the results.

5 Results

5.1 Scalability Experiment

It has not been possible to run any scalability experiment using the UPPMAX virtual machines. However, the simulation of fetching data from the GitHub API in parallel with up to four consumers using the GitHub Actions workflow has provided an insight into what can be expected for speedup results, excluding any overhead from communications between the producer and consumers, although this can be considered minimal considering the initial design does not require more than one form of communication from each given there are no errors experienced. Being that each consumer is run in parallel with an approximate equal date range, the times have been fairly equal except for unforeseeable cases of the API rate limit being exceeded, which can cause a consumer to take longer than others. Unfortunately, although the results were printed to CLI, the workflows the print statements were saved to before the workflow minutes expired seem to have disappeared from the repository.

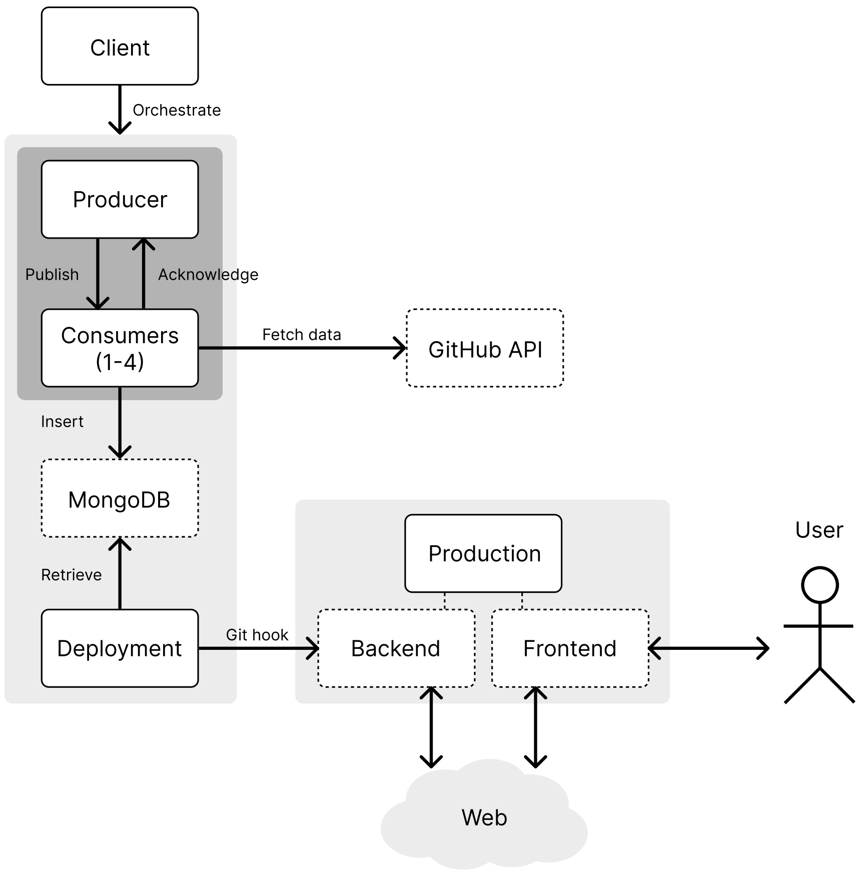


Figure 1: Overview of the distributed system architecture

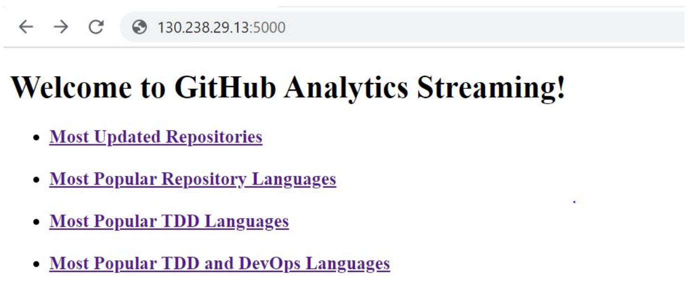
5.2 Top N Results

For analysing the result of Github Analytic System we have developed a flask application to be run on the production machine. Here we are fetching the data as JSON file which is the data combined from the different consumers using MongoDB. According to our architecture, we will be using Githooks to push the data from deployment to production where it can then be processed and displayed on the frontend flask application.

In the frontend flask application we have designed the following pages: an index page and four pages to show of Top N result for each questions. Once the flask application is started, we can navigate to the URL http://⟨floating-ip⟩:5000 by placing the floating IP of the production machine.

Then we can proceed to the result page for each of the question. For all these pages by default we are displaying the Top 10 result for each of the question which can be further changed to fetch Top N result by altering the textbox. Additionally we have displayed the date range for which the data has been generated.

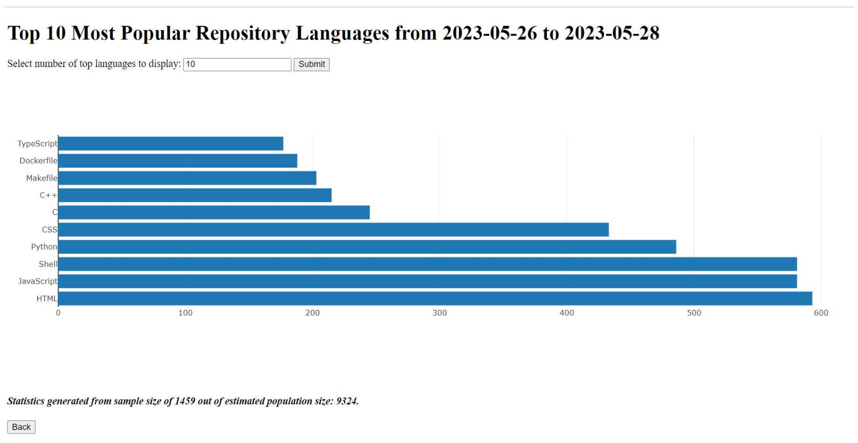
From 3 we can see that HTML is the most popular language from the date range that we have fetched Github analytics data. But we cannot say with certainty that HTML is the most popular language as HTML code or file can also components for other application using other languages like Flask. But based on the other top ranked languages we can definitely say that Javascript, Shell and Python are among the most popularly used languages.


130.238.29.13:5000

Welcome to GitHub Analytics Streaming!

- Most Updated Repositories
- Most Popular Repository Languages
- Most Popular TDD Languages
- Most Popular TDD and DevOps Languages

Figure 2: Flask Application - Index Page


Top 10 Most Popular Repository Languages from 2023-05-26 to 2023-05-28

Select number of top languages to display: 10 Submit

Statistics generated from sample size of 1459 out of estimated population size: 9324.

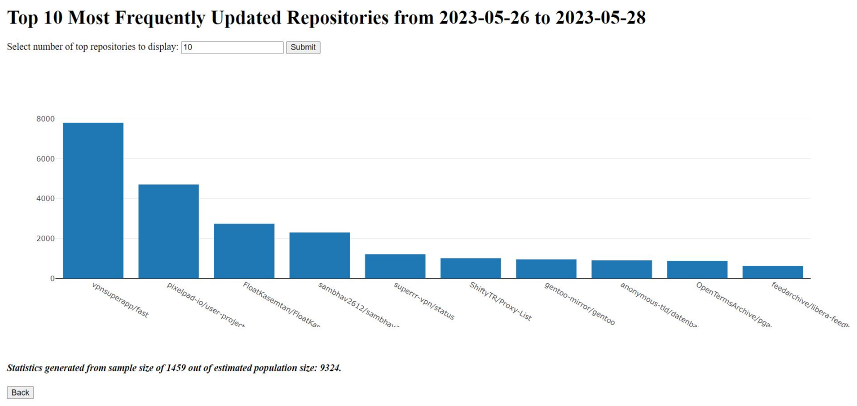
Back

Figure 3: Flask Application - Most popular language

Based on the graph 4, we can assume most of the frequently updated repositories are bot driven commits or automated commits which are generally used for logging mechanism or for database purposes. The amount of commits within such a short time span is not something that is humanly possible, therefore, not much valuable insights could be derived from this result.

Based on our analysis of TDD languages in Github repositories from figure 5, we can conclude that Test Driven Development approach is popularly deployed in web based development in Javascript and HTML, and for scripting based languages such as Python and Shell.

While analysing the most popular TDD and Devops from figure 6, we can conclude that there isn't much difference from the most popular TDD languages. It is clearly evident there is a close correlation between both TDD approach and Devops, where the top 5 languages are Shell, Javascript, Python, HTML and CSS.

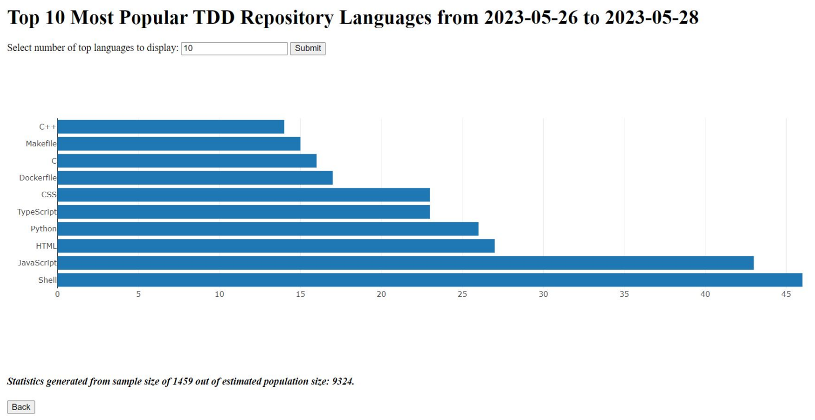

Top 10 Most Frequently Updated Repositories from 2023-05-26 to 2023-05-28

Select number of top repositories to display: 10 Submit

Statistics generated from sample size of 1459 out of estimated population size: 9324.

Back

Figure 4: Flask Application - Most frequently updated repositories


Top 10 Most Popular TDD Repository Languages from 2023-05-26 to 2023-05-28

Select number of top languages to display: 10 Submit

Statistics generated from sample size of 1459 out of estimated population size: 9324.

Back

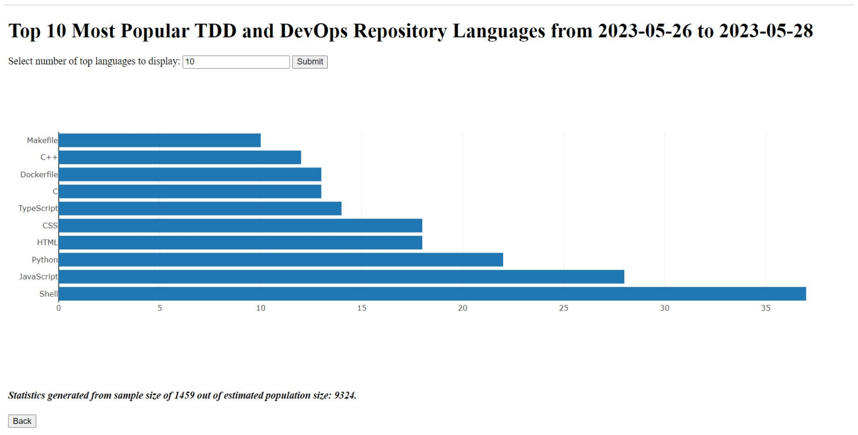
Figure 5: Flask Application - Most popular TDD lanuages

6 Discussion

Analysis of the data reveals a strong correlation between the top N responses to each question, with the exception of the top N languages generally used by repository projects. This correlation is particularly evident when examining repositories with the highest frequency of updates with the most commits pushed within the specified date range. These repositories predominantly involve automated bot commits facilitated through CI/CD using GitHub Actions.

Consequently, concentrating GitHub analytics streaming on the most frequently updated repositories essentially provides insights into DevOps repositories. It is noteworthy that the top N languages for DevOps and TDD repositories share striking similarities. However, it should be clarified that this study focuses on DevOps repositories given specific implementation of TDD, thus accounting for the observed correlation.

Although it is plausible that numerous repositories implementing CI DevOps are geared towards testing it should not be inferred that all CI testing is exclusively intended for TDD. Rather, it could pertain to various testing and software development practices. Uncertainty surrounds whether the top N languages used in TDD repositories are exclusively related to TDD practices, or if they comprise repositories that deviate from TDD or lack testing altogether. This uncertainty stems from the analysis' dependence on the accuracy of keyword matches in repository tokens, commit messages, and workflow file names. It is assumed by using this method that there are many false negatives and false positives in the fetched dataset.


Top 10 Most Popular TDD and DevOps Repository Languages from 2023-05-26 to 2023-05-28

Select number of top languages to display: 10

Statistics generated from sample size of 1459 out of estimated population size: 9324.

Figure 6: Flask Application - Most popular TDD and Devops languages

The assumption is that less frequently updated repositories might not commonly employ testing methods, particularly TDD and DevOps. This supposition reinforces the correlation between frequent updates and testing practices, regardless of the use of DevOps. Nevertheless, it appears that non-DevOps repositories are unlikely to rank among the top N most updated repositories given the relentless frequency of automated updates.

Below table 1 summarizes some of the strengths and weaknesses in our distributed solution to the GitHub analytical streaming problem, and regarding the problem in general.

Table 1: Pros and cons of the distributed GitHub analytical streaming framework

Pros	Cons
Modularity: simple, flexible and maintainable architecture for a distributed infrastructure	**Complex analysis**: TDD implies sequential failing unit test cases before each development pushed to repo; DevOps repositories may not use GitHub Action workflows, or vice versa; There is the assumption repository data accurately conveys such information
Pagination: up to 1000 repositories per calendar date fetching; up to 5000 hits per repository commit and other data	**Rate Limit**: GitHub API rate limit can cause data and time loss, and restarts
Paralellization: as consumer VMs increase, temporal efficiency increases; modifiable for vertical and additional horizontal scalability parallelization	**Time-inefficiency**: API fetching GitHub analytical data is impractical with a single node, especially for dynamic responses to user requests in real-time
NoSQL Database: parallel distributed MongoDB document insertion without conflict	**Errors (400/500 status codes)**: GitHub API connection/server can return errors causing data and time loss, and restarts
Persistence: allows for continuous daily/weekly/monthly/annual updating of GitHub analytical data	**Resource limitations**: horizontal scaling: limited to 4 consumer VMs; vertical scaling: limited to 2 CPUs maximum per VM; API authentication: limited to 4 GitHub tokens and associated usernames
Dynamic User-focused Graphical Analysis: UI displaying of any N top results between 1 and 100 results for each question	**Security** Relevant if the data is sensitive

7 Improvements

There is much room for improvement. Currently, the infrastructure algorithm implementation, with a limitation of four consumers and GitHub tokens, is only parallelizing the overall date range being fetched from the GitHub API. This can be improved by using a new consumer for each repository commit messages and workflow files data fetching, as the source code has been modularly designed and is ready. Furthermore, for each consumer, the date range fetching is iterative per each day in the date range, which allows for vertical scaling with parallelizing between processes. This was not focused on so much for this project because when using a Medium flavor virtual machine there is only a maximum of 2 CPUs available for this.

From a security standpoint, the current implementation may pose risks if sensitive data were to be involved. If the data is breached somewhere throughout the system it could cause potential data leaks or unauthorized access. In such cases using a federated system or distributing data across multiple servers might be more suitable to prevent security breaches.

8 References

[1] F. Beuke, “Madnight/githut: Github language statistics,” 2016. [Online]. Available: https://github.com/madnight/githut

[2] I. Grigorik, “Igrigorik/gharchive.org: Gh archive,” 2012. [Online]. Available: https://github.com/igrigorik/gharchive.org

[3] “Cloud data warehouse to power your data-driven innovation.” [Online]. Available: https://cloud.google.com/bigquery

[4] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 155–165.