Graphical Lines of Code

An easy to use GUI to count lines of code

Ben Greenberg, Spencer Howell, and Daniel Troutman
Team Name: GLOC

I. INTRODUCTION

Our project is a tool called Graphical Lines of Code, or GLOC, based on the popular tool Count Lines of Code (CLOC). CLOC is a command line tool that allows developers to get a detailed breakdown of the contents of their codebase, including the programming languages used and the number of lines of each language, comments, and blank space. [1]

While the CLOC tool has proved to be popular with developers, with over 13,000 stars on GitHub, we believe it would be even more helpful if it provided an easy to parse graphical output. By using charts, graphs, and colors, we can make parsing the results of the tool quicker and easier. This would also allow developers to generate visuals for project updates. In addition, we plan on providing the tool via a web interface, allowing any developer to use the GLOC tool without downloading or installing any software on their machine.

To our team's knowledge, no fully-featured product like this is currently available to developers. While there are a variety of "count lines of code" programs, with features we can investigate adding to our project, no options with a robust graphical output exist. The closest thing that most developers currently use is the "Languages" menu on GitHub repositories, which simply shows the percentage of the codebase that consists of each language. We believe making a tool as accessible as this GitHub feature, with more detailed output, will allow developers to quickly and easily gain a better understanding of their codebase and share that understanding with others.

Our team consists of three graduate students who have experience as software engineer interns at a variety of organizations. We believe this background allows us to see the needs of developers and provide solutions that fill a gap in the current software development ecosystem.

II. CUSTOMER VALUE

A. Customer Need

Our primary customer is a software developer working on a project for their work, for a personal project, or just for fun. Software developers are always searching for ways to improve the health of their codebase - whether that be with automatic testing, code review processes, linting and formatting, or other tools. The goal for these developers is to keep their codebase clean, easy to modify and extend, and free from bugs and other challenges. There are many tools available to help with these goals, many of which are open source, free, and community maintained.

B. Proposed Solution

Our proposed tool will allow developers to measure the composition and health of their codebase with an easy-to-use tool. Using GLOC, they will be able to quickly and easily generate graphs and charts describing their codebase, and use this to measure progress, track trends, and report their findings to their team members and clients. While tools like CLOC exist, we believe GLOC will make these code reports more convenient and accessible to developers.

C. Measures of Success

We will know if customers are benefiting from our tool by a couple of metrics. The first is usage: if we see our tool is being used after it is published, then we will know that others find it useful. The second will be user feedback: we will present our completed tool to other software developers (the students in our class) and have them use GLOC on their own repositories. We will then gather their feedback on the usefulness of the product and make adjustments as needed. This feedback will include categories such as ease of use, user interface design, and detail of the output.

III. TECHNOLOGY

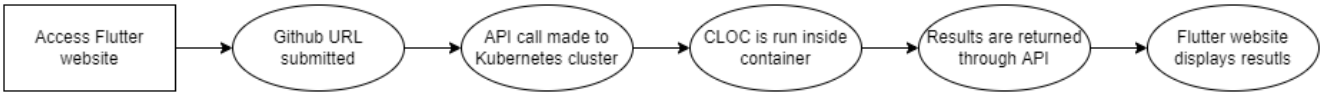
A. System

From a developer perspective, our goal is to have the user provide a link to the repositories they wish to analyze into our Flutter UI. The Flutter UI then communicates with the back-end API that interfaces with the actual CLOC application through a background job runner. After the CLOC job is finished analyzing the git repositories, the job runner will then store the results in a Redis in-memory database temporarily. The API will then wait for the Flutter UI to request the results before sending them back.

Our original goal for a MVP was to provide a simple web-interface that users can use to run the CLOC application without having to open a terminal environment or install the applications on their own machines. We were able to also add some enhancements like language history over commits as well as importing and exporting of results.

B. Tools

We used a cluster of raspberry pi 4s to build our Kubernetes cluster that hosted the CLOC API and web app. We used Flutter to build our front-end and Flask and Redis to



build our back-end API. To make the deployment process easier, we hosted both the API and the web app inside of Docker images that we were able to point the Kubernetes cluster to pull from as needed. We also leveraged GitHub Actions to add a CI pipeline to check our API for any possible security vulnerabilities.

IV. TEAM

A. Skills

While we have not made this kind of project before, we had some reated experience. All of us worked together on a senior design team, and we built a mobile application in Flutter. While our interface is not designed as a mobile app, the cross-platform language means we could use our experience to develop for a browser. We also have experience using APIs from a previous digital archaeology class as well as in our software engineering internships.

We were familiar with some of the tools we used for this project. Figma and Flutter were used in our senior design project, so we worked well with them. For container technology, we all have a little experience with Docker from the digital archaeology class. Other technologies such as Kubernetes were less familiar to us as we have not used them before. There was extensive documentation and tutorials available that we used to create those parts of the project.

B. Roles

Since we each have different specialties in front-end, API, back-end, and automation, we all focused on parts that that worked best for our project. Ben worked on automations and the cluster infrastructure that will run the CLOC application. Daniel assisted Ben with the API and also developed the front-end features. Spencer would then take these features and apply layouts, styles, and other views that best worked with our design.

While we initially planned to have a rotating project owner and others would be the developers, over time we shifted to working within our domains. Features would first be added on the back-end, then propagate to API updates, front-end support, then final design and styling. Through this process we would each work to make sure the feature is supported at every level. This cycle continued until we were satisfied with the features and functionality of our product.

V. PROJECT MANAGEMENT

A. Schedule

We were able to complete our prototype and even add a few extra features within the semester timeframe. We met between classes on Tuesday and Thursday and discussed blocking items and status updates. Communication was also maintained through instant messaging and voice calls outside of our meetings.

- 2/12/22 - 2/18/22: Proposal
- 2/19/22 - 2/25/22: Initial Designs & User Story
- 2/26/22 - 3/04/22: Start Back-End Setup & Architecture
- 3/05/22 - 3/11/22: API Development & API Testing
- 3/12/22 - 3/18/22: API Development
- 3/19/22 - 3/25/22: UI Updates
- 3/26/22 - 4/01/22: Finish Prototype
- 4/02/22 - 4/08/22: UI Updates
- 4/09/22 - 4/15/22: UI Updates
- 4/16/22 - 4/22/22: Configuration & History Updates
- 4/23/22 - 4/29/22: UI Updates
- 4/30/22 - 5/06/22: Wrap-Up & Finish Presentation

B. Constraints

Since this project does inspect code, we only wanted open-source code to start with. We only run code on a private server and don't store any of the actual source material. If someone did want to run measurements on their code, they should own the material or have the right to distribute it for this purpose. If they then decide to use it with our tool, they should know the risks of sending source code to a third-party such as us. There are currently not any ethical or social concerns with building or using our product.

C. Resources

For testing and demonstrations, we used Github to find and download open-source projects.

D. Descoping

We were able to implement our MVP and add some additional features so descoping was not necessary. We planned to have the front-end and back-end modular so we could still demonstrate one without the other. While that is still possible to some extent, our software still relies on some assumptions about the other side like where the front-end is hosted and the architecture of the back-end server.

VI. PROCESS SELECTION

To give us the best chance at success with this project, we decided to use an iterative design process. In the iterative design process, you create a prototype and release betas with more features in each version as you complete the concept. We feel this process aligns with our work style and project because we had an idea for a prototype and a few features. While other processes like scrum may work for different groups, our small team did not want the overhead of backlogs, reviews, and retrospectives. While flexibility is a great benefit for larger teams, we feel that three members naturally gave us this side-effect. Another option we could have used instead of iterative design is the waterfall process. In the waterfall process, the entire product is planned and takes place in a very linear fashion. We are unfamiliar with

some of the technologies we used so it is hard to estimate how much each part would take. If we estimated wrong and the development takes longer than expected, we may not have the time to implement the minimum viable product by the end of class. With iterative design, we got a working prototype, then added features and acquired feedback. If we were not able to complete all the extra options we had in mind, we would still have the core features that satisfied our original goals and be useful as a development tool.

VII. USE CASES

This section describes the primary use case of our application. The primary actor is a software developer working on a non-trivial project. The user goal for this developer is to gain a better understanding of the contents and health of their codebase, and to share that information with others. For the basic flow the user will take to gain this understanding, they will perform the following steps with GLOC:

1) Copy the URL of their repository hosted on GitHub
2) Go to the GLOC website, paste their URL into a text box, and click the "Analyze" button
3) Wait on a loading screen while our servers run the CLOC utility and generate a graphical representation of the results
4) Once done, the user can view these graphs and interpret the results
5) The user can click a download button below the graph to save a json representation of the results to easily share with others in presentations, emails, or instant messages
6) The recipients of this shared result can import it into the tool so they can view it without running the count process again

Our goal was to make these steps as intuitive and easy to perform as possible.

VIII. ARCHITECTURAL VIEWS

A. Development View

One part of our application is the front end that the users will interact with. Here, there are several different structures developed in Flutter to give users a complete and easy experience. One of these structures is the results page where the charts will interact with data acquired through the API. Another interaction is between a configuration page and the web request code. This interface creates customized commands that are sent to be run by our back-end servers. A third structure is the interaction between Flutter and a user's local computer. If someone wishes to save their code statistics, they can download it through web requests. All of these structures and interactions are part of a development view to help solve the problem of effective code exploration.

B. Deployment View

Another view that we implemented is a deployment one. The entirety of the deployment of our application lies within a private Tailscale VPN. The CLOC API is a Flask app inside of a Docker container that uses the RQ library to connect with a Redis deployment. This container is deployed on a Kubernetes pod in a Raspberry Pi K3s cluster with 3 replicas to distribute any incoming requests. The Redis deployment is a separate stateful set also deployed on the Kubernetes cluster. It allows the separate Flask pods to use the Redis deployment as a centralized in-memory database to keep the jobs synced across the pods. The API pods are exposed through the built-in traefik load balancer with an ingress entry that sends any traffic to the pods if they match the url https://gloc-api.homelab.benlg.dev. Similarly the Flutter UI is also deployed in a similar manner to the API with the ingress entry url being https://gloc.homelab.benlg.dev. This domain name is managed through Cloudflare to point any requests to the ip addresses of the nodes in the VPN. In our deployment, the API has CORS restrictions to only allow requests from the Flutter UI deployment, while the Flutter UI can be accessed by any user as long as they have access to the VPN. While we use many tools to support our deployment, each one serves to make the processing faster, more secure, or more robust so we can have a strong deployment scalability.

IX. TESTING

A. Unit Testing

Our unit testing consisted of simple functions that we used for url validation. We wanted to make sure that incorrectly formatted urls received the right message that corrected the problem. Such checks were making sure the url was hosted on github.com and had the correct scheme.

B. Bottom-Up Integration Testing

We were also able to implement integration tests by requesting data from our back-end. By sending a request from our front-end we were able to test the load balancer, API service, and response with both valid and invalid requests. Through this method we can make our server is online and processing jobs correctly.

X. STATIC ANALYSIS

Since we are running user input (the github url) in our API service, we wanted to make sure any potential security issues were found and fixed. We used a static analysis tool called Semgrep to automatically detect problems and suggest fixes. When we ran it on our API code written in python, it found four issues. One issue was a warning that the service might be publicly exposed with our current parameter. Another one it was that it suggested to use environment variables to store configuration variables. The last two were warning that we ran subprocesses without a static string. We addressed all of these warnings by either implementing the suggested fixes, or verifying that part of the code was safe.

XI. PROJECT MEASUREMENT

A. Goal

Our goal was to create a reliable back-end that automatically counts lines of code and returns the results to be displayed in an interactive front-end. Throughout the

project we could measure certain aspects to see how we were progressing over time.

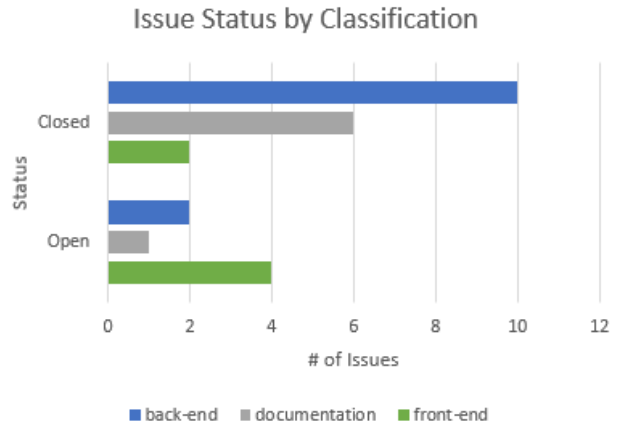
B. Direct Measurement

Since we built a tool that examines lines of code, we used it on our project to get some insights. While this data is easily measured, it may not tell us as much about the status and how close we were to reaching our goal. We can see that most of our code is in Dart which supports out Flutter front end. We assumed that as we make more progress on the front end, Dart lines of code will increase which did end up happening as expected. One thing we did not know, is how many lines of code it will take to get a quality interface; especially since something like this is very subjective.

C. Proxy Measurement

We compiled a list of all the GitHub issues created at that point in the semester and classified them by front-end, back-end, or documentation. By plotting them on a chart by their status and classification, we can see what issues were getting closed and which ones were staying open.

Row Labels	back-end	documentation	front-end	Grand Total
Closed	10	6	2	18
Open	2	1	4	7
Grand Total	**12**	**7**	**6**	**25**



According to our chart we can see that there is a large ratio of front-end issues that were still open compared the the back-end ones. We cant compare the number of issues in each classification since that can vary based on how specific they are and what they are tracking. By the end of the semester, more front-end issues were created and closed so the status ratio looks similar to the back-end.

XII. DEPLOYMENT

Our project has a front-end and back-end that must communicate with each other. They communicate through http requests that deliver json for parameters and data. If both are not deployed properly, only limited functionality will be available.

For the immediate future, we would like to host our flutter front end as well as the api on the cluster. Once we make sure it is secure, we can expose it to the public internet. If someone didn't want to use our tool or wanted to run it themselves, below are some deployment options for the front-end and back-end.

A. Front-end Deployment

1) With repo clone:

- Clone the repo and run the flutter app with "flutter run -d chrome"

2) With docker image:

- Pull and run the docker image "bgreenb3/gloc-ui:latest"

B. Back-end Deployment

1) Run the API:

1) Clone the repo
2) Go to the api directory
3) Supply values for the environment variables
   - REDIS_HOST
   - REDIS_AUTH
   - FLASK_HOST
   - SECRET_KEY
4) Install the dependencies with pipenv install
5) Run the Flask app with the cloc.sh script

2) (Optional) Deploy a Redis server in a k8s cluster:

1) Clone the repo and go to cluster/k8s/redis/
2) Replace the masterauth/requirepass in redis-config.yaml with your own redis password if desired
3) Run and apply all .yaml files in cluster/k8s/redis with "kubectl -f apply <each file here>"

XIII. LESSONS LEARNED

Throughout the semester we faced various challenges and obstacles that we had to overcome. We realized that staying organized and general team structure is hard to maintain unless significant resources can be dedicated. We initially tried the rotating product owner roles but that was a lot of overhead to manage with the team so we pretty much just stuck with our developer specialties.

Another lesson we learned is the complexities of scaling a project to work with multiple users. While we could have just had the CLOC script run locally on the end user's machine, we wanted cross-platform support and flexibility that could not be achieved by only running locally. Since we had the case for a centralized server, we then had to deal with supporting multiple requests at once, we started out using a library called Shell2Http which automatically handled endpoints and running the counting script, when we needed more flexibility, we transitioned to a Flask server with Executor functions. We then ran into an issue of the Flask deployments not having access to all jobs being run resulting in the front-end getting bad results. This caused us to transition from Executor to RQ adding a Redis deployment to act as a centralized in-memory database to hold the results of the jobs run with RQ.

XIV. FUTURE WORK

Even through GLOC is usable and working, we want to continue development to accomplish some goals. The first change we would like to make is splitting the code into multiple repositories. In the class repo, the front-end, back-end, and infrastructure specific code is all together so splitting it up will make it easier to maintain and deploy. While the project is deployable in its current state, we want to make it even simpler with automatic configuration and setup. We also want to keep a demo of the project running so we can show it off to employers or just demonstrate what we built for the class.

REFERENCES

[1] AlDanial, cloc. 2022. Accessed: Feb. 17, 2022. [Online]. Available: https://github.com/AlDanial/cloc