Name: Saicharan Reddy
Email: saicharan.reddy1@gmail.com(primary) / bim2016002@iiita.ac.in (alt)
Institute: Indian Institute of Information Technology, Allahabad [IIIT-A ↗]
Institute Location: India
Current Location : Dallas, Texas, United States
Course: Dual Degree, Information Technology - IV year
Specialization: Information Technology
Github Username : mrsaicharan1
Linkedin Profile: https://www.linkedin.com/in/sai-charan-r-0ba8998a/
Resume Link: G-Drive Link
GSoC 2019 Completion Link: GSoC 2019 Project

Implementation of GitLab Data Collection Workers & Test Coverage Improvement

GSoC Project proposal for CHAOSS

Synopsis

- Augur is a data aggregation platform which generates open-source software health metrics & sustainability statistics by collecting data from a plethora of sources alluding to issue trackers, communication systems, open source repos etc.
- Augur enables users to compare the performance of their repositories(in terms of metrics rather than a competitive fashion) with respect to other open source software repositories.
- The backend of Augur builds metrics based on the data aggregated in the API which in turn is consumed by the frontend to create visualizations as a function of time.
- The primary goal of this project is to congregate data pertaining to GitLab issues, commits, merge requests amongst other entities & store it into the unified data model ecosystem of Augur. Unit & Integration tests would also serve as a vital goal in this project.
- The project where the worker implementation would go: Augur Repository.

- Potential Mentors: Sean Goggins, Gabe Heim & Carter Landis

Benefits to the Community

-
    - The GitLab Worker project would fuel the Augur API to analyze around 546,000 open source projects and nearly 100,000 users hosted on GitLab[source]. This would help us to gain access to GitLab specific repo information and build metrics around several entities such as issues, merge requests and contributors.
    - This would help Augur in widening the number of data sources on which analytics could be performed & aid the project in advancing to its goal of becoming the prime open source data collection metrics & sustainability platform
    - In future projects & extensions, the captured data could also be compared to other projects hosted on GitHub/ BitBucket and a suitable analysis can be performed based on their activity & metrics. Additionally, frontend visualizations could also be built around it to gain more insights about their trends and historical data.

Current Status of the Project

- Presently, the project works on a unified data model ecosystem which extracts data from Git repositories(Github), issue trackers, mailing lists, Linux Badging program, SCC counter among other sources. This project would be mainly focusing on the workers which do the job of populating the tables from APIs and data sources.
- Whenever the Augur server is whipped up, the workers begin the data collection process from the integrated sources.
- Currently, we have workers for the following:
    - A *GitHub* worker which pulls issues, contributors, PRs, repositories & other info to make sense of the data through meaningful metrics
    - A *Facade* worker which focuses on the data from GitHub and provides an analysis who's actually contributing to the the project on a commit-by-commit basis
    - A *Linux Badge* worker which aggregates data from the Linux Foundation's badging program
    - An *Insight* worker which which generates summarizations about raw data corresponding to pull requests, commits & contributors
    - A *Repo Info* Worker that collects repo metadata such as followers, license information, brach info, watchers etc.
- The GitLab worker needs to be implemented right from scratch. GitLab's internal data model is similar to that of GitHub's and ergo, the GitHub worker can be used as a reference throughout the project. For example, Merge Requests are similar to Pull

Requests while the commits and issues stay intact on both of the platforms. An elaborate description about the implementation is given in the **Approach** Section

- Attributing to the Testing Part of the project, there are many missing Unit & Integration tests which are a vital part of the project. This can be accomplished by enforcing a test-driven development workflow. Additionally, API tests & metric tests need to be finished to ensure that the visualizations presented are accurate

Goals

- The primary goal of this project is to implement a worker module similar to the pre-existing workers by understanding the different entities and parameters present in the GitLab API data model.
    - Implement a runtime module which creates a Flask server to configure the Broker & Worker configs, register tasks to the queue and initiate task processing.
    - Implement the aggregation functions for the basic information related to a GitLab project and populating the SQLAlchemy models with the formatted API results -- Issues, Contributors, Merge Requests(akin to GitHub's Pull Requests), Projects (akin to GitHub's Repositories)etc.
    - Integrating data collection functions for all the other tables relevant to GitLab such as *issues_contributors, issue_assignees, pull_request_teams etc.*
    - Addition of new tables to the unified data model(specific to GitLab) -- Access Requests, Environments & Project Deployments.
- Alongside the worker implementation, API tests, unit and integration tests for data collection would also serve as one of the targets for this project.
    - Implement API & Metric tests for newer models
    - Unit & Integration tests for data collection methods
- Test suites for standard methods(if any)

Deliverables

Deliverable 1

- GitLab Runtime Module which configures the Server, brokers & workers, CLI setup and task processing routes
- Worker Module Implementation with Basic GitLab Data Aggregation functions -- Issues, Projects(Repos) & Merge Requests
- Unit tests & Integration Tests for Data Aggregation functions implemented during Phase 1
- Write up a blog post about the implementation of the Phase-I Deliverable

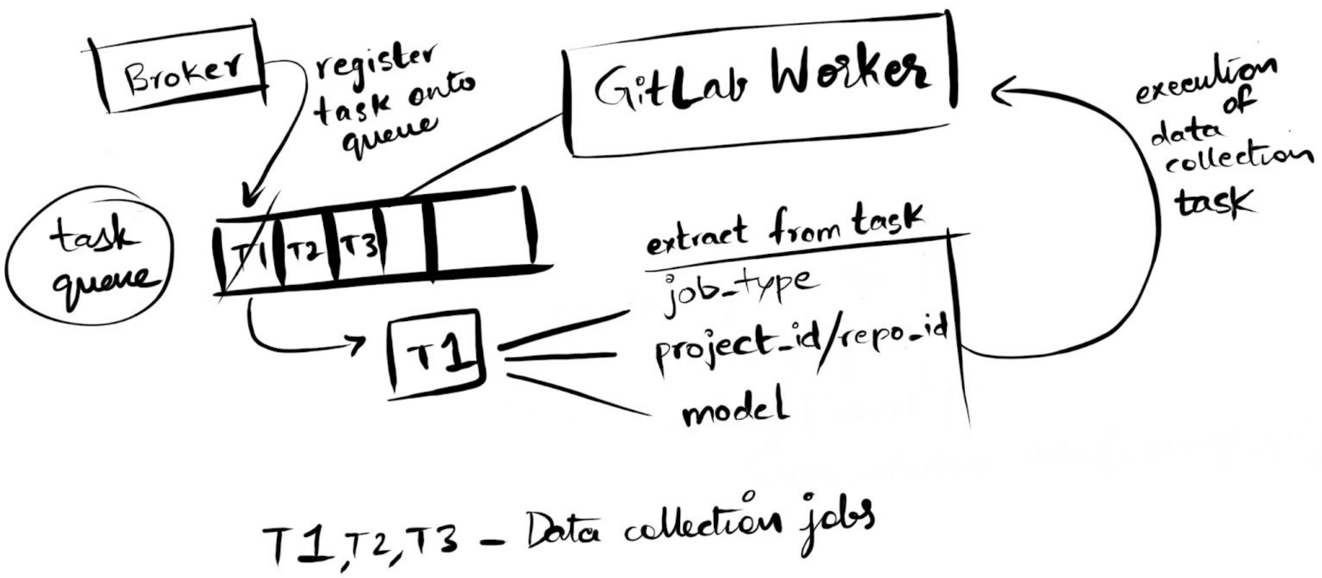
Deliverable 2

- Implementing data collection functions for models related to Merge Requests i.e the models of the format *pull_requests_X* where X=(reviewers, meta etc..) - (separate worker)
- Implementing data collection functions for models related to Issues i.e the models of the format *issues_X* where X=(assignees, events, labels etc..)
- Addition of new models and data collection scripts for the same -- Deployments, Environments, Branches
- Write up a blog post about the implementation of the Phase-II Deliverable

Deliverable 3

- Data Aggregation functions for the remaining models relevant to GitLab,
  - Commits & Commits_X(where X=parents, comment_ref etc.)
  - Repo_X (where X=test_coverage, stats, meta etc.)
- Integration tests for existing GitHub collection functions
- API & Metric Tests for the new models -- Deployments, Environments & Access Requests
- Unit & Integration tests for data collection modules and helper methods implemented during phase II and III
- Update API docs, Augur config files & startup scripts to include GitLab workers.
- Finish up pending tasks(if any) from Phase II and III
- Write up a blog post about the implementation of the Phase-III Deliverable

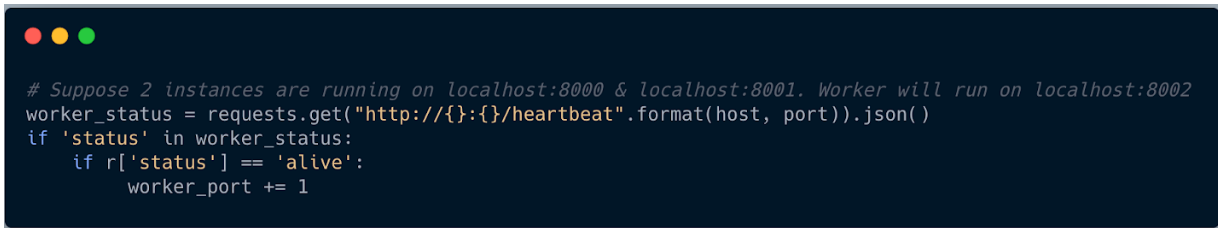
Approach



Blueprint of the GitLab Worker Module

Phase - I

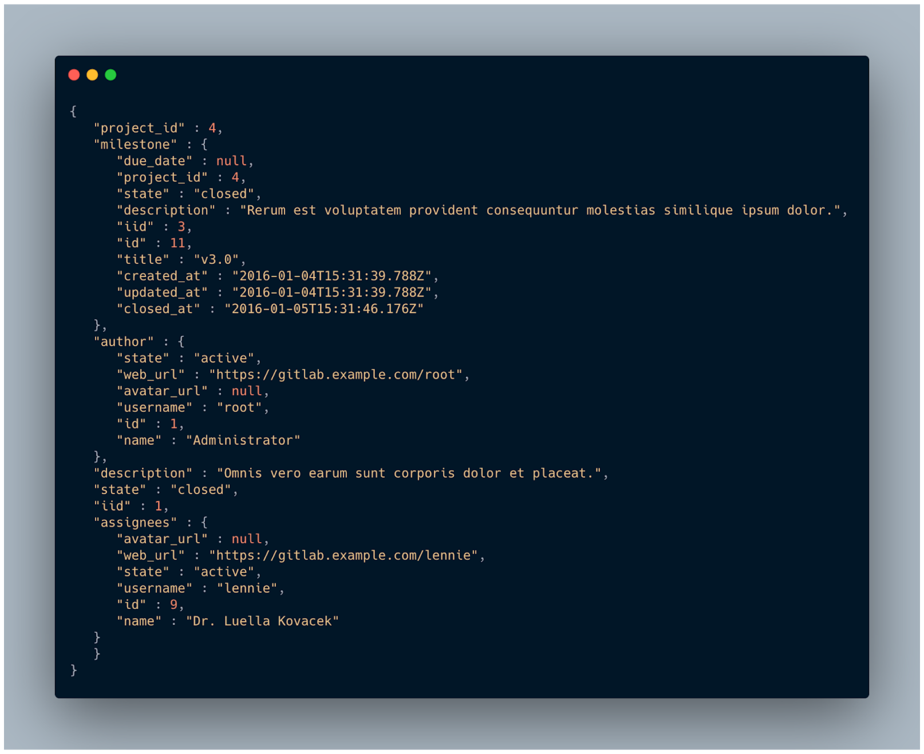
- The GitLab worker requires a runtime module. To implement this module:
  - A function which sets up the flask server and broker would be implemented. This server would contain 3 routes, namely:
    - *Register_task:* The register_task route would act as the endpoint which enqueues the task and gets the current statuses of each task in the queue
    - *Heartbeat:* This route specifies if the server is alive
    - *Config:* This route returns the configuration of the worker
  - A main function which would be coupled with a command line interface creation kit known as *Click.* This would aid us in the manual host/port configuration of the Augur API. This function would then search for an available port to run on by checking the heartbeat of each **host+worker_port** combination. Ex:



```python
# Suppose 2 instances are running on localhost:8000 & localhost:8001. Worker will run on localhost:8002
worker_status = requests.get("http://{}:{}/heartbeat".format(host, port)).json()
if 'status' in worker_status:
    if r['status'] == 'alive':
        worker_port += 1
```

Fig 1.0

- As soon as the process ends, the server is killed.
- The core implementation of the worker resides in the *Worker* Module:
  - The Worker Module would be implemented by declaring a class with *__init__ methods* comprising the initialization of the Task Queue, SQLAlchemy Database Engine, Broker connection & worker configuration.
  - **Data collection functions for issues** would be have to be implemented by hitting the GitLab API at https://gitlab.com/api/v4/projects/:id/issues
    - The response would be in the following manner:



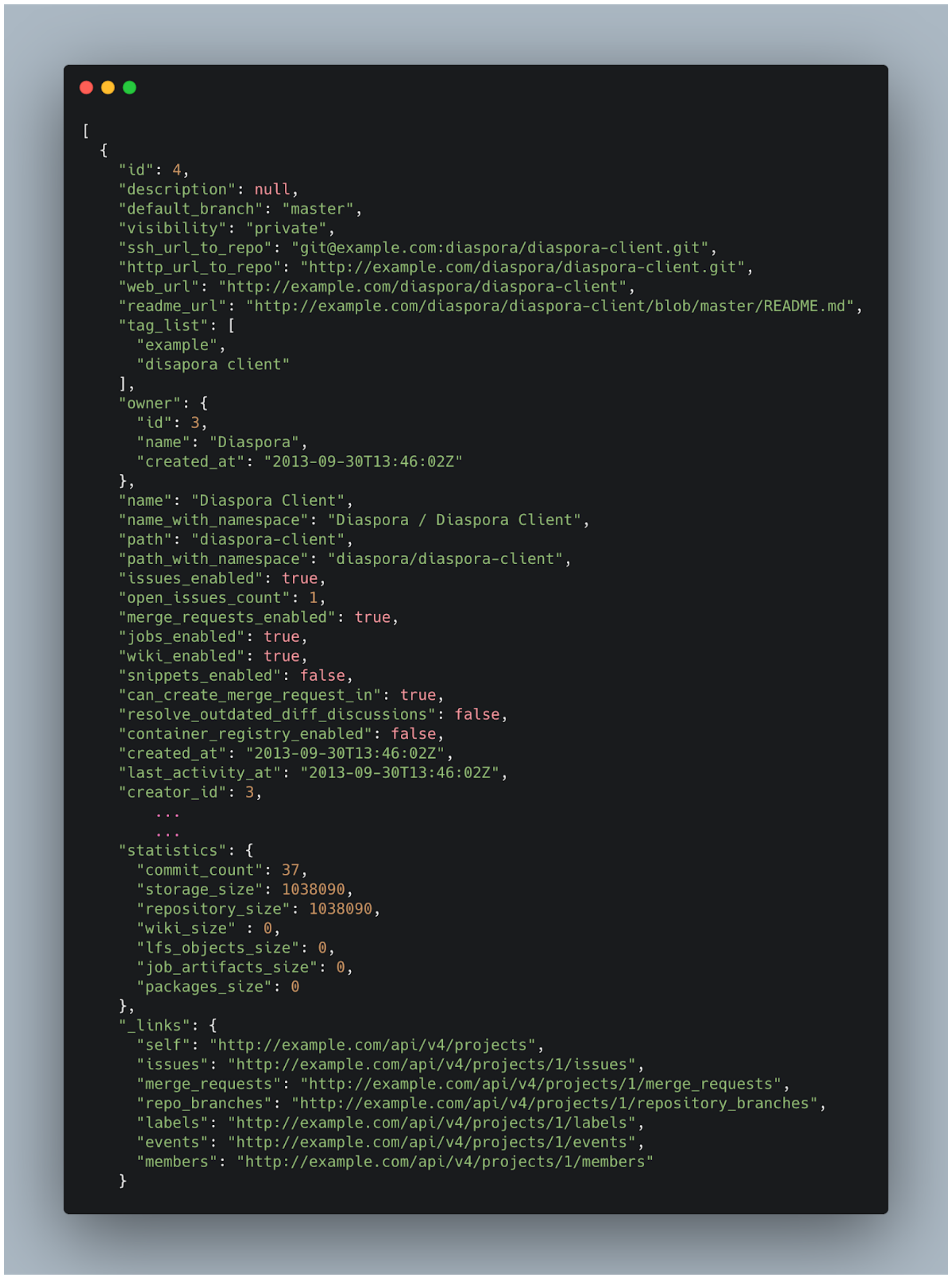
```
{
   "project_id" : 4,
   "milestone" : {
      "due_date" : null,
      "project_id" : 4,
      "state" : "closed",
      "description" : "Rerum est voluptatem provident consequuntur molestias similique ipsum dolor.",
      "iid" : 3,
      "id" : 11,
      "title" : "v3.0",
      "created_at" : "2016-01-04T15:31:39.788Z",
      "updated_at" : "2016-01-04T15:31:39.788Z",
      "closed_at" : "2016-01-05T15:31:46.176Z"
   },
   "author" : {
      "state" : "active",
      "web_url" : "https://gitlab.example.com/root",
      "avatar_url" : null,
      "username" : "root",
      "id" : 1,
      "name" : "Administrator"
   },
   "description" : "Omnis vero earum sunt corporis dolor et placeat.",
   "state" : "closed",
   "iid" : 1,
   "assignees" : {
      "avatar_url" : null,
      "web_url" : "https://gitlab.example.com/lennie",
      "state" : "active",
      "username" : "lennie",
      "id" : 9,
      "name" : "Dr. Luella Kovacek"
   }
   }
}
```

Fig 1.1

    - We extract the necessary parameters such as the GitLab URL, project_owner etc & check if any issues need updation. We also check for duplicates before inserting the issues into our data model.

        ■ Whenever we hit an endpoint for any model, we'll need to keep track of the GitLab API rate Limit corresponding to a specific API key. A function would be written which takes the rate-limit info from the request headers

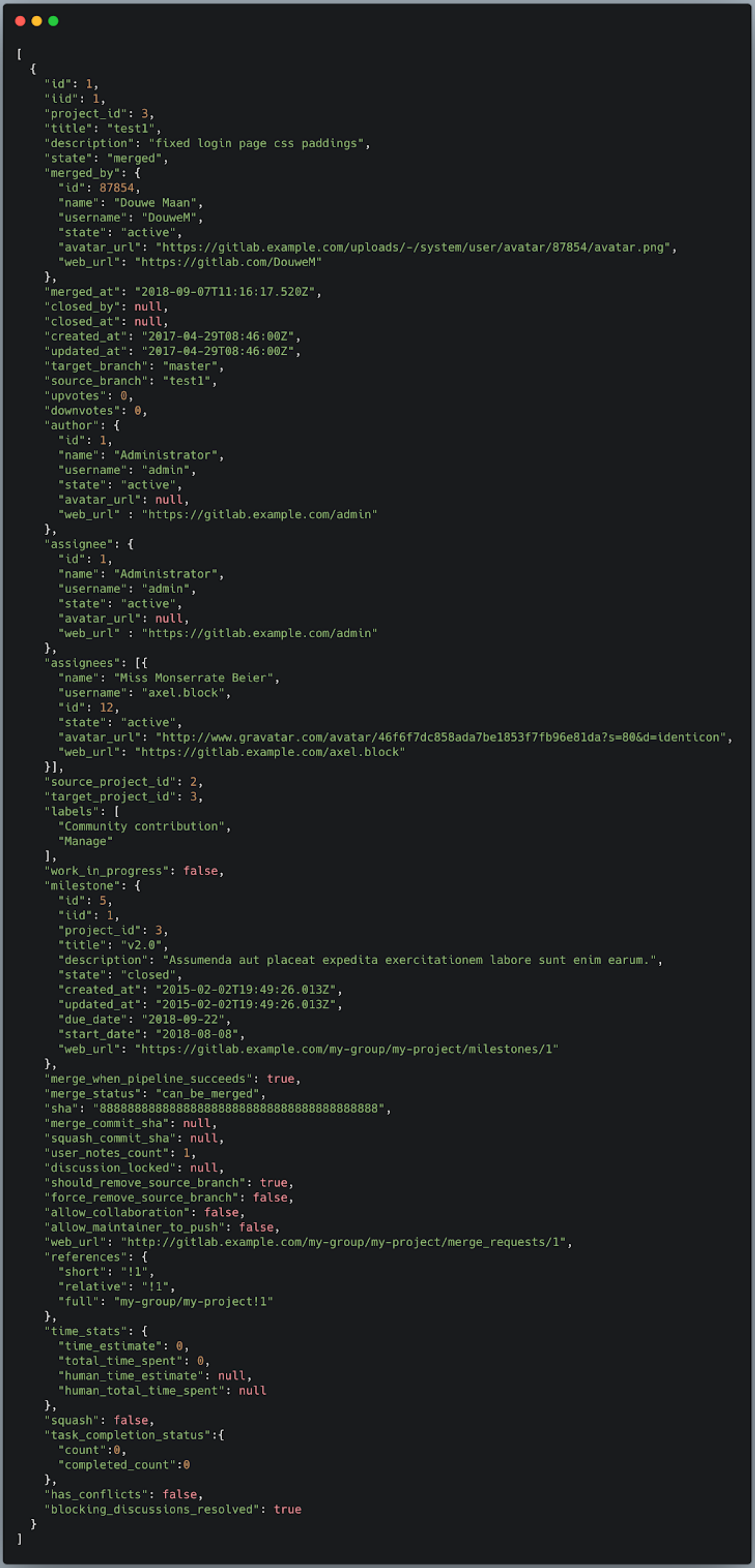
- **Data collection function for Projects/Repositories** would be implemented by sending requests to *https://gitlab.com/api/v4/projects/*
- The API response for this request would be in the following way:



```
[
  {
    "id": 4,
    "description": null,
    "default_branch": "master",
    "visibility": "private",
    "ssh_url_to_repo": "git@example.com:diaspora/diaspora-client.git",
    "http_url_to_repo": "http://example.com/diaspora/diaspora-client.git",
    "web_url": "http://example.com/diaspora/diaspora-client",
    "readme_url": "http://example.com/diaspora/diaspora-client/blob/master/README.md",
    "tag_list": [
      "example",
      "disapora client"
    ],
    "owner": {
      "id": 3,
      "name": "Diaspora",
      "created_at": "2013-09-30T13:46:02Z"
    },
    "name": "Diaspora Client",
    "name_with_namespace": "Diaspora / Diaspora Client",
    "path": "diaspora-client",
    "path_with_namespace": "diaspora/diaspora-client",
    "issues_enabled": true,
    "open_issues_count": 1,
    "merge_requests_enabled": true,
    "jobs_enabled": true,
    "wiki_enabled": true,
    "snippets_enabled": false,
    "can_create_merge_request_in": true,
    "resolve_outdated_diff_discussions": false,
    "container_registry_enabled": false,
    "created_at": "2013-09-30T13:46:02Z",
    "last_activity_at": "2013-09-30T13:46:02Z",
    "creator_id": 3,
        ...
        ...
    "statistics": {
      "commit_count": 37,
      "storage_size": 1038090,
      "repository_size": 1038090,
      "wiki_size" : 0,
      "lfs_objects_size": 0,
      "job_artifacts_size": 0,
      "packages_size": 0
    },
    "_links": {
      "self": "http://example.com/api/v4/projects",
      "issues": "http://example.com/api/v4/projects/1/issues",
      "merge_requests": "http://example.com/api/v4/projects/1/merge_requests",
      "repo_branches": "http://example.com/api/v4/projects/1/repository_branches",
      "labels": "http://example.com/api/v4/projects/1/labels",
      "events": "http://example.com/api/v4/projects/1/events",
      "members": "http://example.com/api/v4/projects/1/members"
    }
```

Fig 1.2

- The SQLAlchemy fields with *pull_requests_merged, pull_requests_opened and pull_requests_count* can be updated by additionally hitting the endpoint a*t response['_links']['merge_requests']*
- Also, by querying this information, th*e pull_request*s table can be updated.
- The functions implementing these subroutines would be separated into a *merge_request_worker* module.
- *Merge_request_labels*, *merge_request_assignees*, *merge_request_comments* etc. would be pulled in the *GLPullRequestWorker* Class
- The response would look like this:

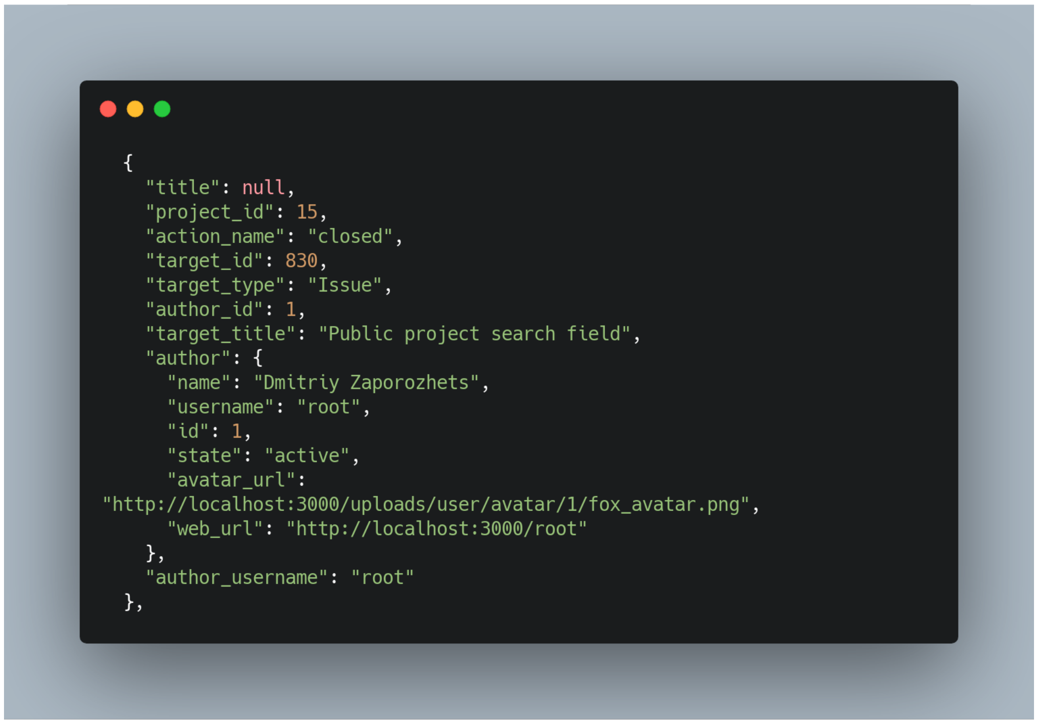


```
[
  {
    "id": 1,
    "iid": 1,
    "project_id": 3,
    "title": "test1",
    "description": "fixed login page css paddings",
    "state": "merged",
    "merged_by": {
      "id": 87854,
      "name": "Douwe Maan",
      "username": "DouweM",
      "state": "active",
      "avatar_url": "https://gitlab.example.com/uploads/-/system/user/avatar/87854/avatar.png",
      "web_url": "https://gitlab.com/DouweM"
    },
    "merged_at": "2018-09-07T11:16:17.520Z",
    "closed_by": null,
    "closed_at": null,
    "created_at": "2017-04-29T08:46:00Z",
    "updated_at": "2017-04-29T08:46:00Z",
    "target_branch": "master",
    "source_branch": "test1",
    "upvotes": 0,
    "downvotes": 0,
    "author": {
      "id": 1,
      "name": "Administrator",
      "username": "admin",
      "state": "active",
      "avatar_url": null,
      "web_url" : "https://gitlab.example.com/admin"
    },
    "assignee": {
      "id": 1,
      "name": "Administrator",
      "username": "admin",
      "state": "active",
      "avatar_url": null,
      "web_url" : "https://gitlab.example.com/admin"
    },
    "assignees": [{
      "name": "Miss Monserrate Beier",
      "username": "axel.block",
      "id": 12,
      "state": "active",
      "avatar_url": "http://www.gravatar.com/avatar/46f6f7dc858ada7be1853f7fb96e81da?s=80&d=identicon",
      "web_url": "https://gitlab.example.com/axel.block"
    }],
    "source_project_id": 2,
    "target_project_id": 3,
    "labels": [
      "Community contribution",
      "Manage"
    ],
    "work_in_progress": false,
    "milestone": {
      "id": 5,
      "iid": 1,
      "project_id": 3,
      "title": "v2.0",
      "description": "Assumenda aut placeat expedita exercitationem labore sunt enim earum.",
      "state": "closed",
      "created_at": "2015-02-02T19:49:26.013Z",
      "updated_at": "2015-02-02T19:49:26.013Z",
      "due_date": "2018-09-22",
      "start_date": "2018-08-08",
      "web_url": "https://gitlab.example.com/my-group/my-project/milestones/1"
    },
    "merge_when_pipeline_succeeds": true,
    "merge_status": "can_be_merged",
    "sha": "8888888888888888888888888888888888888888",
    "merge_commit_sha": null,
    "squash_commit_sha": null,
    "user_notes_count": 1,
    "discussion_locked": null,
    "should_remove_source_branch": true,
    "force_remove_source_branch": false,
    "allow_collaboration": false,
    "allow_maintainer_to_push": false,
    "web_url": "http://gitlab.example.com/my-group/my-project/merge_requests/1",
    "references": {
      "short": "!1",
      "relative": "!1",
      "full": "my-group/my-project!1"
    },
    "time_stats": {
      "time_estimate": 0,
      "total_time_spent": 0,
      "human_time_estimate": null,
      "human_total_time_spent": null
    },
    "squash": false,
    "task_completion_status":{
      "count":0,
      "completed_count":0
    },
    "has_conflicts": false,
    "blocking_discussions_resolved": true
  }
]
```

Fig 1.3

- Integration tests for these data collection functions would be written to ensure that the data is intact and & ready for SQLAlchemy table updation.
- Alluding to the complexity of the relationship between issues and PRs(*MRs here),* the implementation of these collectors could be divided into 2 separate workers.

Phase II

- From Fig 1.1, Population of *issues_assignees* model is plausible as the response also contains info pertaining to *assignees* which could be pumped into an *assignees* python dictionary and then pumped into the SQLAlchemy model.
- Events associated with issues such as close, opened, linked to another issue can be retrieved from the endpoint https://gitlab.com/api/v4/projects/projects/:project_id/events
- Response:
  - 

```
{
  "title": null,
  "project_id": 15,
  "action_name": "closed",
  "target_id": 830,
  "target_type": "Issue",
  "author_id": 1,
  "target_title": "Public project search field",
  "author": {
    "name": "Dmitriy Zaporozhets",
    "username": "root",
    "id": 1,
    "state": "active",
    "avatar_url":
"http://localhost:3000/uploads/user/avatar/1/fox_avatar.png",
    "web_url": "http://localhost:3000/root"
  },
  "author_username": "root"
},
```

Fig 1.4

- The responses need to be filtered out by returning only those entries which have the *target_type* as *"issue"*
- These entries are ready to be loaded into the SQLAlchemy model.

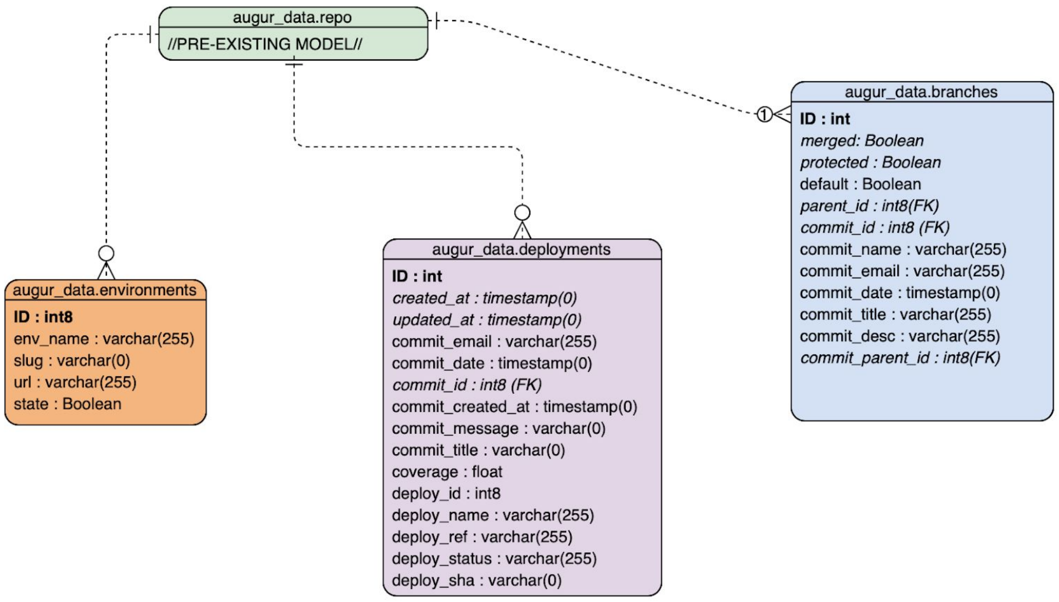
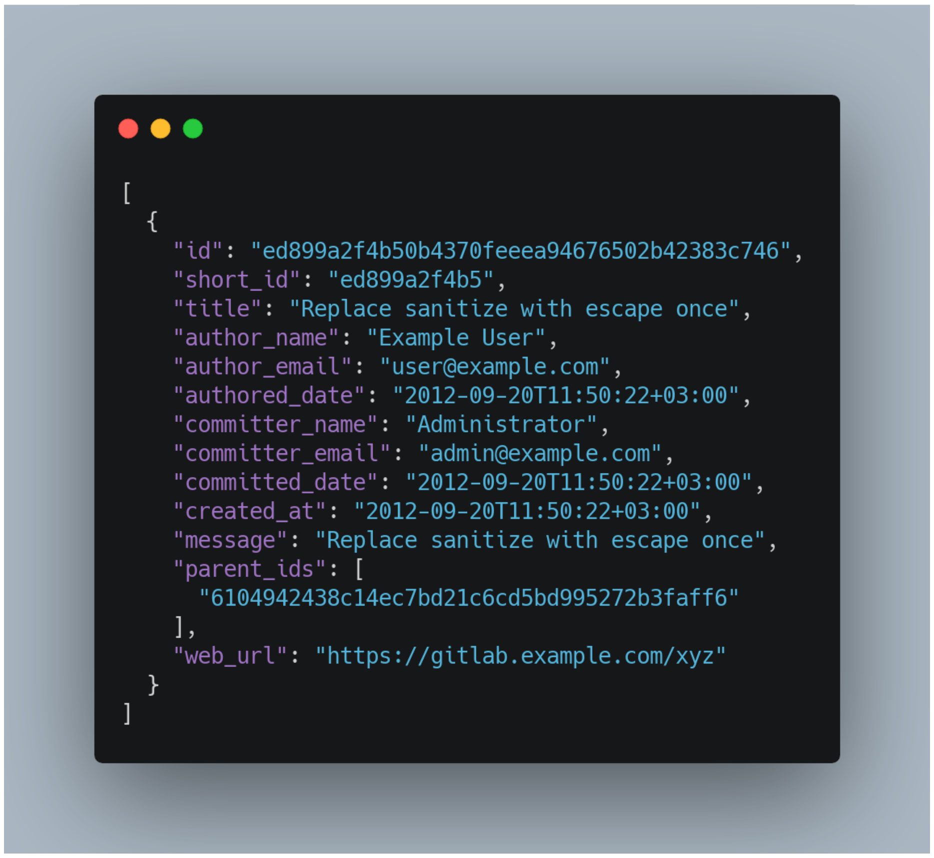
- For Pull Requests/ Merge Requests, we have the response for a specific project at Fig 1.3. To obtain issues that would be closed if the Merge request were to be merged, we'd have to hit the endpoint at "/projects/:id/merge_requests/:merge_request_iid/closes_issues" which would return the issue details. This would be used while updating the merge request table.
- The addition of new models, Deployments, Environments & Branches would have the following ER model:
  - 
augur_data.repo
//PRE-EXISTING MODEL//
augur_data.branches
ID : int
merged: Boolean
protected : Boolean
default : Boolean
parent_id : int8(FK)
commit_id : int8 (FK)
commit_name : varchar(255)
commit_email : varchar(255)
commit_date : timestamp(0)
commit_title : varchar(255)
commit_desc : varchar(255)
commit_parent_id : int8(FK)
augur_data.environments
ID : int8
env_name : varchar(255)
slug : varchar(0)
url : varchar(255)
state : Boolean
augur_data.deployments
ID : int
created_at : timestamp(0)
updated_at : timestamp(0)
commit_email : varchar(255)
commit_date : timestamp(0)
commit_id : int8 (FK)
commit_created_at : timestamp(0)
commit_message : varchar(0)
commit_title : varchar(0)
coverage : float
deploy_id : int8
deploy_name : varchar(255)
deploy_ref : varchar(255)
deploy_status : varchar(255)
deploy_sha : varchar(0)

Fig 1.5
- The existing *augur_data.repo* has a many-one relationship with these new models. To put it elaborately,
  - Each project/repo would have 0 or more environments associated to it.
  - Each project/repo would have 0 or more deployments associated with it.
  - Each project/repo would have at least one branch(master) related to it.
- The data collection models will be populated by accessing the data from the API endpoints:
  - `GET /projects/:id/environments`
  - `GET /projects/:id/deployments`
  - `GET /projects/:id/repository/branches`
- Foreign Keys, which are marked as *FK* are commit_id for deployments; parent_id, commit_parent_id, commit_id for branches.
- SQLAlchemy models would be updated by passing in the dictionary-formatted data from each of these tables.

Phase - III

- In addition, contributors are pulled from the commits API `[https://gitlab.com/api/v4/projects/:id/commits](https://gitlab.com/api/v4/projects/:id/commits)`. The request's response would be in the following manner:

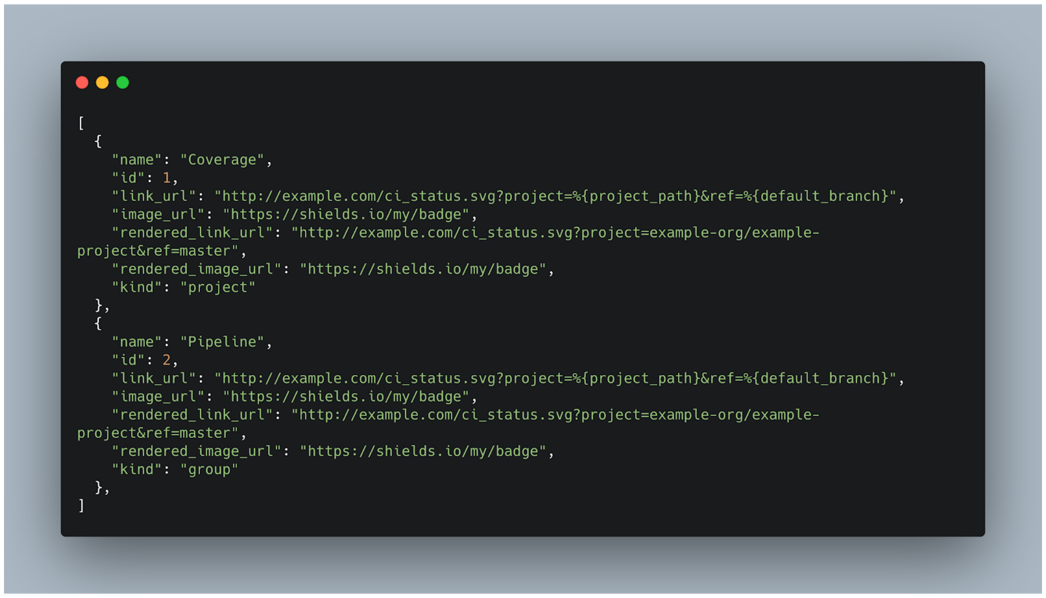


```
[
  {
    "id": "ed899a2f4b50b4370feeea94676502b42383c746",
    "short_id": "ed899a2f4b5",
    "title": "Replace sanitize with escape once",
    "author_name": "Example User",
    "author_email": "user@example.com",
    "authored_date": "2012-09-20T11:50:22+03:00",
    "committer_name": "Administrator",
    "committer_email": "admin@example.com",
    "committed_date": "2012-09-20T11:50:22+03:00",
    "created_at": "2012-09-20T11:50:22+03:00",
    "message": "Replace sanitize with escape once",
    "parent_ids": [
      "6104942438c14ec7bd21c6cd5bd995272b3faff6"
    ],
    "web_url": "https://gitlab.example.com/xyz"
  }
]
```

Fig 1.6

- Grouping the *author_emails* into a list would provide us with the repository's contributors which in turn would be updated in the *contributors* table. Duplicate checks will also be performed in the function itself.
- We also extrapolate the number of commits which require updation (scenarios such as commit history modification due to force pushes)
- Parent commits can be accessed easily with the help of the parent_ids list from the response.
- **Collection subroutine for Repo_X(X=badging, labor, libraries) Information** is handled by the API endpoint : *gitlab.com/api/v4/projects*
  - The sub-routine will grab the information from the badges API whose response is as follows:
    - Similar to the other data collection functions, the GitLab badging function will aggregate the badging name, kind, link & image URLs into the data model.
    - This function will also check for any existing badges which need updates by checking the number of badges which are already

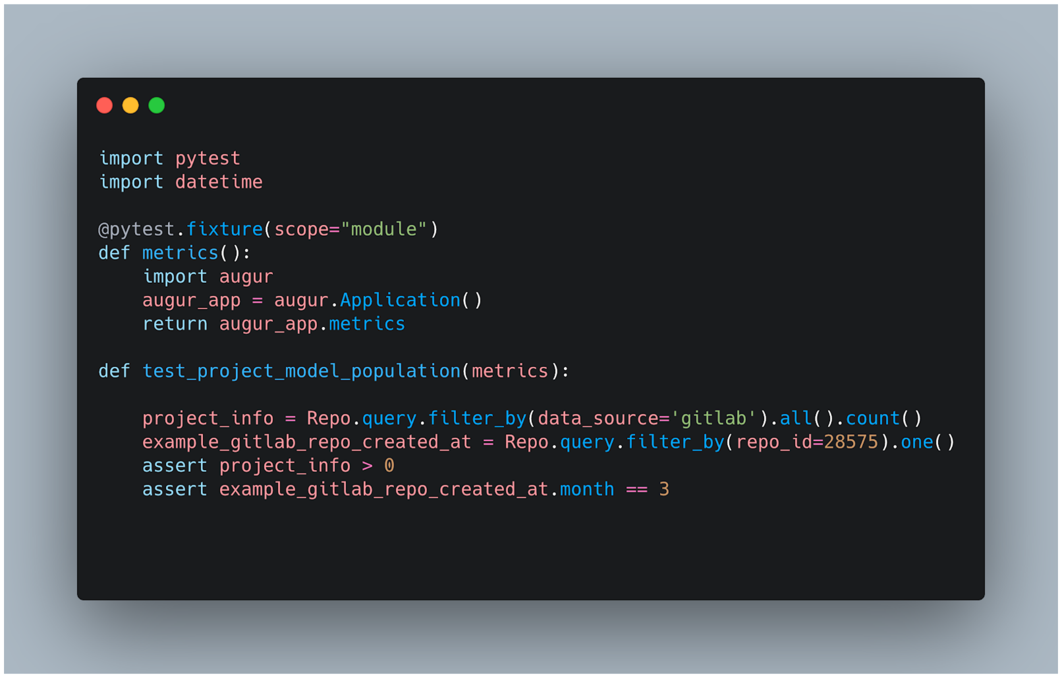
present. Post this, a suitable number of insertions & updations will be performed on the model.

- 

```
[
  {
    "name": "Coverage",
    "id": 1,
    "link_url": "http://example.com/ci_status.svg?project=%{project_path}&ref=%{default_branch}",
    "image_url": "https://shields.io/my/badge",
    "rendered_link_url": "http://example.com/ci_status.svg?project=example-org/example-
project&ref=master",
    "rendered_image_url": "https://shields.io/my/badge",
    "kind": "project"
  },
  {
    "name": "Pipeline",
    "id": 2,
    "link_url": "http://example.com/ci_status.svg?project=%{project_path}&ref=%{default_branch}",
    "image_url": "https://shields.io/my/badge",
    "rendered_link_url": "http://example.com/ci_status.svg?project=example-org/example-
project&ref=master",
    "rendered_image_url": "https://shields.io/my/badge",
    "kind": "group"
  },
]
```

Fig 1.7

- For the pre-existing GitHub Data Collection Worker, the data being collected by the workers needs to be validated by integration tests. Issues, commits, pull_requests etc. need tests to ensure consistency of data.
- Using *pytest*, we can set up the API in a separate function in the test module. In the same file, separate test cases would be implemented as functions which would be asserting the expected & actual values.
- Integration & Unit tests for aggregation functions implemented during Phase II & III would be written to check if the data collection process was successful. A mock test is given below:



```python
import pytest
import datetime

@pytest.fixture(scope="module")
def metrics():
    import augur
    augur_app = augur.Application()
    return augur_app.metrics

def test_project_model_population(metrics):

    project_info = Repo.query.filter_by(data_source='gitlab').all().count()
    example_gitlab_repo_created_at = Repo.query.filter_by(repo_id=28575).one()
    assert project_info > 0
    assert example_gitlab_repo_created_at.month == 3
```

Fig 1.8

- API docs for the GitLab Worker will be updated. Additionally, the GitLab Worker will be included in the startup scripts & the new models will be added into the schema.
- The generated config file would also include the GitLab Worker config in the appropriate blocks.

Timeline

Period	Task
After proposal submission [April 1 - May 18]	- Increase the test coverage by adding maximum number of unit & integration tests - Implement 1-2 metrics to understand how the data collection process works & how it's consumed by the frontend - Fix bugs related to the workers
Week 1 and 2 [May 18 - May 31]	- Implementation of GitLab Runtime Module which configures the Server, brokers & workers, CLI setup and task processing routes - Worker Module Implementation with Basic GitLab Data Aggregation function for **Issues**
Week 3 and 4 [June 1 - June 15]	- Data collection subroutines for **Projects(Repos) & Merge Requests(PRs)** - Unit tests & Integration Tests for Data Aggregation functions implemented during Phase 1 - Finish pending work for evaluation(if any)
Week 5 and 6 [June 16 - June 27]	- Integration of data collection functions for all models related to Merge Requests i.e the models of the format *pull_requests_X* where X=(reviewers, meta etc..) - Addition of new models and data collection scripts for the same -- Deployments, Environments, Branches
Week 7 and 8 [June 30 - July 17]	- Implementing data collection functions for models related to Issues i.e the models of the format *issues_X* where X=(assignees, events, labels etc..) - Write unit & integration tests for pull_request_X aggregation functions

Week 9 and 10 [July 13 - July 26]	- Data Aggregation functions for the remaining models relevant to GitLab, - Commits & Commits_X(where X=parents, comment_ref etc.) - Repo_X (where X=test_coverage, stats, meta etc.) - Addition of new models and data collection scripts for the same -- Deployments, Environments, Branches
Week 11 and 12 [July 27 - August 10]	- API & Metric Tests for the new models -- Deployments, Environments & Access Requests - Remaining Integration tests for data collection modules for *issues_X* where X=(assignees, events, labels etc..) - Update API docs, Augur config files & startup scripts to include GitLab workers. - Finalize implementation with mentors & fix issues (if any)

About Me

Bio

I'd like to describe myself as a learning entrepreneur and a Software Engineer who always loves to learn and build tools with new technologies. I'm an AWS Certified Professional and I was also a successful *Google Summer of Code* Student with the Organization FOSSASIA in 2019.

Adding to that, I've also built my own startup named Freeflo which builds AI and machine learning products for other non-tech startups. I've also won numerous hackathons across India and was a member of the National Champion Team of Smart India Hackathon 2019, the world's largest hackathon.

Skills:
Programming Languages: Python, C/C++, Java, Javascript
Backend Technologies - Flask, Django, Node.js
DB technologies - PostgreSQL, MongoDB, MySQL
Frontend Technologies - Ember.js, HTML, CSS, JS
Cloud Services - Docker, AWS

Contributions & Microtasks

- I've been an active contributor to the Augur Project since February 2020. I've implemented a new metric, wrote unit & Integration tests pertaining to other workers
- I've also completed the GitLab Data Collection Microtask
- I'm also one of the collaborators for the Augur Repository.
- Contributions to the Augur Projects - https://github.com/chaoss/augur/pulls?q=is%3Apr+is%3Aclosed+author%3Amrsaicharan1
- List of Micro-tasks completed: https://github.com/mrsaicharan1/chaoss-microtasks/blob/master/README.md
- Other open source projects I've contributed to :
    - https://github.com/fossasia/open-event-server/commits?author=mrsaicharan1
    - https://github.com/fossasia/open-event-frontend/commits?author=mrsaicharan1

Additional Info

- Post GSoC, I'll continue to contribute to the project and review co-developers' code. I'd also be willing to help new developers get up to speed with the Augur project & aid them in fixing their issues.
- The **core reason I chose to work with Augur** is the well-built community and extremely helpful maintainers. Whenever I went through the codebase and wondered about a certain subroutine and its application in the data aggregation process, the maintainers never failed to reach out & clear the air!
- In addition to the above, I've always been interested in data engineering/data science and wanted to learn about how data is collected reliably at scale for further processing such as anomaly detection & other data science techniques.
- For maximum success, I laid out a concrete plan on how to stay on track with the project. Before the actual coding period, I would be familiarizing myself with the whole codebase, particularly with the implementation of the workers, the schema definitions & how the metric & API tests were written.
- Adding to the above, each week, I'd be utilizing the working days to implement the ideas laid down in this proposal & also read the docs for any new library/technology we'd be using along the way. Attending the weekly stand-up meetings about the goals accomplished and the targets for the following week would help me in channeling the development process.
- I'd be working for **40 hrs/week** during the period of GSoC (if selected)

- During the summer, I would be having **no commitments** other than GSoC (if selected)