Extensible GA4GH Client Library/SDK and Command Line Interface implemented in Rust

Personal Information

- **Name**: Aarav Mehta
- **Email**: aarav.mehta.mat22@itbhu.ac.in (college mail id) / aarav05mehta@gmail.com (personal mail id)
- **GitHub**: https://github.com/aaravm
- **Phone no. :** +91 9653321136
- **Major:** Mathematics and computing
- **University:** Indian Institute of Technology, Varanasi
- **Time zone**: UTC+5:30

Motivation

I am a 2nd year undergraduate, studying at IIT (BHU) Varanasi. I have been doing some kind of software development since high school (mainly web development). But after coming to my college, I was introduced to a whole new world and I have learnt about so many new and interesting languages, frameworks and fields.

I was introduced to the world of open source during December 2023, and I fell in love with the idea of open source- to prevent some private company owning an important library, instead it being maintained in a community, where anyone can come and check the work of the community and contribute to it. I got to know about GA4GH, as one of my seniors worked here 2 years back. He had a glowing review of GA4GH, and when I got into the community, I felt that it was one of the best communities, being very interactive and welcoming to new members.

As a first step to this, I was assigned a task of support for private docker images in task executors, assigned to me here: https://github.com/elixir-cloud-aai/tesk-api/issues/55
I first discussed the implementation plan with Pavel, Grigorii and Ruslan. Then, they assigned me this issue, and I have been implementing the plan discussed. Till now, it has been a great experience. I am now acquainted with the coding practices that are expected from me.

Through GSoC, I want to implement a command line interface, which will make the library and CLI accessible to a wide developer audience. This way, the popularity of GA4GH will improve, and more organizations will use the services provided. This means more information available

to everyone, which may improve lives. Further development of CLI and Library is also possible, as I intend to make the repository contributor friendly

During this period, I currently have no other commitments. I'll have summer vacations which I will be spending at my hometown. I intend to remain connected with the community and contribute to it even after GSoC and expand this project to a next level.

Technical Skills

I am currently a sophomore at IIT (BHU) Varanasi in the field of maths and computing, pursuing my B.tech+ M. tech degree. I have a CGPA of 9.68 out of 10, and am 3rd highest in my class of 60 students. I have completed the courses of Discrete Mathematics, Computer System Organisation, Computer Programming, Data structures and algorithms, Probability and Statistics, Linear Algebra with an A grade.

I am a very curious person, and I have explored a lot of fields, and have had prior experience in competitive programming, open source, web development, web3 development, system level programming, machine learning.

I have done competitive programming, and currently am a specialist on codeforces. Apart from that, I am always exploring and learning different things and participating in hackathons, competitions etc. I have gotten a silver medal in the prestigious month long hackathon, Inter IIT Tech Meet 12.0 (held by IIT Madras) in the field of web3, where I was responsible for the backend, written in move, a rust based language. Other than that, I have gotten the first prize in Silicon Maze (held by NIT Surathkal) in the ML track, and gotten 1st Prize in Cosmomath, NSSC'23 (held by IIT Kharagpur).

I am also an open source contributor, with 2 merged pull requests in the optuna library, written in python: https://github.com/optuna/optuna/pull/5322 and https://github.com/optuna/optuna/pull/5306

I came All India Rank 1905 in JEE Advanced, India's most prestigious Engineering exam, where every year, over 1 million students apply. I have also qualified for the Indian National Mathematics Olympiad (Final qualifier for the Indian Team for International Mathematics Olympiad), by coming within top 25 in my state in its qualifier exam.

Some of my past projects have been: (For more details, click the link)

- Futures on chain
- Social Chain
- HealthCare 3.0
- Zip(Web Assembly)
- Image Captioning model
- Neural ODE's

Contents

Project Details

Brief Background

The Global Alliance for Genomics and Health

The Global Alliance for Genomics and Health aims to provide frameworks and standards for working with genomic and other health-related data. Thousands of healthcare practitioners and researchers access, process, analyse the data, stored in various databases, and increasingly base prescriptions and treatment plans on the obtained results. As the actual data, as well as the tools and workflows used to generate them, are valuable commodities, it is important that they can be found, accessed and reused in an interoperable manner. The GA4GH standards promote and ensure fair data principles throughout the field of biomedical data exchange and analysis.

The GA4GH Cloud Work Stream

The Cloud Work Stream is focused on creating specific standards for defining, sharing, and executing portable workflows and accessing data across clouds. Simply said, The Cloud Work Stream helps the genomics and health communities take full advantage of modern cloud environments by bringing algorithms to data.
There are 4 API standards:

- Share tools/workflows (TRS),
- Execute individual jobs on clouds using a standard API (TES),
- Run full CWL/WDL workflows on execution platforms (WES),
- Read/write data objects across clouds in an agnostic way (DRS).

Related Work:

The main aim of this project is to develop a highly extensible client library and command line interface (CLI) for interacting with GA4GH environments.
To date there are, to my knowledge, many command line interfaces made to interact with the API's, none of them in rust. There were not many GA4GH-compliant systems in the days when these cli's were made. However the situation is now changing, with several companies such as DNAstack, SevenBridges, LifeBit, and others recognizing the need for interoperability. Here, I will analyze the existing implementations, and check if it is possible/efficient to try to re-use them:

[GA4GH CLI]

The Data Working Group of the Global Alliance for Genomics and Health has defined an API to facilitate interoperable exchange of genomic data. This repository uses runners to handle the CLI. It uses 3 classes:

- AbstractClient is a base class that defines common behavior and methods that all client classes should implement
- HttpClient is a specific implementation of a client that extends AbstractClient and provides concrete implementations for the abstract methods defined in AbstractClient. It is responsible for interacting with the API
- LocalClient class is designed to communicate with a backend service running locally, which provides methods for accessing and manipulating genomic data. It also extends AbstractClient

TES CLI

Task Execution Service (TES) API provides a standard mechanism for orchestrating complex analyses across different compute environments. Mock-TES is a Connexion-based mockup service implementing parts of the GA4GH Task Execution Service (TES) API schema.

- This Python code defines a client for interacting with a mock Task Execution Service (mock-TES) using Swagger/OpenAPI specifications
- Bravado Library is used for interacting with the mock-TES API
- Client Class: The `Client` class is initialized with a base URL (`url`) and an optional JWT token (`jwt`) for authentication. It creates a `bravado` Swagger client (`self.models`) using the Swagger JSON file at `{url}/swagger.json`. It also sets up an API key if a JWT token is provided.

Multi-WES CLI client:

WES describes a protocol for running the same genomic data analysis in multiple cloud environments. This CLI works with the following workflow:

- setup.py defines the entry_points, which calls the main.py in WesCli, where the commands in the args are parsed through, calling different files on the basis of the command called. The different commands are:
    - run
    - get
    - upload
    - download
    - Status

- All the functions in the respective files call the WES API (respective functions defined in WesCli.py), while handling errors and default values.
- The example directory shows the formatting of the various spec files, and how they should look so that the commands run without error, which are also run by the tests
- The tests folder individually tests all different functions used, their integrations and different cases for whether tests pass or fail.

TRS-CLI

The Tool Registry Service (TRS) API provides a standard mechanism to list, search, and register tools and workflows across multiple registries.

Importantly, TRS supports tools and workflows described by the Common Workflow Language (CWL), the Workflow Description Language (WDL), and Nextflow—three of the most widely used workflow languages for running analyses on genomic data

- Has a Sphinx documentation builder
- Class TRSClient: Client to communicate with a GA4GH TRS instance defined in client.py. Has many different methods, designed to interact with the TRS API like
    - Post_service_info
    - Post_tool_class
    - _validate_content_type
    - get_tool_classes
    - Get_versions

  And so on.

- models.py: numerous classes helping manage TRSClient
- Test_client.py: various tests to verify all the functions

DRS-CLI

In order to analyse genomic data in the cloud, a researcher must use an access tool to retrieve the file. Today, however, data repositories are crowded with files. As a result, the process for retrieving a data set is complex and inefficient. To address this challenge, the GA4GH Cloud Work Stream has developed the Data Repository Service (DRS) API, which provides a standard way to retrieve a dataset regardless of the repository's underlying architecture.

This file has the same structure as TRSClient, with a class of DRSClient and different functions corresponding to the DRS API.

Ga4gh-drs-client

This is a command-line application for requesting omics data and metadata from web services that are compliant with the Data Repository Service (DRS) API Specification. This can be run using pypi or using docker.

- The main code for accessing the api and building the cli is in ga4gh-drs-client/ga4gh/drs
- A very comprehensive series of tests in ga4gh-drs-client/unittests, with all methods being tested

Drs-compliance-suite

- A very comprehensive repository to use the DRS. This can be run natively, using pypi and using docker
- Supports only the version drs 1.2.0
- Option for multiple command line arguments like --server_base_url, --drs_version, --config_file and so on.

Project Goal & Milestones

The aim of this project is to develop a highly extensible client library and command line interface (CLI) for interacting with GA4GH environments, using Rust for its development. It will support key GA4GH standards (DRS, TRS, TES, WES, Passports), focusing on modularity, performance, and cross-platform usability.

The following milestones can be defined:

- **Milestone 1 (MS1):** Create services.rs, a base class in rust to call any and all API's, and all rust classes will be implemented on top of this
- **Milestone 2 (MS2):** Create 5 different classes on top of services.rs to call each respective API from the library
- **Milestone 3 (MS3):** Add authentication to the python cli to obtain the tokens that can be passed to the API's headers
- **Milestone 4 (MS4):** Develop the python cli further to incorporate calling DRS, TRS and WES from the python CLI

- **Milestone 5 (MS5):** Create bindings using `pyo3` and `maturin` to allow python cli to call the functions from the rust library
- **Milestone 6 (MS6):** Allow Auto Code Generation from OpenAPI specs to support multiple GA4GH API versions, to reflect updates to the API in the CLI, using the rust generator.
- **Milestone 7 (MS7):** Create bindings with C and Go using the `cbindgen` tool crate like `cdylib`, and the `cgo` library (for go) to allow users to call functions from C and Go, and enhance the cross-platform usability
- **Milestone 8 (MS8):** Create bindings with Javascript using WebAssembly, and creating basic index.html to allow users to call functions from JavaScript and enhance the cross-platform usability
- **Milestone 9 (MS9):** Produce comprehensive documentation and engage with the GA4GH community
- **Stretch Goals:** Explore the integration of Confidential Computing features, supported by an open specification and development server from GENXT.

Guidelines: About - Rust API Guidelines

Proposed Model

We will have 4 major divisions in our repository:

a. **A highly extensible client library repository written in rust**
   i. Implementing the core functionality of interacting with GA4GH APIs using Rust.
   ii. Exposing functions/methods that perform actions like querying DRS objects, executing TES workflows, etc.
   iii. Defining error handling mechanisms that can be responded to in Python
   iv. Writing tests to ensure that the repository works as expected
b. **An interactive python cli, through which user can easily interact with the library. Initial implementation: https://github.com/genxnetwork/ga4gh-cli (written in python)**
   i. Parsing command-line arguments for the CLI, and having the ability to read and accept files
   ii. Using maturin crate and pyo3 library in rust to integrate the rust library with the CLI.
   iii. Calling Functions: Writing functions in Python that call the corresponding functions in the Rust library, passing arguments and handling return values appropriately.

iv. Error Handling: Translate errors from the Rust library into Python exceptions, providing meaningful error messages to the user.
v. User Interface: Provide a user-friendly interface for users to interact with the GA4GH APIs, writing documentation properly and in detail to make sure users can easily interact with the cli

c. **Auto Code Generation from OpenAPI specs to support multiple GA4GH API versions**
   i. Saving all the specs of API's as .yaml in the file repository, taking it from online github repositories, where the API is generate
   ii. Generating all the repository using the openapi generator, using the command line code
   iii. Integrating all the code with existing directories
   iv. Documenting this process well, and provide a guide for how to do this, to make updating the command line interface as easy as possible
d. **Cross Language Bindings in Python, C, Go, and Javascript**
   i. Creating new repositories for each language, and call the functions tes, trs, drs and wes in the functions and edit them in specific ways depending on the language
   ii. Running specific commands for each language to generate the bindings in each language
   iii. Adding tests to ensure the correct output is being returned, and to root out any errors
   iv. Adding documentation to make the bindings easily generatable, and easy to reproduce.

File Structure of the repository:

```
├── Cargo.lock
├── Cargo.toml
├── cbindgen.toml
├── tes.yaml
├── trs.yaml
├── wes.yaml
├──drs.yaml
├── src/
│   ├── lib.rs
│   ├── service.rs
│   └── src/
│       ├── tes.rs
│       ├── aai.rs
│       ├── trs.rs
│       ├── drs.rs
│       └── trs.rs
```

```
├── cli/
│   ├── cli.py
│   ├── utils.py
│   ├── auth.py
├── tes/   //similarly for trs, wes, drs
│   ├── src/
│   ├── docs/
│   ├── cargo.toml
├── c/   //similarly for javascript and go
│   ├── src/
│   ├── tests/
│   ├── cargo.toml
├── examples/
└── tests/
└── Readme.md
```

Implementation Details

Setup

```
$ cargo new my-project
    Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

A highly extensible client library repository written in rust

Writing service.rs:

Service class provides a generic way to make HTTP requests to a specified base URL with optional authentication and handles logging and error checking.

Libraries used:

a. Serde: framework for serializing and deserializing Rust data structures efficiently and generically.
b. Reqwest: Use the reqwest library (https://crates.io/crates/reqwest) for making HTTP requests in Rust. It provides a clean and ergonomic API similar to requests in Python

Why choose Reqwest over Hyper?

Reqwest supports all the functionalities we require, and is very easy to use and intuitive, whereas Hyper is complicated, and its code easily breaks. It is not used unless absolutely required.

Basic code structure:

```
use reqwest::Client;
use reqwest::header::{Header, AUTHORIZATION};
use serde::{Deserialize, Serialize};
use std::error::Error;
use log::{debug, error}; // Assuming you're using the `log` crate for logging

// Define a struct for authentication credentials (optional)
#[derive(Debug, Serialize)]
struct Credentials {
    username: String,
    password: String,
}

// Define a struct for the service object
pub struct Service {
    base_url: String,
    auth: Option<Credentials>,
    token: Option<String>,
}
pub impl Service {
    // Constructor
    fn new(base_url: String, auth: Option<Credentials>, token: Option<String>) ->
Result<Service, Box<dyn Error>> {
        if base_url.is_empty() {
```

```
            return Err();
        }
        Ok(Service { base_url, auth, token })
    }
```

```
    // Function to make a request
    pub async fn request(&self, method: &str, endpoint: &str, data: Option<serde_json::Value>,
params: Option<serde_json::Value>) -> Result<serde_json::Value, Box<dyn Error>> {
// adding urls and headers
        let url = format!("{}/{}", self.base_url, endpoint);
        let mut headers = HeaderMap::new();
        headers.insert(CONTENT_TYPE, HeaderValue::from_static("application/json"));
// checking for username and auth, and tokens
// Now, using reqwest library to call the API:
        let client = Client::new();
        let response = client
            .request(method, &url)
            .headers(headers)
            .json(data)
            .query(params)
            .send()?;
// Handle the  different codes for response
match response.status() {
            reqwest::StatusCode::OK => {
                        // do something with the response
            };
            reqwest::StatusCode::UNAUTHORIZED => {
                        // show the error
            };

// Print the response/error
```

Writing src/tes.rs

```
pub use super::Service;
```

```rust
pub struct TES {
    service: Service,
}
impl TES {
    pub fn new(base_url: String, username: Option<String>, password: Option<String>, token: Option<String>) -> Result<Self, Box<dyn Error>> {
        let service = Service::new(base_url, username, password, token)?;
        Ok(TES { service })
    }

    pub async fn create(&self, task: &serde_json::Value) -> Result<serde_json::Value, Box<dyn Error>> {
        self.service.request("POST", "ga4gh/tes/v1/tasks", Some(task), None).await
    }

    pub async fn status(&self, task_id: &str, view: &str) -> Result<serde_json::Value, Box<dyn Error>> {
        let endpoint = format!("ga4gh/tes/v1/tasks/{}", task_id);
        let params = serde_json::json!({ "view": view });
        self.service.request("GET", &endpoint, None, Some(&params)).await
    }

    pub async fn list(&self) -> Result<serde_json::Value, Box<dyn Error>> {
        self.service.request("GET", "ga4gh/tes/v1/tasks", None, None).await
    }
}
```

The remaining classes: aai.rs, trs.rs, drs.rs, wes.rs will be made similarly, calling the API similarly with the added task of authentication.

Authentication:

The users are redirected to a webpage, and their login token is obtained and stored locally, which is used everytime it needs to be used while calling the API

The code for login and authentication is as follows:

a. The user is redirected to a webpage, where they enter their credentials, and a token is generated, which is saved locally. This is done in python, and the process is written in the next section.
b. Then we simply retrieve the token from the file by adding the following function:

```rust
fn get_token() -> Option<String> {
    let token_path = Path::new("~/.ga4gh-cli.token");
    let mut file = match fs::File::open(token_path) {
        Ok(file) => file,
        Err(_) => return None,
    };
    let mut token = String::new();
    if let Err(_) = file.read_to_string(&mut token) {
        return None;
    }
    Some(token.trim().to_string())
}
```

c. Then, we simply add headers with the token while calling the functions:

```rust
let token = get_token()?;
let headers = [("Authorization", format!("Bearer {}", token))];
```

An interactive Python Command Line Interface

The python cli will have the following functions:

i. **Constructor:** __init__(self, config): constructor, which initializes the CLI object
ii. **Input:**_process_input_file(self, input_file, params): This method processes an input file containing a JSON task.
iii. **Authentication:** A login page functionality, partially implemented here, which redirected to a login page, where their login and password is entered by the user, and a token is generated, which is saved locally, which can be retrieved by the rust libraries for API calling
iv. **Calling respective functions using API's:**get_aai_instance(self): This method returns an instance of the AAI class, initialized with the base URL, username, and password from the configuration, using the maturin to develop and eventually call python functions in the rust library. **Explained in more detail at the end of this section.**
v. aai_login(self): This method logs in to the AAI instance using the credentials from the configuration and stores the authentication token.
vi. get_tes_instance(self):returns an instance of the TES class, initialized with the base URL, username, password, and authentication token from the configuration.
vii. tes_create(self, task_file, params): This method creates a task using the TES instance. It processes the input file, replaces placeholders, and then creates the task

viii. tes_status(self, task_id, view, attest): This method retrieves the status of a task from the TES instance.
ix. tes_list(self): This method lists all tasks from the TES server by calling the corresponding method in the TES instance.
x. wes_create(self, workflow_file, params), wes_status(self, workflow_id, view, attest), wes_list(self): similar to TES and AAI, these allow us to access the different methods of the wes api
xi. drs_get(self, object_id), drs_put(self, object_file, params), drs_delete(self, object_id), drs_list(self): these allow us to access the different methods of the drs api
xii. trs_get(self, tool_id), trs_put(self, tool_file, params), trs_delete(self, tool_id), trs_list(self): these allow us to access the different methods of the TRS (Tool Registry Service)

The basic version of this is already implemented in
https://github.com/genxnetwork/ga4gh-cli/blob/master/src/cli.py

Integrating and calling functions from the rust module using Python bindings:

Step 1: First, install maturin using `$ pip install maturin.`

Step 2: Create a new folder called Python, with a Cargo.toml file and the directories src/, which contains a function calling all 5 api's.

Step 3: Format the functions using this format:

```
use pyo3::prelude::*;

<<functions>>

#[pymodule]
fn my_project(_py: Python, m: &PyModule) -> PyResult<()> {
   <<function_declarations>>
   Ok(())
}

<<tests>>
```

Step 4: Then run `$ maturin develop`

Then, the functions can be called directly from the cli by importing the file.

Reference: **https://medium.com/@MatthieuL49/a-mixed-rust-python-project-24491e2af424**

Auto Code Generation from OpenAPI specs to support multiple GA4GH API versions

Code can be generated automatically using openapi-generator, using API specs. OpenAPI Generator is an open-source tool that generates applications based on OpenAPI v2.0/v3.0 documents. It can create code for client libraries, server stubs, documentation, and configuration. OpenAPI Generator automates much of the API development process.

We can automate our API generation process using the following steps. This creates ready to use rust code, as I did here: https://github.com/aaravm/tes-library-rust-autogenerated

The code can be Generated automatically by:

Step 1: Run the command in the terminal, where tes.yaml has the openapi specs of tes for generating the library for TES. Similarly, run the command for trs.yaml, wes.yaml, drs.yaml

```
openapi-generator-cli generate -g rust \
-i /home/aarav/dev/openapi/tes.yaml \
-o /home/aarav/dev/openapi \
--additional-properties=useSingleRequestParameter=true
```

I did this with the TES open api spec, and I generated this:
https://github.com/aaravm/tes-library-rust-autogenerated

Step 2: Understand the code, and add majority of the code in the main classes to make the calling of API safer, error free, efficient, and increasing modularity and performance.

For example this is the code generated for TES create Task, that can be added in tes.rs:
https://github.com/aaravm/tes-library-rust-autogenerated/blob/d1de1a48215d4a8d9358a7dbfe80abd103885d59/src/apis/task_service_api.rs#L14

Step 3: Using the generated code, make the documentation and add the subsequent step by step procedure for changing the version of the API

Reference: https://www.twilio.com/docs/openapi/generating-a-rust-client-for-twilios-api

Cross Language Bindings in Python, C, Go, and Javascript

1. **Creating bindings in C:** Using cbindgen tool to generate C bindings for a dynamic library (cdylib)

   Step 1- Create a new folder called C, with a Cargo.toml file and the directories src/, which contains a function calling all 5 api's.

   Step 2 - Install cbindgen using the command

```
cargo install --force cbindgen
```

   Step 3- Make the rust compatible for cbindgen to convert by doing the following steps:

   Add #[no_mangle] above the functions we want to call like tes.rs, trs.rs etc.

   Add extern "C" to the function definition to make sure the functions adhere to the C calling convention.

   Step 4- Add the following to the cargo.toml file:

```
crate-type = ["cdylib"]
```

   Step 5- Run the following command to build the c header file:

```
cbindgen --config cbindgen.toml--config cbindgen.toml --lang c --crate lib.rs --output bindings.h
```

   Step 6- Include the bindings in the c code, and call the functions for writing tests in the tests folder. Add a Readme for documentation

Reference- [Rust FFI and cbindgen: Integrating Embedded Rust Code in C - DEV Community]

2. **Creating bindings in Javascript:**

   Step 1: Create a new folder called C, with a Cargo.toml file and the directories src/, which contains a function calling all 5 api's.

   Step 2: Add #[wasm_bindgen] to the top of the functions just defined

Step 3: Add this to the Cargo.toml file:

```
[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2"
```

Step 4: Run the following command:

```
wasm-pack build --target web
```

Step 5: Create an index.html to finally call the functions, and add the following code:

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>tes-api</title>
  </head>
  <body>
    <script type="module">
      import init, { request } from "./pkg/hello_wasm.js";
     <!-- Call the functions here -->
    </script>
  </body>
</html>
```

Reference:
https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm

3. Creating bindings in Go:

To call functions in Go, the steps followed till the production of the header files (ending in .h) are enough. Then, Go can easily call functions using the cgo library directly from the go interface.

4. **Creating bindings in Python:**

Python bindings were already created, when using it to call the rust library in the CLI.

Confidential Computing Integration (stretch Goal):

Integration of Confidential Computing features can be done by using Trusted Execution Environments (TEEs).

Trusted Execution Environment is a secure area within a processor. It guarantees that the code and data loaded inside it are protected with respect to confidentiality and integrity. Essentially, TEEs provide a kind of ‘safe room’ for sensitive operations, ensuring that even if a system is compromised, the data within the TEE remains secure. Here, we use it to ensure that when the users enter their credentials, their safety is not compromised.

Applications inside the TEE are considered trusted applications. The data stored on and processed by TAs is protected, and interactions -- whether between applications or the device and end user -- are executed securely.

TEEs enable the following services:

- Secure peripheral access. TEEs can directly access and secure peripherals such as the touchscreen or display, offering protection for fingerprint sensors, cameras, microphones and speakers.
- Secure communicatioo with remote entities. These environments can secure data, communications and cryptographic operations. Encryption private and public keys are stored, managed and used only within the secure environment.
- Trusted device identity and authentication. Some TEEs use Roots of Trust, which enable the legitimacy of a device to be verified by the connected service with which it is trying to enroll.

Timeline

I propose the following timeline for completing the milestones of this project. I believe I will require less time than I have outlined, and therefore I hope to work on the stretch goal of confidential computing and hopefully complete it as well

- **Community bonding:** May 1 - May 26

In this period I will be in contact with my mentors to set up communication channels and work policies. Furthermore I will be focusing on getting the project basics set up. This includes:

- Working with my mentor, and discussing with the community to any proposed change in the project proposal
- Setting up the repository on github, making the basic structure of the files, setting up the major divisions and adding the required files.
- Setting up the workspace environment for the project locally

Phase 1

- **Week 1:** May 27 - Jun 2
    - Create the base class of service.rs (**MS1**) in the src directory, which is responsible for interacting with the API's
    - Add a test folder, where the services.rs can be tested, and made sure the correct output is returned
- **Week 2:** Jun 3 - Jun 9
    - Create the basic python library of command line interface and implement the authentication and token retrieval webpage (**MS3**)
    - Create a basic rust library on top of services.rs that retrieves the token and passes it to the API headers to retrieve information
- **Week 3:** Jun 10 - Jun 16
    - Create the basic versions of the 5 API libraries on top of the services.rs and make sure all the libraries are able to call their respective functions (**MS2**)
    - Add tests to make sure the right output is being obtained
- **Week 4:** Jun 16 - Jun 23
    - Create bindings for the rust code (**MS5**) to make sure it can interact with the CLI using the pyo3 and maturin
    - Add tests to make sure the right output is being returned
- **Week 5:** Jun 24 - Jun 30

    - Complete the python cli to make it easy to interact with the users, and completely integrate the generated bindings the python cli, to make a basic version of the tool completely (**MS4**)
    - Take help of the previously implemented command line interfaces to add features in the python cli
    - Add documentation to make understanding of the workflow easier and easier to install and use the cli.

- **Week 6:** Jul 1 - Jul 7
    - Complete the making of the python CLI.
    - Generate code from the openapi-generator using the specs of all the api's, and generate the airtight, modular and error free rust code (**MS6**)
- **(Phase 1 evaluation):** July 8 - July 12

  Deliverable:

  A user accessible command line interface, that is able to talk with all of the API's, and is able to handle the responses received, returning the responses correctly to the user.

Phase 2

- **Week 7 - 8:** Jul 8 - Jul 21
    - Integrate the auto generated code to the main library to complete the rust library complete and error free, handling all the cases and exceptions
    - Add documentation to the code, to ensure support multiple GA4GH API versions and make sure that updating the library due to changes in the API are very easy to change and update and ensure version compatibility (**MS6**)
- **Week 9:** Jul 22 - Jul 28
    - Create bindings with C and Go using `cbindgen` tool crate like `cdylib`, and the `cgo` library (for go) to allow users to call functions from C and Go (**MS7**)
    - Add tests to ensure that the bindings are returning the correct output or not
- **Week 10:** Jul 29 - Aug 4
    - Create bindings with Javascript using WebAssembly, and creating basic index.html to allow users to call functions from JavaScript (**MS8**)

    - Add tests to ensure that the bindings are returning the correct output or not, and to ensure that users are easily able to access the library

- **Week 11:** Aug 5 - Aug 11
    - Comprehensively add the documentation of using the tool, and ensure that all the areas have been documented properly(**MS9**)
    - Completely check if all the functions are being tested and ensure all the functionalities are being tested or not.
- **Week 12:** Aug 12 - Aug 18
    - To wrap up any unfinished tasks or deal with unexpected delays, I intend to reserve this week as a buffer.
    - Ask the community for any suggestions on how to improve the project, and discuss any potential changes or additional features to be added(**MS9)**
    - Do the finishing touches in the project
- **Week 13:** Aug 19 - Aug 25
    - Preparing a short report on the project
    - Preparing a short slide deck with intro/background/motivation, results/implementation, discussion/outlook (3-5 slides)
    - Posting about it on LinkedIn and on Medium
- **Final submission: Aug 26**

  Deliverable: The final submission will be a comprehensive rust library + command line interface for communicating with GA4GH API's, with easy to upgrade and maintain different versions of API's, and cross language bindings to access the functions in other languages like C, Python, Go and JavaScript, with all tests and documentation being uploaded on github.

Availability

I will be available throughout GSoC and I have no other commitments during this time. I will be devoting about 35 hours a week to my goals during GSoC with GA4GH. I will always be

available through email and slack. If I am behind my proposed timeline, I will work overtime for the next few days to make things go according to the plan.

Post GSoC

Being part of such a collaborative and welcoming organization is a great opportunity for me to learn more and contribute. So, I have already planned to contribute more to the organization even after the GSoC period, and also work on the stretch goal if it is not completed. I will also apply to become a mentor next year, as I believe it will be a great opportunity to pass on everything I have learned during the summer. I'll be an active member of the community and keep contributing to the organization

References

- What is GA4GH?
- GA4GH
- https://github.com/ga4gh/ga4gh-client/