Rust Code Analysis as a Service

Final report

Rust Cleaners

Arsh Kabarwal

Joseph Carpman

Kipp Corman

Olivia Ornelas

Department of Computer Science

Texas A&M University

05/05/2022

Table of Contents

1 Executive summary

With the emergence of smart contracts and programs on the blockchain, there is a learning curve for creating secure contracts and programs. It can be difficult to understand and comprehend all the security measures that need to be present in the programs. Our service will help educate developers with many of these pitfalls by evaluating the errors and providing easy to read errors. Since code that is submitted to the blockchain can not be amended or changed, it is important that the code is secure so malicious attackers can't take advantage of the flaws in the unsecure code. In addition, many Rust developers fail to find security errors or runtime flaws in their code which leads to vulnerabilities. They may not have the time and storage to manually download static analysis tools and run the tools on their code.

The Rust Code Analysis as a Service was created to help the average developer by allowing users to quickly and clearly review the security issues with their Rust code. This service is a free service which allows developers with financial constraints to use the service. The service provides easy to understand results that makes debugging quick and hassle-free for Rust developers that want to deploy to the blockchain or simply have a secure error-free project.

The project had a slight pivot from Solana Code Analysis as a Service to Rust Code Analysis as a Service. This is because Soteria, a tool specifically for Solana smart contracts, wasn't implemented in time and there were no other open source static analysis tools specifically for Solana smart contracts. However, this service can still be used for smart contracts. The final user interface includes a zip file upload/repo link upload feature, auto-email functionality, and a code report with highlighted lines. The biggest focus of this project ended up being on providing a valuable user experience so that it would make sense to use this product over downloading and running the tools through the command line.

The final result of the project was a Software-as-a-Service tool that contains two major Rust static analysis tools: Clippy and Cargo Audit. However, the project was designed to be able to seamlessly add additional tools or support new languages by simply converting tool outputs to SARIF format. The user can upload a zip file or submit a GitHub repository, The user can then choose which tools they want to run on their code using a checkbox functionality. If the user doesn't want to wait for their report, they can enter their email address. Once the report is fully generated, the email will then be sent to the user. The errors are shown clearly by allowing the users to click through the errors. The user can view the Clippy errors either by a sorted priority level or by file. There is also a file tree that displays where the error lies, so a user can easily spot the errors.

The team primarily used GroupMe for correspondence and Discord during voice calls, screen sharing, and sending relevant articles. For reports, slides, and other team documentation, the team used a shared Google Drive. Bi-weekly meetings were conducted which focused on discussing each member's availability to the project for that week, any potential obstacles that a team member faced, and divided up tasks. Gantt charts were also used to track progress during the production cycle. This was also useful to make sure the team was completing tasks on schedule. Although this is more of a looser management style, the team remained accountable and were able to meet deadlines on time.

2 Project background

Describe the general scope of your project, and the specific problem that your project tackles. Go from general (e.g., search and rescue robotics) to specific (e.g., GPS navigation). This section (and all its subsections) can be expanded from those in the proposal.

2.1 Needs statement

The current barrier to entry of blockchain smart contract development is too high. The difficulty of modifying or updating deployed smart contracts demands robust code review tools to catch programming errors before deployment. Currently, code review tools that analyze code meant for blockchain requires the user to download the tool. Additionally, most tools require the user to send portions of their code directly to the company in order to get a quote for how much the tool will cost. Our project aims to create a low cost platform that doesn't require the user to download anything in the form of Software-as-a-Service but still provides static analysis of tools, mainly for Rust programs. Rust was chosen since it is the primary programming language of Solana smart contracts.

2.2 Goal and objectives

The goal of this project is to implement a Software-as-a-Service tool which allows users to quickly and clearly review their code before it is deployed to the blockchain. This tool should be as user friendly as possible to lower the very high barrier of entry to smart-contract deployment. The minimum viable product would allow users to submit and run their code, and then visually show what lines have potential security/performance issues. In addition to this, the product will review a user's code using multiple tools and present the results in an easy to understand way. Additional features we implemented are an interface that highlights code where errors are prevalent, a progress report that allows users to sign up for an email notification. Due to the time constraints and unforeseen drawbacks, an area to view history and progress over time was removed changed due to issues with logging and not have user accounts

Objectives for this project include the ability to reduce the cost of using a blockchain code review tool by 20% and identify at least as many vulnerabilities in their code as the existing products do. Because we couldn't test current tools in the marketplace, it was hard to determine, however through some research we learned that our tool could've prevented wormhole attacks by revealing vulnerable dependencies or functions.

Additional objectives include:

1. The product will be able to review code using at least 3 code review tools and produce results from each on the same dashboard.
2. The product will be able to accept code from at least file upload and github url.
3. The product will preserve up to 10 prior submission**s** for comparison. A comparison feature wasn't implemented due to the inability to create user accounts and time constraint.
4. The product will preserve code review report data for historical modeling. This criteria was met to an extent because the code review report is stored but it allowed for faster retrieval of results and an ability to share the results via custom website link.

2.3 Design constraints and feasibility

One constraint in our project is that we are limited to using open source static analysis tools that already exist. We do not have the time or expertise to create our own, which means we are relying on others. For our objective of showing as many problems in the submitted code as existing tools, this entirely depends on how effective the tools we end up finding are. After looking and reviewing tools, only a couple open source tools provided human readable feedback that was unique from other implemented tools. Another constraint will be a time constraint. This project has to be completed within about 3.5 months which can cause a lot of features and functionalities to be left out or uncompleted. Finally, there is

a potential financial constraint. If this product ends up gathering a lot of users over time, there would be additional costs to start up more servers so that the users don't have extremely slow responses. All of their past entries would still need to be stored, and over time with people possibly submitting entire codebases this could become quite large too.

2.4 Literature and technical survey

- Bloqchain Science Code Review
  The Bloqchain Science Code Review[1] is an automated tool for the static analysis of the source code which provides a detailed security code analysis. This tool supports many coding languages except Rust. The Solana Code Analysis as a Service will focus primarily on dealing with Rust. In addition, the Bloqchain Science Code Review requires a developer to contact the owners of the tool in order to utilize the tool. The Solana Code Analysis as a Service code review will allow a user to submit their code and immediately receive results, making the process faster and smoother. The Bloqchain Science Code Review is geared towards businesses rather than to regular developers or coders which Solana Code Analysis as a Service will target.

- Soteria
  Soteria[2] is a security services and audit tool specifically for Solana Programs. Soteria can be used to detect vulnerabilities and tell the user why a specific command was flagged or where the code is vulnerable. Solana Code Analysis as a Service will be faster than Soteria. Whenever source code is submitted to the Soteria application, the earliest time range the results will take is 24-48 hours even for the simplest code. Solana Code Analysis as a Service will have a GitHub auto-update functionality that Soteria doesn't. This will allow the user to easier see how the tools are affecting their code. The issue with Soteria is that it is in an early beta stage and the command line tool didn't work properly.

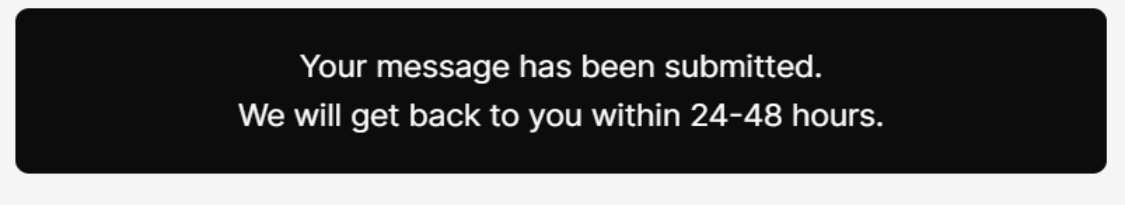

Figure 1 - Image of Soteria's message after code is submitted

- Anchain.AI
  Anchain.AI[3] is an AI-Powered Blockchain Intelligence Platform for security, risk, and compliance. This service requires a user to request a demo in order to use the services. This service would be useful if needed for a large scale project that needs an immense amount of inspection and auditing of the safety of the smart contracts. Solana Code Analysis as a Service is more feasible for the average developer in getting quick and reliable analysis. The Solana Service will skip hurdles that large scale development projects can't.

- SonarQube
  SonarQube[4] is a tool for inspecting code security and code quality with the use of code reviews. SonarQube utilizes clear visuals and dashboards that display vulnerabilities and bugs. Rust is not one of the 29 languages that this service evaluates. Solana Code Analysis as a Service will use similar visual elements to SonarQube, but it will be different because it is specifically used for Rust and will be for code that is deployed to the blockchain. SonarQube has a Community

version that is free, but this feature can be difficult to users with novice experience. Solana Code Analysis as a Service will be very user-friendly and super easy to use.

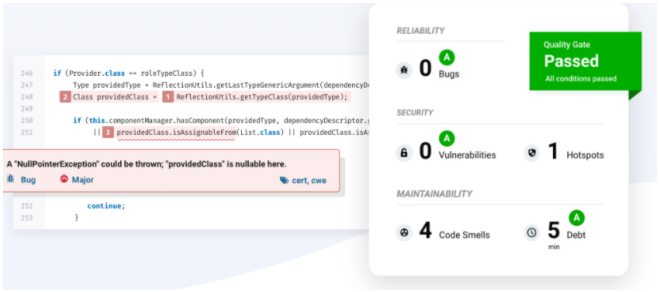

Figure 2- Image of SonarQube's Code Analysis Dashboard

- DeepSource-
  DeepSource[5] is a fast and reliable static analysis platform that allows for self-hosted development. DeepSource analyzes every pull-request made in the user's provided repository. It generates solutions for the bug fixes automatically. The service works with all of the languages used for Solana. It is also free for personal accounts and small teams, but it costs money for any teams larger than three. Solana Code Analysis as a Service allows for easier flexibility on how users want to use the static analysis tools. Since Solana Code Analysis as a Service uses many static analysis tools, there is a larger chance that an error is detected than just one tool which DeepSource uses. The Solana Code Analysis as a Service also allows code to be submitted directly without a needed repository on GitHub, GitLab, or Bitbucket.

2.5 Evaluation of alternative solutions

- Alternative solution 1: Create a dashboard that allows users to login, upload code, or submit a github repo URL to be reviewed. On this platform, the tool will use different types of open source code review tools (primarily for Rust) to analyze the code and show where vulnerabilities or bugs are within the code and suggestions on how to fix them. In addition to this, the tool would also be able to keep track of that file or repo's review history since the user logs in. A downside of this solution is that it requires the most user data to be stored within the project's data servers. As a result, the tool will have higher security needs and an increased negative environmental impact. Despite this, it is the best solution because it would allow the user to easily access their previous versions of reviewed code, has easier access to Github repos, and aims to make the code review process more personalized than the other solutions.

- Alternative solution 2: Allow users to submit code in any language that is popular on the blockchain. This would cause our product to potentially gather a larger user base, but require more work before it would be up to the same standard. Since there are so many existing solutions that do static code analysis, it is important that our project is able to do a good enough job that someone would want to use it over those alternatives. Focusing on just 1 (and maybe 2) languages that are used in Solana lets our project focus on using many analysis tools to create a quality product.

- Alternative solution 3: Removing the login feature which wouldn't allow the web application to store the user's upload history and prevents the tool from connecting to GitHub efficiently. Doing this would be beneficial because users wouldn't save time not having to login and their data

wouldn't have to be stored on the tool's servers. This can save money, complexity, and lower the environmental footprint of the tool. Conversely, removing the login feature would make connecting to Github more complicated, requiring users to constantly re-login, and wouldn't allow the tool to effectively store the user's previous history using the tool on different versions of the code submitted.

- Alternative solution 4: Create a new software code analysis tool instead of using open source code analysis tools online. For this project, using already created tools made more sense because of the time constraints and to help differentiate from other existing products that we analyzed above. All of their products implemented their own static code analysis rather than automatically running lots of existing open source tools. This solution also allows our service to show what percentage of open source tools found certain vulnerabilities to help determine how accurate it is.

- Alternative solution 5: Implement dynamic analysis. This will allow the user to see how their smart contract is performing while being executed. Dynamic analysis can provide significantly more feedback on the security and execution of the application on the blockchain. A con of this solution is that it can be difficult to trace vulnerabilities in a runtime solution. Many of these vulnerabilities will be very difficult to be changed because code on the blockchain is immutable. It would be more costly to fix the errors than just simply creating a new amended smart contract.

3 Final design

3.1 System description

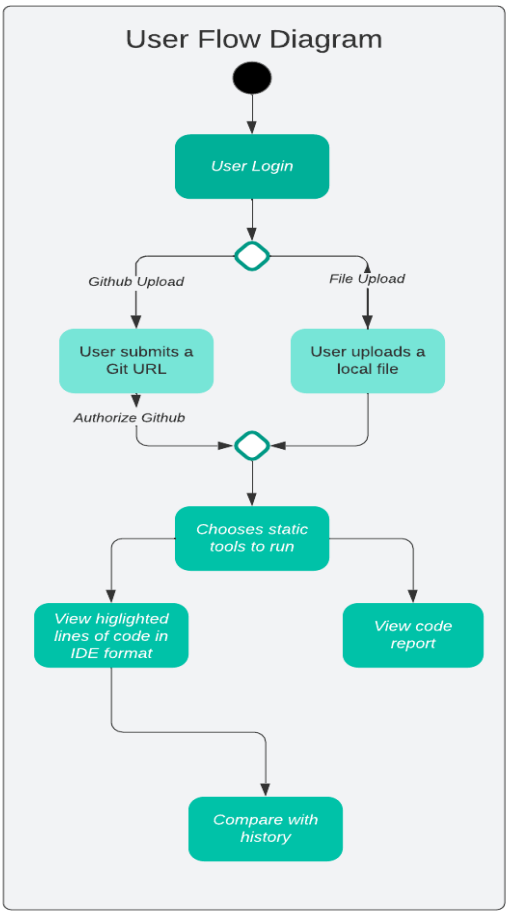

Figure 3 - Image of CDR User Flow Diagram

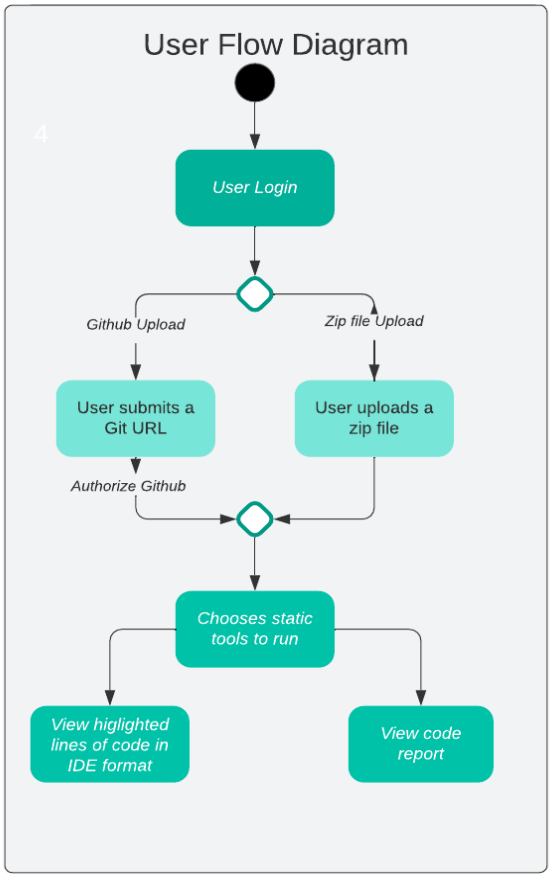

Figure 4 - Image of Final User Flow Diagram

The first image above was the user flow diagram for the Solana Code Analysis as a Service project for the Critical Design Review. The following description is the description written at the Critical Design Review. An added component is the ability for the user to choose which static tools they want to use. There will be a checkbox functionality that displays all the available tools they can run. If multiple tools are chosen on the same code, a carousel feature will enable the user to toggle between the different results from the different tools. The user will be able to comfortably see where most of the errors and vulnerabilities lie. A code report will also simultaneously be created which will establish how secure the code is. The raw results are stored inside our database to provide faster rendering of results if the same commit of a repo is being analyzed again.

The second image above is the final user flow diagram. Everything stated in the Credit Design Review was implemented except for the compare with history feature. This was not implemented because of the time constraint. Due to time and practicality, the user doesn't have the option to login in order to use the tool. Instead a landing page was created that allowed the user to click ‘get started’ which redirected to the input page and option, depicted in the diagram/

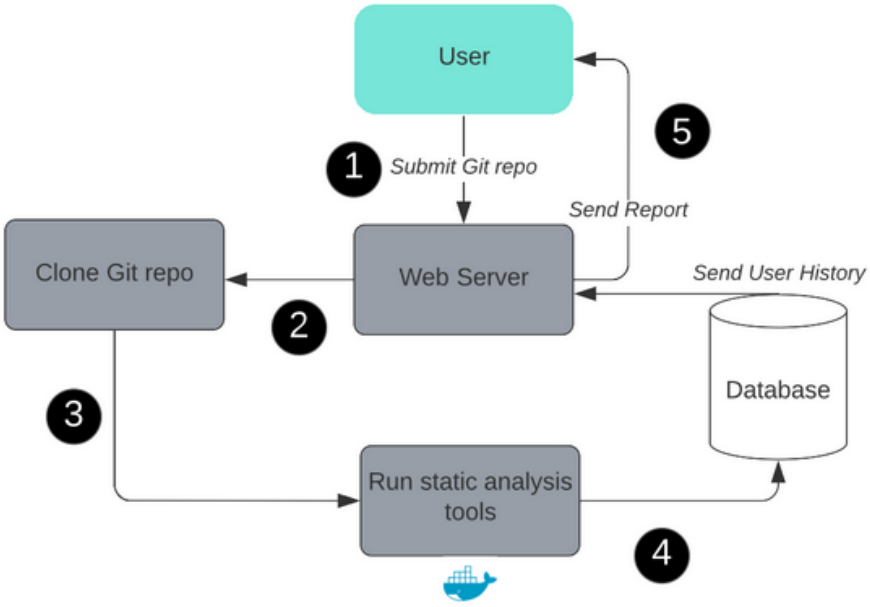

Figure 5 - Image of Final Product Flow Diagram

The image above is the updated system design diagram for this project. The main difference here is that we have decided against using a separate server to handle the code analysis tools due to simplicity and the goals of our project (everything in gray is on the same server). The original reasoning was to create a more distributed system so that it could scale better, but in the short time span of this project we feel that our time is better spent adding more features to the website. If additional scalability is needed, we plan to horizontally scale by using more AWS servers rather than distributing our tasks across multiple systems.

As before, the web server was hosted with AWS Elastic Beanstalk and includes a Docker container that runs all of the code analysis scripts. Once the user sends in their Git repository, the server will assign their request an ID and keep track of what Git repository they are running, the current commit, and which tools they chose to run. This information is stored in a database table in AWS DynamoDB. Then, the server checks the second database table to see if another user has already done analysis on the same commit. If not, it will clone the code locally, run all of the code analysis tools on it, and save the results in our database. The database will store the commit ID and Git repo being analyzed so that we can match it up to past requests made for the same repo, as mentioned earlier. This is useful in order to cache results to avoid rerunning the same exact commit ID multiple times for different users, and to help us show more useful information by comparing with past versions that the same user has already run. The web server will then pull these results and display it in an easy to read format. This format will look similar to an IDE where the file directory is displayed on the left, and by clicking on individual lines of any file you are able to see what warnings are coming from each section of the code. There is also an additional section that focuses on the type of warning rather than what is in each file, giving the user the freedom to use whichever view they would like. Finally, all of this information as well as a summary of their code will be returned to the user.

Another important feature of our database is not having to wait for the analysis tools to run. Once a request is made, the ID is automatically added to the URL, and it will pull the results from the database

when the request is complete - even if the user shuts down their computer or leaves the page. Additionally, they can choose to receive an email with this URL once it is done.

3.2 Complete module-wise specifications

The Rust Code Analysis as a Service uses a Docker container to contain all of the dependencies to run all the different tools. This saves plenty of time for users by preventing the need to download every dependency to run the different tools, and allows us to easily add additional code analysis tools by just adding them to our container. To run all of the tools on the user provided code, bash scripts are used to run every command in a command line interface. This automates the process of manually typing out the commands. The bash script locates where each static tool should be run. For example, the cargo audit tools need to be run where there is a Cargo.lock file and clippy needs to be run where the rust files are. The script stores the output of each tool in a different json file.

Then, Javascript is used to read through the json files and print out all the key information in a readable format for the user. Since the results from the tools are all stored into json, the key-value pairs are run through a for loop to store the most important information. A javascript file is created for every static tool to read the output of the tool. Therefore, each tool has a json file, where the tool's output is stored, and a javascript file, where the output is parsed through.

At the completion of this project, the user can choose between Cargo audit and Clippy. Cargo audit analyzes security vulnerabilities for crates in Cargo.lock files. Clippy interprets and catches mistakes in Rust code. After some research and attempting to incorporate Soteria, a tool that identifies and eliminates security vulnerabilities built specifically for Solana contracts, it was determined that UI design and modifying Clippy and Cargo audit to account for potential edge cases was more important. This was mostly due to the nature of Soteria and how its setup was different from the format of the other two tools. Another tool we started integrating towards the end was Mirai, a tool that can be used to determine overall correctness, catch potential panics from within the program being executed, and find security bugs. If more time was available, this tool would have been included in our final product.

The front end is done using React. React is used to create the website for the service. The website currently takes in a GitHub url and clones the repo using a middleware function to automatically download the Github repository. Prime React is used for user interface components for the website. It was used to output a directory tree of the uploaded repository. The directory tree allows the user to click on different branches of the repository to see where different files are stored. The checkbox for the analysis tools was also created with Prime React. Future user interface elements will also use Prime React to ensure continuity.

The website is hosted on Amazon Web Services. AWS allows the website to be viewed with a specific link. The database to store all the information will also be stored on AWS. DynamoDB will specifically be used because it is a noSQL database that is included in the AWS free tier. Although the data we plan to store is technically structured, the main thing that will take up space is just the user's code and the code analysis results. With lots of users, this could end up being a lot of text and noSQL is better for scalability. In our database we will be using the Git repo (partition key) and commit ID (sort key) combined for the primary key. The partition key ensures that anything with the same value will go on the same partition, so by using the Git repo we will be able to easily check if any other commits for the same

repo have been run before without looking across database partitions. The database objects will store information about the Git repo being analyzed as well as what code analysis tools were run and what the output was. Since some of the tools take a long time to run, and we plan to only add more, having a caching system is important. Finally, though using the database in this way greatly reduces the runtime and improves our overall user experience, some users may not want their results to be stored, especially if their code is not public. The user will have the option of not storing anything, though that also means that all added functionality, like faster result loading, will not be available.

Our original plan was to store all of the code along with the code analysis results together in our database. However, this would use a lot of space in our database and likely exceed the free tier as many repositories include thousands of lines of code, many of which may not even be in Rust. For this reason we decided to only store the analysis results, and pull the Git repo each time even if it has been run before. The time it takes to re-clone a repository is also much less than the time it takes to run the tools, so it should not impact the user experience very much.

Figure 6 - Image of Submission Database Object

The first database table is shown above. It's purpose is to associate an ID with each request made by a user so that the server will know where to pull results from when the user comes back to the page at a later date. For example, if they choose to sign up for an email notification, it will include a link with their ID and the server will check this table to see what repository it needs to clone and get the results for.

```
{
  "gitRepo": {
    "S": "ClementTsang/bottom"
  },
  "commitID": {
    "S": "9882a9bd088abbaad51e4ce8564293cff34fde3d"
  },
  "toolOutputs": {
    "L": [
      {
        "M": {
          "name": {
            "S": "Clippy"
          },
          "output": {
            "L": []
          }
        }
      },
```


Figure 7 - Image of Analysis Database Object

In order to get the results, it will need to contact our second database table which is shown above. This table keeps track of the raw tool output for everything that has been run on a given repository before (the Clippy output has been excluded because it is so long).

The reasoning for splitting up the tables like this is to keep track of what tools each user wants to run, otherwise they may see results from tools that an earlier user chose but they did not.

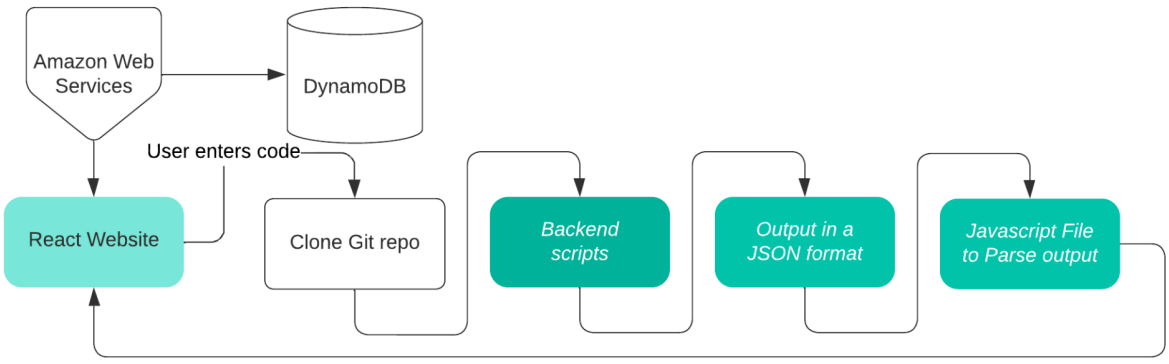

Figure 8 - Image of Subsystem hierarchy

3.3 Approach for design validation

How did you test your system to ensure that it did what it was designed to do? If these validation procedures were revised during the semester, please document the changes and provide a justification.

Most of the project's goals are quantifiable and easy to test. Different GitHub repositories were used to test the tools and the highlighting functionalities. These repositories consisted of repositories with Solana smart contracts as well as repositories with basic Rust code. Using different repositories allowed the team to test edge cases for the tools.

The file upload and github upload tasks were tested by submitting files of different lengths along with identical files hosted on github. This feature was validated if the result of each file is identical to it's github counterpart. The result must also be non-empty to test that the upload is working correctly. On the live build, the team made sure to not require that identically named files be the same, because a local version can be different than the repository version.

The emailing functionality was tested by using a test email address. If the correct report was emailed to the test email address, this would show that the functionality works correctly.

To ensure that the overall design is effective and does not have security vulnerabilities, the website was put under a moderate load with a couple users trying to utilize tools on the different repos. If more time was available, the website would be put on a larger load to see how it would fare. An upgrade in the container size would most likely be needed for a larger mass of users. Validation was also done to verify that the user's code is safe.

4 Implementation notes

Docker Image:

Creating a Docker image with all analysis tools included was the first step in creating the application. By doing this first, anyone working on the project can simply pull the image and automatically have all necessary tools installed on their machine. Additionally, when hosting the website the tools do not need to be downloaded once again.

After downloading Docker and becoming familiar with the basic commands, a Dockerfile was created based on the official Ubuntu image. Then, commands were added to the file to install the necessary components for running Rust analysis tools. Finally, the individual tools were added to the file. There were usually more installations required than what was listed on the website, and if someone were to recreate this product I would recommend using ours as a starting point.

```
FROM ubuntu:18.04
MAINTAINER kipp
# Update
RUN apt-get update
# Install node.js v14
RUN apt-get -y install curl && curl -sL https://deb.nodesource.com/setup_14.x | bash && apt-get -y install nodejs
# Install Rust
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
ENV PATH="/root/.cargo/bin:${PATH}"
# Install Git
RUN apt-get update && apt-get -y install git
# Install clippy
RUN apt-get update && apt-get -y install build-essential
RUN rustup update && rustup component add clippy
RUN cargo install clippy-sarif
# Install cargo-audit
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get -y install pkg-config
RUN apt-get update && apt-get install libssl-dev
RUN cargo install cargo-audit
```

Figure 9 - Image of Dockerfile

Once a basic version of the React website is created, commands to copy the website code into the container and build/start up the application were added to the end. This allows the website to be accessed via localhost while being run on the container as long as the port that is being used is marked as exposed when creating the Docker container.

Running the Tools:

User input from ‘Body.js’ is sent to ‘Git.js’ (frontend) which will send post requests to ‘Server.js’ (backend). If the commit ID is already saved in the database, the result information is sent to a subfolder named based on what tool was chosen. Once done, a program is run to find all files in that directory for the tool, and place the data into a json object array so it can be better interpreted by the frontend. Otherwise, the tools are run using bash scripts.

```
#finds where Cargo.lock is and then cuts Cargo.lock in order to run cargo audit on the toml file
CARGO_FILES=$(find downloads/ -name "*Cargo.lock" | rev | cut -c11- | rev | tr -d '\r')
SAVE_DIR=$PWD/downloads/rc_cargo_audit_results
mkdir -p -- "$SAVE_DIR" # make directory to save all files in
#goes through Cargo.toml files and runs cargo audit
I=0
for f in $CARGO_FILES
    do
        echo $f
        cd $f
        cargo audit -n -q --json > $SAVE_DIR/cargo_audit_output$I.json


        (( I++ ))

        cd -
done
```

Figure 10 - Image of Cargo Audit script

Because these tools provide additional output about how it runs, they are run with the '-q' quiet flag set to avoid seeing what the program is doing. For instance, pulling from the tool's personal database, building the program or other internal processes not useful for showing the results to users. Tools must be run in the same folder as Rust files (for Clippy) and .lock files (for Cargo Audit). Clippy will take longer to run because it has to build the program before reviewing the code. If someone wants to run the tools locally, follow download instructions from the tool's website or the product's dockerfile. One the tools are installed, run the line in the bash scripts that include the tool's name, replacing variables with the desired result destination.

Reading Tool Output

Tools chosen have a tag to produce the results in json format. This is important and used to create json object arrays which are looped through on the frontend side to display the results in a comprehensive way. Each script runs the specific tool and saves the output to .json files. An additional script (as needed) is used to find the tools and further format it so it's easier to interpret. Below is an example of what the bash script might look like.

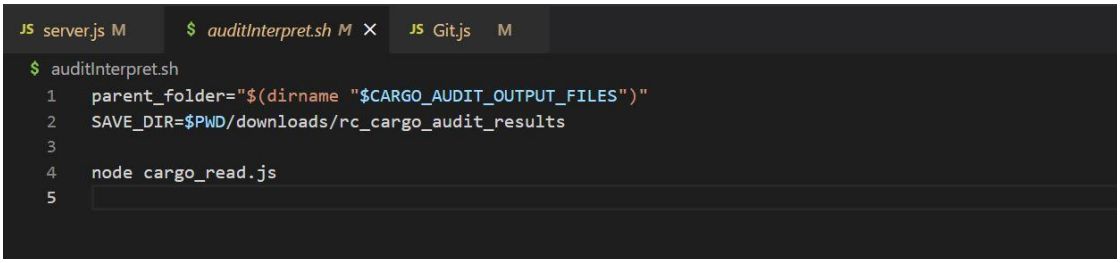

Figure 11 - Image of Cargo Audit Script used to interpret the .json results in an easier to read format

(i.e remove redundant brackets and only get relevant information about the problem and solution)

Because Clippy runs on each individual rust file, 'clippyjoin.js' is used to create a list of each output file location and its results. This is then placed into a file called clippy_output.json located in the 'downloads' folder. If one looked at this file, the data would be listed in order of runs where 'uri' is the file location. The output here is in SARIF format, so adding additional tools to our product is as easy as converting their output to SARIF and then using the same backend/frontend we have already created to display it in a readable format.

Figure 12 - Results from running cargo clippy

Cargo audit provides more cluttered data because the tool includes format designed to make it easier to read in a terminal. As a result, the .json file is more cluttered with large chunks of information in the 'description' section. Because of this, the information is sent to 'Git.js' and placed into a json object array based on the location of newline characters.

Frontend Design

Prime React was used for the interface elements on the website. The Rust Cleaner website has a routing menu which was from a Tab Menu template from Prime React. A hash router was used to link all the subpages to the homepage. There were three subpages: input page, documentation page, and the team

page. The icon in the menu is also linked to the home page. The checkbox was also created from a Prime React template. The file upload button, input text buttons, and normal buttons were Prime React components as well. The color theme for the prime react components is ‘saga blue’; more themes for the components are available on the Prime React website. The background for the project is .scss based but converted to .css and consists of small dots floating at a specific time towards the top of the screen. Credit for this design is provided in the ‘app.css file’. A loading bar is shown within a card component while the results load. After a unique Id is created for the git repo in the dynamoDB database, an option to receive an email notification for the finished results are provided. This is because the unique ID is displayed in the URL and the results will be saved to the database at the unique Id’s location. If a user wants to go back to that run’s results or share with colleagues, they’d be able to without having to run the entire project because the product is pulling the result data from a database instead of having to run everything again.

Figure 13 - Image of exported Checkbox feature

```
import React, { Component } from 'react';
import { TabMenu } from 'primereact/tabmenu';
import {Menubar} from 'primereact/menubar';
import './body.css';
import { HashRouter, Route, Link, BrowserRouter } from 'react-router-dom';

export class TabMenuDemo extends Component {

    constructor(props) {
        super(props);

        this.state = {
            activeIndex: 3
        }

    }
    navigateToPage1 = (path) => {
        window.location.hash = path;
        <Link to="/"></Link>
    }
    navigateToPage2 = (path) => {
        window.location.hash = path;
        <Link to="/input"></Link>
    }
    navigateToPage3 = (path) => {
        window.location.hash = path;
        <Link to="/documentation"></Link>
    }

    navigateToPage4 = (path) => {
        window.location.hash = path;
        <Link to="/team"></Link>

    }
```

Figure 14 - Image of Routing menu with implemented hash router

The displayed errors are shown with a Prime React accordion component. This allows the user to click on the warning title where they can see an opened error message. SyntaxHighlighter was used to highlight the error line. The SyntaxHighlighter takes in a startingLineNumber and an endingLineNumber in order to highlight a line or block of code. An overview of the warnings is also provided. Color and sorting is based on the categories at https://rust-lang.github.io/rust-clippy/master/ , the Clippy developer's website.

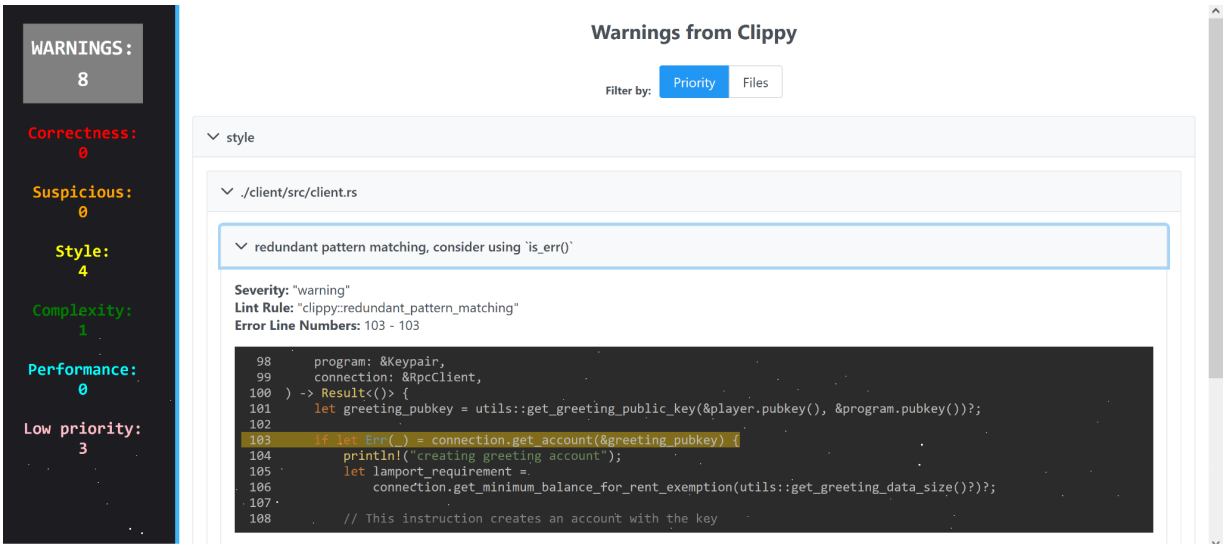

Figure 15 - Image of Accordian errors and Syntax Highlighting User Interface

```
<SyntaxHighlighter
    language={"rust"}
    style={darcula}
    startingLineNumber={y.lineNumbers - 5}
    showLineNumbers={true}
    wrapLongLines={true}
    wrapLines={true}
    lineProps={(lineNumber) => {
    const style = { display: "block", width: "fit-content" };
    if (lineNumber == y.lineNumbers) {
        style.backgroundColor = 'rgba(255, 204, 0, .40)';
    }
    return { style };
    }}>
        {this.codeSplicer(y.file, y.lineNumbers - 5, parseInt(y.lineNumbers) + 5)
    }
    </SyntaxHighlighter>
```

Figure 16 - Image of SyntaxHighlighter

Icons in a sidebar are used to navigate to different results panels, including the file tree which displays all files in the repo. When a file is clicked, that code can be seen on the panel warnings previously were. Cargo audit displays its results in an accordion format, based on crate dependency. This is to display multiple warnings for the same crate in one area instead of having repeating crates on the UI.

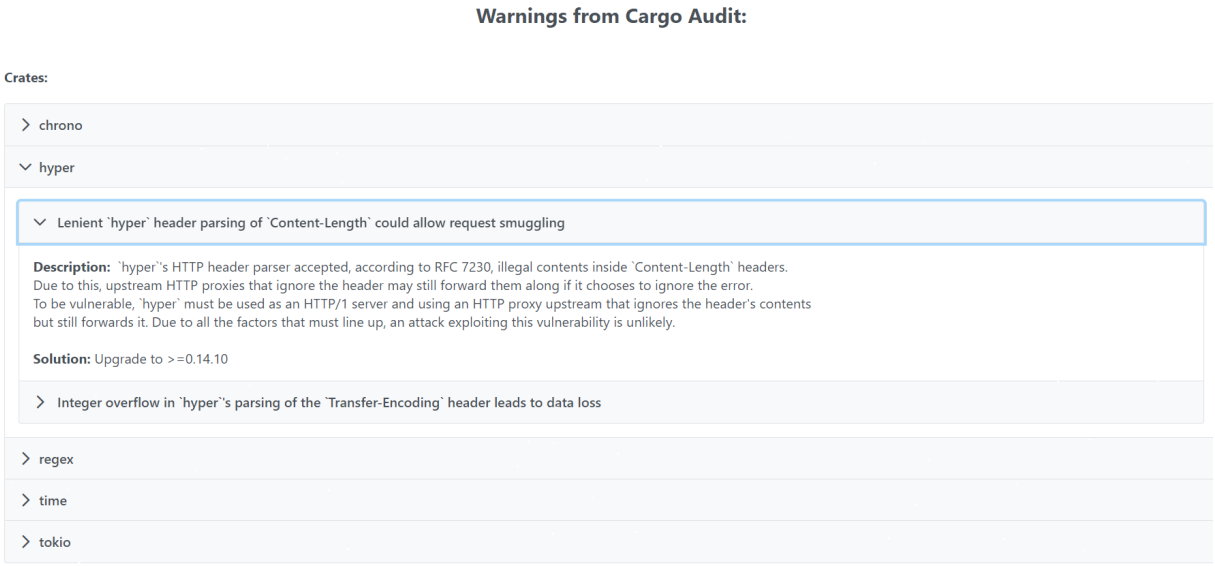

Figure 17 - Cargo audit results display

Cargo audit's data is placed in a .json object array in order to loop through the data efficiently. Because of its output, a 'new tool' string is attached to the start of each new crate in order to provide a simple way to split the Cargo audit output. This is because some new line characters were used as spaces in the tools original output. Cargo clippy had a similar feature but provided the filename in order to preserve that format when being sorted by priority. The ability for the repo's code to be visible is designed in the 'filetree.js' file. In this file, a filetree is created through the imported 'nodeservice' based on the files in the original repo. If there is data, a call to retrieve clippy's results is also made in order to incorporate highlight functionality with the actual code snapshot.

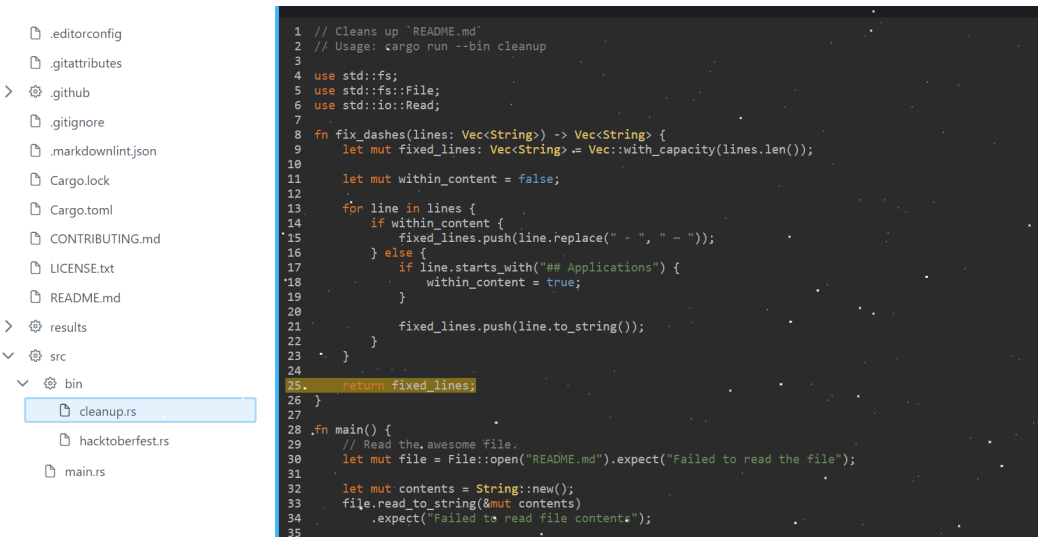

Figure 18 - Highlighted code in fileTree directory space

Front-end/Back-end Communication:

The backend server uses Express.js to create endpoints that the frontend can make requests to in order to send or receive data. Under each endpoint is the code that will be executed whenever a request is made to the specified URL, using the data provided by the frontend.

```
// When submit is pressed, generate an ID that can be used for email
app.post('/submit', (req, res) => {
    var url = req.body.repo; // save the url for use soon
    var tools = req.body.tools;
```

Figure 19 - Example of backend endpoint

```
fetch(server_Url + '/submit', {
    method: 'POST',
    mode: 'cors',
    credentials: 'same-origin',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ 'repo': url, 'tools': tools })
}).then((response) => {
    response.json().then((result) => {
        window.location.hash = `/results?id=${result['id']}`; // Change URL to include ID
```

Figure 20 - Example of frontend fetch

In the example above, the frontend makes a fetch request to the '/submit' endpoint after a user enters a GitHub repository and clicks submit to have their code analyzed. The frontend includes the GitHub URL entered by the user and all of the tools they checked to run in the body. This is sent to the backend so that it can make a new entry in the database which logs the user's request and associates it with an ID. Looking back at the frontend in the last few lines of code, the backend sends its result back and then the frontend is able to change its URL to include this ID.

This is the first step in the process of running the analysis tools for a specified repository that is done if the URL does not already include an ID (it would include an ID if accessed through email, or an earlier request). After this is done, a second fetch request is made to '/github' which first searches the database for cached results based on the ID, and then runs all of the tools if it doesn't find any previous runs. After being run and converted over to SARIF format, the outputs are sent over to the frontend to display.

Similarly, file uploads are done once the user clicks the download icon on the file upload button but before submit gets pressed. Once that is clicked, axios is used to post the chosen zip folder content (as a file object) to the backend. Once at the backend, a write is done into the 'downloads/' folder. After this, extract zip is called to extract the contents into 'downloads/'. Once the zip folder is downloaded, the program runs the same way it does for Git repos, without the ability to save to the product's database. This is due to how file uploads are done, and the lack of a commit ID for acting as a key.

```
app.post('/upload', (req, res) => {
    let file = req.body;
    fs.writeFile('./downloads/' + file.fileName, file.data,
        if (err) {
            console.log(err);
        }
        else {
            console.log("file downoaded, waiting on extract
        }
    });


    res.send("'finished':'done'");

});

async function extractZip(source, target) {
    try {
        await extract(source, { dir: target });
        console.log("Extraction complete");
    } catch (err) {
        console.log("ExtractZip failed:", err);
    }
}
```

Figure 21 - File uploads

There are also fetch requests created to retrieve the contents of a file whenever a user clicks on it in the file tree, and to create the directory tree but the logic is mostly the same.

Database:

DynamoDB was used to cache the tools analysis results so that users can view their results from a URL received in an email, and not have to re-run the same repository if submitted by multiple users. The first step in doing this was to understand how DynamoDB works. We created a table that uses the Git repo as the partition key and commit ID as the sort key, combined to make up the primary key. The partition key ensures that anything with the same value will go on the same partition, so by using the Git repo we will be able to easily check if any other commits for the same repo have been run before without looking across database partitions. Then, AWS credentials were stored in a file outside of the Git repository and 2 functions were made to send data to the table and receive data from it based on the key. An example of sending data to the database is shown below.

```
// Save output to DynamoDB
let save = async function () {
    // Wait for file read json for each tool
    await promiseClippy;
    await promiseAudit;
    await promiseMirai;
    console.log("we are done with file read, time to save !");

    var input = {
        "gitRepo": repoData[0] + "/" + repoData[1],
        "commitID": sha,
        toolOutputs
    };
    var params = {
        TableName: "analysisResults",
        Item: input
    };
    docClient.put(params, function (err, data) {
        if (err) {
            console.log("DB error - " + JSON.stringify(err, null, 2));
        } else {
            console.log("Saved tool output to DB");
        }
    });
}
save();
```

Figure 22 - Image of sending tool output to DynamoDB

```
function searchResults(repoData, commit) {
    var params = {
        TableName: "analysisResults",
        Key: {
            "gitRepo": repoData[0] + "/" + repoData[1],
            "commitID": commit
        }
    };
    return new Promise((resolve, reject) => {
        docClient.get(params, function(err, data) {
            if (err) {
                console.error("Unable to read item. Error JSON:", JSON.stringify(err, null, 2));
                reject();
            }
            else {
                resolve(data);
            }
        });
    });
}
```

Figure 23 - Image of receiving tool output from DynamoDB

Once this was done, it was straightforward to add code that checks if the repository and commit ID combination is already in the database, and to just pull the results from there instead of running the tools again. However, this does not cover allowing a specific URL to always retrieve cached results for the email notification feature. For this, a second table was created with an ID as the primary key. Whenever a user submits a repository for review, an ID is automatically generated for this purpose. Then, an entry is added to this table which includes the GitHub repo url, tools chosen, and commit ID. The user is redirected to a new URL which includes this generated ID (results?id=-4wqEJqM_pe4X2fjyKIXM). Finally, by ensuring that the first step of the results page is to pull this ID from the second table and gathering the repo results from the first table, this URL can be used in emails to provide a static link which always displays the results without running anything new.

Email Notifications:

Amazon Simple Email Service API was used to send emails to the users. First, an email address for Rust Cleaners was created using Amazon Workmail: rust_cleaners@rustcleaner.net. The API requires a source and recipient email address. The recipient email address was stored by an input box that requested the user to submit their email if they would like an emailed report. The team had to send a request to AWS to verify the rust_cleaners@rustcleaner.net email. This would allow the code to send emails to anyone who entered their email instead of the default of only two recipient email addresses.

```
sendEmail(recipient,link) {
    if(recipient === ''){
        console.log('empty Email');
    }else{
    console.log("Inside function");
    let params = {
      Source: 'rust_cleaners@rustcleaner.net',
      Destination: {
        ToAddresses: [
          recipient
        ],
      },
      ReplyToAddresses: [],
      Message: {
        Body: {
          Html: {
            Charset: 'UTF-8',
            Data: link
          },
        },
        Subject: {
          Charset: 'UTF-8',
          Data: 'Hi, This is report!',
        }
      },
    };
   return AWS_SES.sendEmail(params).promise();


    }
};
```

Figure 24 - Image of emailing function

Hosting the Website:

Our application was deployed using the AWS Elastic Beanstalk command line interface. As the application should already be containerized using the Docker image discussed earlier, hosting it is pretty easy. The instructions can be found here. First, the project directory should be turned into an Elastic Beanstalk project by using the initialize command. Then, by running create the project can be hosted on an EC2 instance which is accessible through an Elastic Beanstalk URL. Our project required a t2.large EC2 instance because running the tools can be quite demanding, and the Docker image was too large to fit on some of the smaller versions.

Once the EC2 instance is up and running, requests can be made to the server just like it was on localhost. To debug issues and view the logs the instance can be accessed by SSH and by running commands such as docker ps and docker exec, any errors or prints can be seen.

When everything is functional on AWS, the last step is to connect it to the domain which was purchased through AWS Route 53. First, an elastic IP address was allocated from the AWS Management Console. This was assigned to the EC2 instance. The domain name of the project was rustcleaner.net. The domain name was registered through Route 53 by creating a hosted zone. Once the domain was bought, Domain Name Service needs to be set up. This was also done by Route 53 by creating a hosted zone and creating a record for the domain.

5 Experimental results

5.1 Description of tests performed

In total, we performed 6 tests on our product. Three tests were performed on the main functionality of the product, that is reviewing newly committed code and returning easy to read results to the user. With these tests we intended to test both the quantity of warnings returned and the run time of the program. The other three tests were performed to test the extra features of our product. These features are not used everytime the product is used and seek to augment the main functionality of the product discussed earlier.

5.1.1 Basic Repo Test

The intention of this test is to demonstrate the basic functionality of the project. For this test, we selected https://github.com/ezekiiel/simple-solana-program as our test repository. This repository was selected because the product was originally designed to audit Solana programs and due to the repo's relatively small size. In this test, we will submit the Repository to the product and measure how many minutes the repository takes to run. Additionally, we will record how many warnings the product returns to the user and their severity.

5.1.2 Large Repo

The aim of this test is to determine how well the product handles very large repositories. For this test, we selected https://github.com/atom-archive/xray as our test repository. This repository was selected for its immense size. The repository includes over 100 individual rust files split among 7 subfolders. As with the last test, we will submit the Repository to the product and measure how many minutes the repository takes to run. Additionally, we will record how many warnings the product returns to the user and their severity. This test is expected to return enough warnings that a simpler UI would be difficult for a user to navigate and understand the results.

5.1.3 Repo Without warnings

This test is designed to determine how the product handles a repository that has no errors. For this test, we selected https://github.com/unrelentingtech/freepass as our test repository. This repository was selected due to its small size and lack of warnings returned. We will submit the Repository to the product and measure how many minutes the repository takes to run. Additionally, we will record how the frontend reacts to the lack of warnings. This test is expected to show how the product handles null values without causing exception.

5.1.4 Report Recall

The intention of this test is to demonstrate the augmented functionality of the report recall feature. For this test, we selected https://github.com/ezekiiel/simple-solana-program as our test repository. The only difference between this test and the first test is that the first test has already occurred and its results have been saved to the database. In this test, we will submit the Repository to the product and measure how much less time the database recall takes relative to the first test. This is expected to take seconds rather than minutes. Additionally, we will record how the warnings compare to the first test, it is expected that they will be identical.

5.1.5 Auto-Email

The aim of this test is to demonstrate the augmented functionality of the auto-email feature. This test will be run at the same time as the first test. The only addition to the first test is that the tester will give an email while the github repository is processing. The tester is expected to receive an email, once the product finishes reviewing the code, that includes a link to the finished result . Opening these results is expected to take seconds rather than minutes. Additionally, we will record how the warnings shown in the new tab compare to the first test, it is expected that they will be identical.

5.1.6 File Tree and File Warning Highlighter

This test can be applied to any other test as it evaluates the file tree and file warning highlighter function available on every result. The intention of this feature is to allow users to scroll through their code and have the warnings shown in the previous summary highlighted for easier debugging on their local files. We selected https://github.com/ezekiiel/simple-solana-program once again for its small size and simple nature. The test will consist of verifying that every warning found on the summary page can be located through the file warning highlighter function as well.

5.2 Testing Results

This section gives the numerical and photographic results of each test described in the previous section.

5.2.1 Basic Repo Test

This test took 10 minutes to complete
This test returned a total of 8 Clippy warnings and 5 uses of deprecated packages

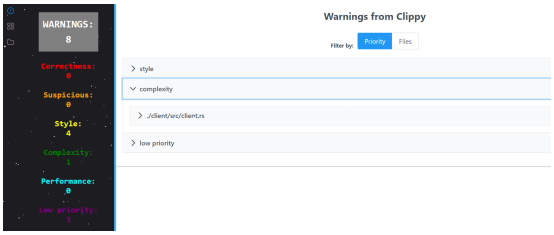

Figure 25 - Returned Clippy Results

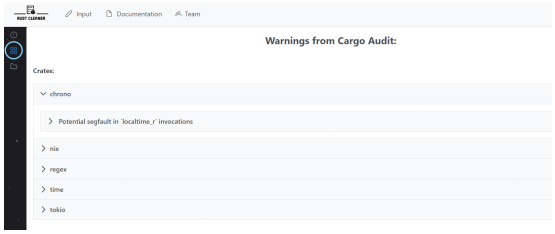

Figure 26 - Returned Audit Results

5.2.2 Large Repo

This test took 18 minutes to complete
This test returned a total of 235 Clippy warnings and 16 uses of deprecated packages

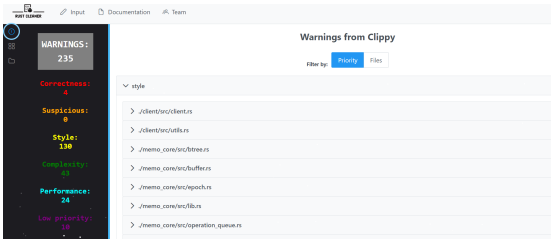

Figure 27 - Returned Clippy Results

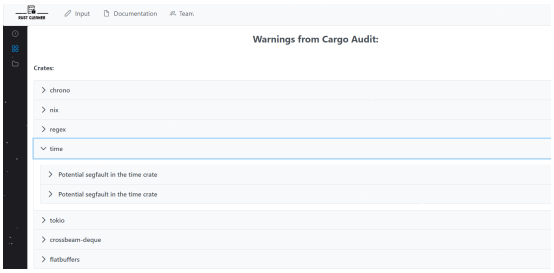

Figure 28 - Returned Audit Results

5.2.3 Repo Without warnings

This test took 6 minutes to complete
This test returned a total of 0 Clippy warnings and 8 uses of deprecated packages

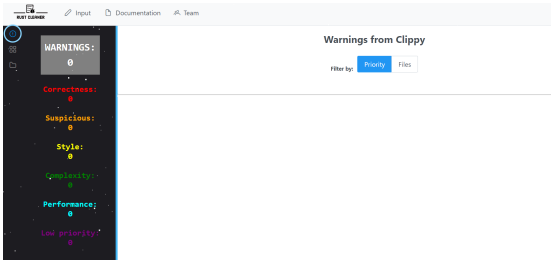

Figure 29 - Returned Clippy Results (empty)

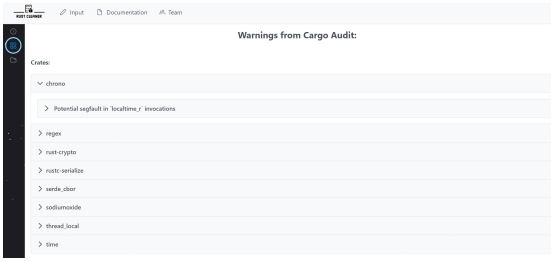

Figure 30 - Returned Audit Results

5.2.4 Report Recall

This test took approximately 10 seconds from submitting the github repo to displaying the results
Results were identical to those found when the repository was run the first time

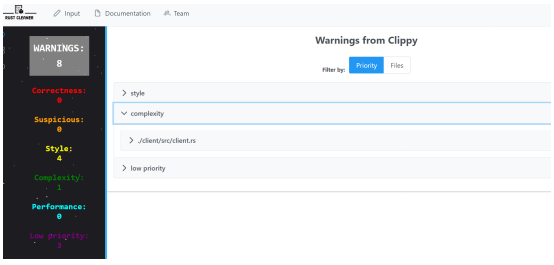

Figure 31 - Returned Clippy Results

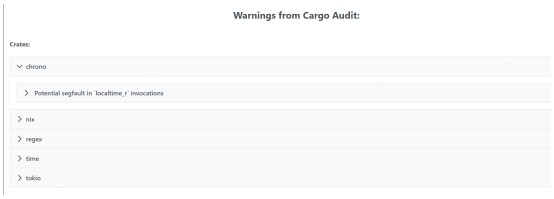

Figure 32 - Returned Audit Results

5.2.5 Auto-Email

This test took approximately 10 seconds from the link opening to the results being displayed. Results were identical to those found when the repository was run the first time

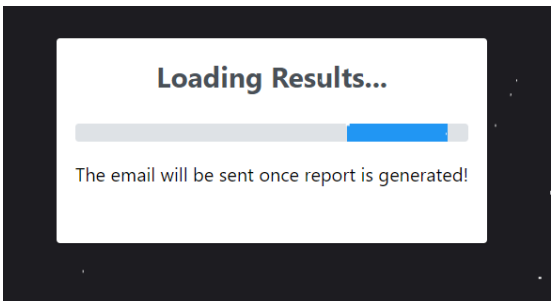

Figure 33 - Email submission

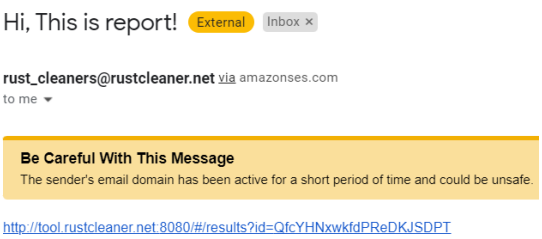

Figure 34 - Email received

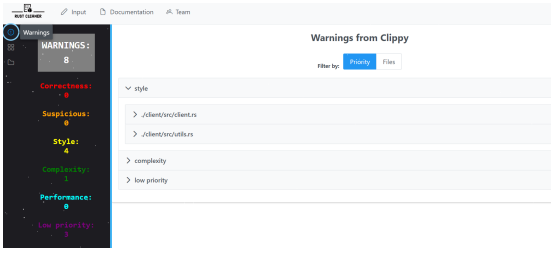

Figure 35 - Returned Results

5.2.6 File warning highlighter

All 8 warnings found in the Clippy summary could be located in the file tree system.
On click warning descriptions matched those in the warning summary page.
Files loaded in under 5 seconds each.

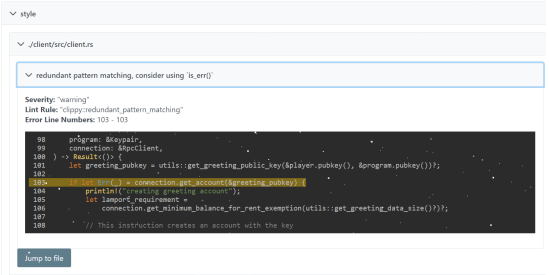

Figure 36 - Clippy Warning

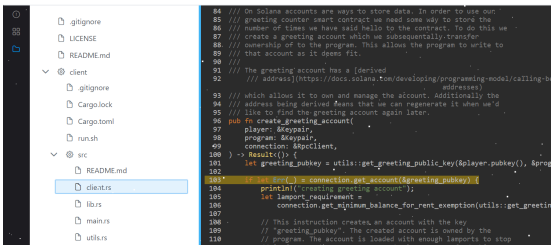

Figure 37 - Warning Found in Highlighter

5.3 Analysis of results

5.3.1 Basic Repo Test

The results of this test show that the basic functions of our product work, and in a very reasonable amount of time. 10 minutes is more time than most developers would like to wait, but this is solved by allowing the user to receive an email when the results are done processing. Once processed, the results are easy to see with a clear severity hierarchy and descriptions of where the warnings occur in each file. Additionally, the warnings for deprecated crates can be seen in the next screen, which will remove duplicate warnings and recommend an action to solve each unsafe crate.

5.3.2 Large Repo

Although the repo was larger, it only took 8 more minutes to complete. As shown in 5.2.2, there were 235 Clippy warnings. If the user attempted to run Clippy locally instead of using our service, they would have to scroll through and interpet hundreds of lines of .json code or terminal output. Instead, our product immediately shows the user what kind of warnings were caught. as well as highlights the area to reduce finding where the warnings occur. Additionally, even though there are many different cargo files, only 16 cargo warnings were shown, this is because duplicate warnings are removed from the output and each solution is only given once. This makes the results much more understandable than that of the command line output.

5.3.3 Repo Without warnings

Despite having no warnings from clippy, the dashboard can still be seen. This proves that the product can handle the case when no issues were detected and as a result much of the variables became null (as seen in the number of warnings found). Even though nothing was found with Clippy, Cargo Audit found deprecated packages. This further emphasizes how the product's results are independent from each other, which is expected.

5.3.4 Report Recall

Because the information was already stored in the database, the results took 10 seconds to load. This is 60 times faster than the original loading time for the repo in 5.2.1. In addition, the results are identical to 5.2.1, which was expected. This validates that the database works as intended, and speeds up loading time after the initial run of that commit ID.

5.3.5 Auto-Email

The Auto-Email function was created to solve the most apparent issue with the product, long processing times. While this addition does not increase speed, it allows the user to do more than wait for their report to appear. This test shows that this feature can be trusted to return identical results to what the user would see if they had waited the entire time. The results given on both clippy, cargo audit, and the file highlighter are identical. This was expected because the email pulls the exact results the first run produced and serves it to the new user.

5.3.6 File warning highlighter

The test shows that both the filetree subsystem and the file highlighting subsystems work. The filetree is easily navigable as it uses the same structure as the user's own repository, and files are loaded within 5 seconds of request. This is a noticeable delay, but it is far from disruptive. Next, the highlighting system works as expected, every warning in the summary can be found by scrolling through the code. On click, the highlighted sections display their warning text as an alert to the user's browser.

6 User's Manuals

The Rust Cleaners service doesn't require any hardware or software installation to use the service. The user is just required to submit their code and since there is a Docker container that contains all the dependencies for the tools, there are no software installations required to use the tools. The Rust Cleaners website contains a documentation page that lists all the steps required to use the tool. A potential can easily follow the steps shown on this page. In addition, this documentation page contains details on the various static analysis tools. The details of the tools include a description of each tool and the tool's Github link, so an user can obtain additional information if they desire.

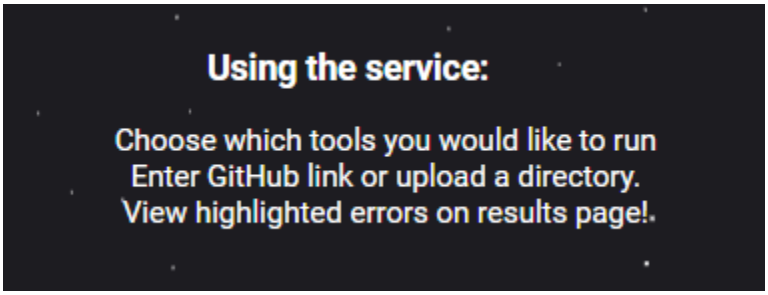

Figure 38 - Image of steps on Documentation page

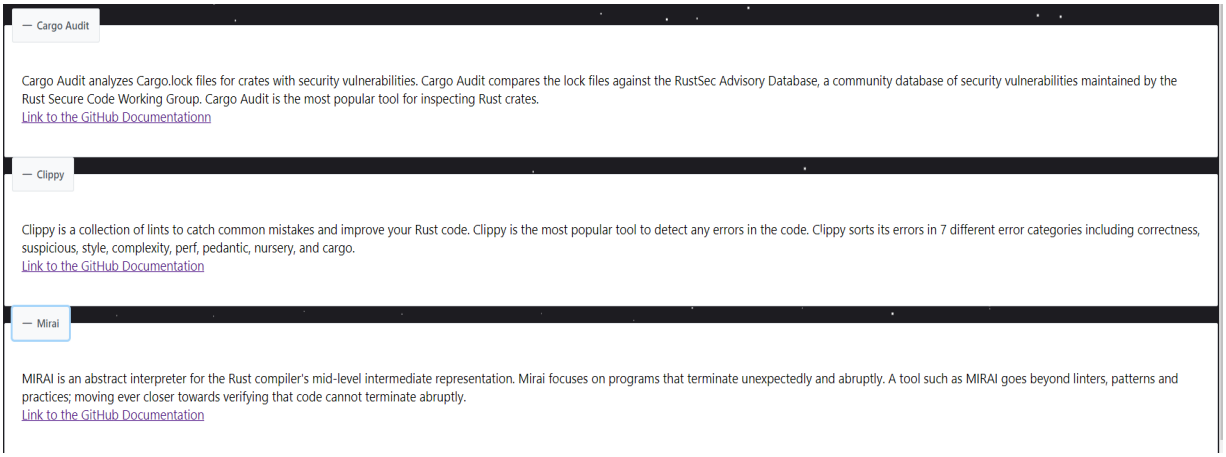

Figure 39 - Picture of details of the available static analysis tools on Documentation page

7 Course debriefing

The team management style of the team is more of a loose, collaborative style. Each week, the team members of Rust Cleaner would discuss their availability on the first day of class. Based on everyone's availability, different tasks were distributed to the members. Usually, members would be given tasks that were associated with what they were currently working on. All of the tasks for each week were written down on a Google Docs page. The delegated tasks were also written down on this page. This team management style worked well for the team. Since all the team members have other projects and classwork to work on, it was important to take in everyone's availability into account. However, there could have been better communication between all the team members. For example, a lot of the code didn't have comments, so it became difficult to understand code written by other team members.

The team has bi-weekly meetings in class in order to address any concerns someone has with their current tasks or to get input from other team members. The team has been using GroupMe to keep in constant contact with each other, as well as sending out updates whenever an important change is pushed or someone believes that something in the code should be changed. This keeps everyone on the team on the same page and prevents team members from working on the same tasks or potentially having contradicting code implementations. Discord was also used for voice chats, screen sharing, and sending helpful links. By messaging each other about what big changes have been made, the team is essentially doing component testing. If the next person who works on it does not see the result that they have shown in our chat, then clearly something was done incorrectly.

The team has continued to use Gantt charts to keep track of who is working on what and what the current progress for each task is. This helped the team to stay on schedule during the production cycle. It also helped with displaying if the team was behind schedule which happened a handful of times. When behind schedule, the team members would discuss with the professor and the TA why the team was behind and how to prevent it from happening again.

An ethical concern that was present was ensuring that there was no negative environmental impact due to the Rust Cleaner service. In terms of environmental impact, Solana is much more environmentally friendly than other traditional crypto that involves mining. However, because the team used cloud servers for the static analysis tools and the website, there will be CO_2 emissions higher than using physical servers. The project minimized the environmental impacts by not using any hardware components. Another ethical concern was user privacy. The database stores all the code uploads which can potentially lead to compromised code. This was prevented by using Amazon DynamoDB which encrypted the code, so the potential hackers can't access the code without the teams' credentials.

The service was tested using different GitHub repositories to ensure that it works for different types of code repositories Since the service worked for the different repositories of different sizes and functionality, it showed the team that the service worked accurately and without error. There could be potential edge cases that haven't been covered by the team. Overall the project works as intended with some feature changes due to feasibility or time constraints. If we got to do this project again, we would incorporate testing with public repos from within organizations the team was or wasn't a part of. In addition to this, as the UI was getting created, and possibly during the time of including a line highlighter, we would start user testing to get input on designs and how a user might interact with the product. If there was more time, there would have been user testing to see if these potential edge cases were in fact covered.

8 Budgets

There were two costs for this project. The first cost was a 14 dollar cost for the domain. The domain that we selected and paid for was rustcleaner.net. This expense matched up to the original proposed budget which stated 5 dollars for the domain. The Rust Cleaner team decided to spend a little more than the proposed budget to obtain a better domain name.

The second cost was a 30 dollar charge for AWS hosting through Elastic Beanstalk. Elastic Beanstalk sets up an AWS EC2 instance for the project. This cost was much less than the original proposed budget which stated 43.24 dollars per month for EC2 servers. The expense was smaller because the Rust Cleaner team chose not to use multiple EC2 servers and chose to use a singular EC2 instance.

In order to keep this service running in the future, an EC2 large instance will be required. This instance will cost 35 dollars a month. The more the tools our service adds, the higher the tiers the service would have to use. Additionally, with more use, this cost may be multiplied. In order to serve a greater number of concurrent users, more instances would need to be provisioned to run in parallel.

9 Appendices

Here you can include any additional information not already in the implementation notes (e.g., detailed circuit schematics, class hierarchy) that would help replicate or maintain your implementation.