

# Section 1: About You

## Goals and Objectives

<b>Why do you want to do a GSoC project with Oppia?</b>	<p>In 2023, when I was introduced to GSoC for the first time, my initial choice of organization was Oppia-Android. However, initiating this <a href="#">discussion</a> I found that Oppia-Android wouldn't participate in GSoC'23, I pursued <a href="#">GSoC'23 with Catrobat</a> instead. Now, in 2024, I aim to fulfill that aspiration. Firstly, the aspects of Oppia that appealed to me were:</p> <p><b>Engaging Learning Experience:</b> I am passionate about integrating educational content with enjoyable elements such as cartoon illustrations. In my own educational endeavors, I frequently utilize techniques like incorporating drawings, narratives, and memes to explain concepts interactively and effectively. Oppia's mission to spark curiosity in learning resonates deeply with me.</p> <p><b>Community Support:</b> Having been involved with various organizations, I have found Oppia's community support to be exceptionally engaging. Despite busy schedules, I found prompt and helpful guidance from the welfare team and communications with PRs and the Android team from day one. This level of support has been invaluable in my experience.</p> <p><b>Skill Alignment:</b> Oppia not only aligns with my existing skills but also presents an opportunity to delve deeper into the codebase, thus enhancing my technical expertise. It offers a perfect balance of familiarity and the chance to further develop my abilities.</p>	
<b>What do you hope to learn/achieve during GSoC?</b>	<p>I hope to learn and achieve a great deal during GSoC, particularly in the context of the project I'm aiming to contribute to. While I have experience with Gradle, test writing, and developing end-user applications in my GSoC'23 journey, I'm eager to delve into more complex developer-oriented projects and technologies such as JaCoCo, Bazel, CI/CD workflows, and automation scripting. This project presents a valuable learning opportunity for me to expand my skill in real-world applications.</p> <p>Furthermore, from mid-January leading up to the application period itself, I've dedicated substantial time to exploring Bazel and JaCoCo. This endeavor has been immensely rewarding, as it has provided me with valuable insights into the importance of implementing code coverage and how it contributes to expediting development processes while ensuring the enabling of automated releases with minimal bugs.</p>	
<b>Which Oppia teams have you collaborated with, and what have you</b>	Developer Workflow & Infrastructure - Android	PR: <a href="#">#5324</a> <b>Updated the Wiki</b> with instructions for successfully setting up and <b>building Bazel on Windows</b> . The previous instructions lacked important details

done on those teams?		regarding: <ol style="list-style-type: none"> <li>1. The necessity of installing and alternating between two versions of Java for running sdkmanager and bazel build.</li> <li>2. Updating Windows file paths.</li> <li>3. Updating the Android platform versions.</li> </ol>
	Welfare team	<p><b>Discussions 1:</b> <a href="#">#5241</a> Aided in resolving the Bazel installation error on Windows.</p> <p><b>Discussions 2:</b> <a href="#">#5045</a> Assisted in resolving the Gradle build error regarding unmappable character encoding</p> <p><b>Discussions 3:</b> <a href="#">#4830</a> Assisted in successfully running the Robolectric tests on Android Studio Bumblebee (Patch 3)</p> <p><b>Discussions 4:</b> <a href="#">#19053</a> Assisted with the onboarding process for the Oppia community.</p> <p>I've occasionally participated in the weekly team meetings.</p>
Contact information	<p><b>Name:</b> Ramadevi  <b>Alias / Preferred Name:</b> RD  <b>Email:</b> <a href="mailto:rd4ramadevi@gmail.com">rd4ramadevi@gmail.com</a>  <b>LinkedIn:</b> <a href="https://www.linkedin.com/in/rd4dev/">https://www.linkedin.com/in/rd4dev/</a></p>	
Preferred method of communication	<p>Email          Google Chat - (part of the Oppia welfare team, leveraging this platform)          Slack</p>	
Which timezone will you primarily be in during the summer?	<p>Indian Standard Time (IST) - GMT+5:30</p>	
(If you are a student) When are your school holidays?	<p>I am currently pursuing an MBA through distance education, where classes are scheduled only on Saturdays and Sundays. Consequently, even during the semester, I have ample time available for work.</p> <p>However, this semester's classes concluded early in March, and the next semester is scheduled to commence only after July or August due to the 2024 political elections. As a result, I will have a significant amount of free time from April until nearly August.</p>	

<b>What other obligations might you need to work around during the summer?</b>	Exams for the distance course may be scheduled, although I haven't received any updates on them yet. However, if exams do take place, I anticipate having around ten days, from morning to afternoon, reserved for exams, leaving my evenings with over 3 hours available for work everyday.
<b>Planned time commitment</b>	<p>I have planned to allocate approximately <b>4 to 5 hours each weekday</b> from May to August, totaling nearly <b>375 hours</b>, and I'm also willing to dedicate time on weekends if necessary.</p> <p>I find it beneficial to stay connected with the codebase even on weekends, as I have a genuine passion for coding. Moreover, it helps me avoid feeling disconnected or struggling to reacquaint myself with the codebase after a two-day gap, which is a common occurrence in coding.</p> <p>In my prior project with Catrobat before GSoC'23, which was of substantial size spanning 350 hours, I dedicated 4 to 5 hours daily. This commitment allowed me to finish the project within 168 hours.</p>

## Section 2: About Your Project

### Project Details

<b>Project title</b> <i>(should match the one on the Project Ideas list)</i>	4.1. Code coverage support and enforcement
<b>Project size</b> <i>(should match the options on the Project Ideas list)</i>	Medium (~175 hours)
<b>Why did you choose this project?</b>	<p>I chose this project for two main reasons. Firstly, my prior experience with writing full-stack projects and tests has prepared me for the challenges inherent in this project. I believe tackling a project of this complexity will significantly enhance my coding skills and improve my workflow efficiency. Over the past month, as I've delved deeper into this project, I've gained a better understanding of its various workflows and inner workings, solidifying my enthusiasm for the project as a whole.</p> <p>Secondly, while I was also interested in other projects, namely 4.2 Multiple Classroom (<a href="#">initial prototype</a> I worked on) and 4.3 Platform Features, I've recognized that 4.3 is no longer part of GSoC, and 4.2 is drawing considerable attention. Consequently, focusing on project 4.1 appeared to be more targeted and worthwhile, offering ample opportunities to explore.</p>

## Required Skills

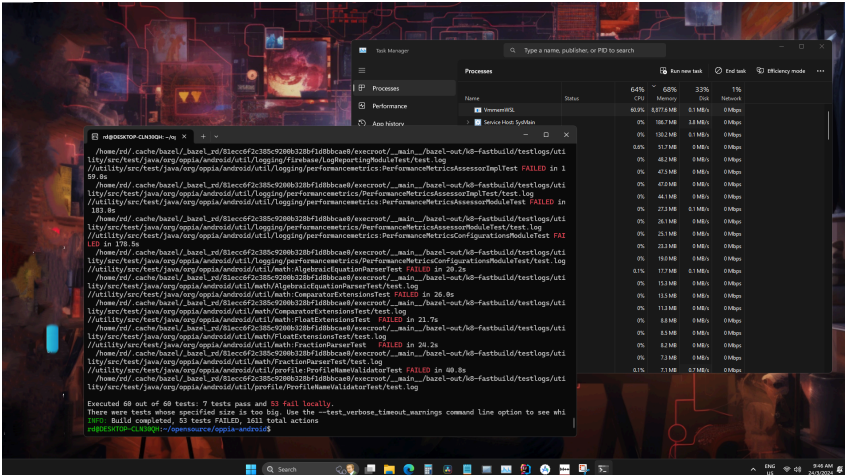
WEB	
I can write Python code with unit tests.	N/A
I can write TS + Angular code with unit tests.	N/A
I have good UI judgment and attention to detail.	N/A
I can debug and fix CI failures / flakes.	N/A
I can write or modify e2e or acceptance tests.	N/A
I can communicate effectively using <a href="#">debugging docs</a> .	N/A
I can write or modify <a href="#">Beam jobs</a> .	N/A
I have participated in QA testing.	N/A
ANDROID	
I can write code and tests in Kotlin.	<ol style="list-style-type: none"> <li>PR <a href="#">#5350</a> - Handling Removal of the CircularImageView dependency from both Gradle and Bazel</li> <li>PR <a href="#">#5196</a> - Fixed lint warnings in "Usability:Typography", "Usability:Icons", and "Usability" categories</li> <li>PR <a href="#">#5218</a> - Fixed Inconsistent Layout Lint Warning</li> </ol> <p>Though specified to submit only merged PRs from Oppia Android, I haven't had the chance to work with Kotlin code with the above issues. Hence, I'd like to include these two PRs for reference as well.</p> <p>Local Story Text Size Branch <a href="#">[code]</a> : I started working on #1491 without noticing the open PR <a href="#">#5290</a>, but I guess the local work could also serve as my exploration and work with the Kotlin codebase</p> <p>Paintroid PR <a href="#">#1288</a> - This is one of my GSoC PRs from last year, which involved Kotlin code for both the landing page frontend and backend, along with corresponding tests.</p>
I can build the app and run	<b>Bazel build success:</b>

## tests using Bazel.

[illegible]

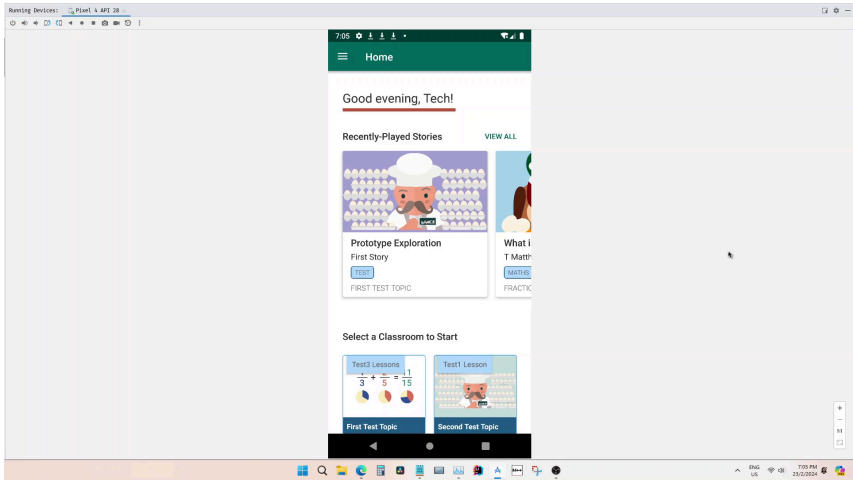
## Bazel tests run:

As mentioned in the Discussions [#5371](#), when running Bazel tests with version 4.0.0, a considerable number of test cases fail.



However, after configuring the environment for Bazel 6.2.0 and executing the tests, all 157 tests pass successfully in the local environment.



	While I'm also working on resolving other test runs, progress has been made with upgrading Dagger and a few other dependencies.
I can make changes to UIs in Android, write tests for them, and manually verify that they work.	PR <a href="#">#5196</a> - Incorporates alterations to <a href="#">text sizes</a> , <a href="#">image densities</a> , and <a href="#">monochromatic adaptive icons</a> .
I can make changes that involve DataProviders (either by creating them, modifying them, or updating code that depends on them).	Local Story Text Size Branch <a href="#">[code]</a> : This code observes changes in profile data using data providers and adjusts font scaling based on alterations in reading text size.
I can make changes to proto files and/or work with generated proto code.	Multiple Class Room Prototype Branch <a href="#">[code]</a> : During my work on the 4.2 prototype, I delved into the topic of proto files to incorporate the classroom chip name. 
I can make changes to production code and/or tests that involve platform parameters.	N/A
I can add new dependencies to the app's Dagger graph and I understand how they are used in tests.	PR <a href="#">#5350</a> - Handled Removal of the CircularProgressIndicator and CircleImage dependency from both Gradle and Bazel

## Project Timeframe

**Note:** Oppia will only be offering a single GSoC coding period timeframe this year, starting on **May 27**. All work for Milestone 1 must be completed and submitted by **Jun 28** for internal feedback (with any revisions due by **Jul 8**), and all work for Milestone 2 must be completed and submitted by **Aug 12** for internal feedback (with any revisions due by **Aug 19**). We will not be able to extend these deadlines.

Coding period	<ul style="list-style-type: none"> <li>I will adhere to the above deadlines.</li> </ul>
---------------	---

## Communication Channels

I can commit to sending daily updates to my mentor by email, each day I work during the GSoC period.	<ul style="list-style-type: none"> <li>Yes</li> </ul>
In addition to the above: how often, and through which channel(s), do you plan on communicating with your mentor?	<p>I would prefer to stay connected <b>at least once</b>, perhaps towards the <b>end of the day</b>, to share the progress of my work with my mentor.</p> <p>I have been communicating with the welfare team and Android team via <b>Google Chat</b>, and I am open to using any medium that mentors prefer. Previously, I have used Slack and email for communications, so those options would also be preferred.</p> <p><b>Yes</b>, as I believe communication would not only help me resolve unexpected issues but also help me stay accountable.</p>

## Section 3: Proposal DetailsProblem Statement

Target Audience	<b>Developers</b> (primarily focusing on code quality, testing procedures, release automation processes)
Core User Need	The core user need for this project is to ensure higher confidence in technical changes made to the Android codebase, and aiding in automating releases more effectively, allowing the team to deliver new features to users faster and with fewer bugs.
What goals do we want the solution to achieve?	The goal of the solution is to implement comprehensive code coverage analysis for Kotlin files in the Android codebase, addressing support gaps and introducing per-unit coverage measurement with configurable percentage enforcement thereby improving development confidence and code quality.

### Section 3.1: WHAT

#### Key User Stories and Tasks that will be implemented

#	Title	User Story Description (role, goal, motivation) <i>"As a ..., I need ..., so</i>	List of tasks needed to achieve the goal (this is the "User Journey")	Links to mocks / prototypes, and/or PRD sections that
---	-------	--	---	---



		<i>that ...."</i>		<b>spec out additional requirements.</b>
1	Implement Target Specific Code Coverage Analysis	As a developer, I need to run code coverage for a specific Bazel target so I can accurately assess code coverage ensuring comprehensive testing.	Execute Code Coverage Script	
			Generate Rich Report	
			Review Code Coverage Analysis	
2	Automate Code Coverage in CI Workflow	As a developer, I need to run automated code coverage in our CI workflow after unit test success so that comprehensive testing is ensured during PR development.	Verify that all unit tests pass successfully.	
			Trigger code coverage analysis using the new script.	
			Generate and upload code coverage reports.	

## Technical Requirements

### Additions/Changes to Web Server Endpoint Contracts

#	Endpoint URL	Request type (GET, POST, etc.)	New / Existing	Description of the request/response contract (and, if applicable, how it's different from the previous one)
1.	N/A	N/A	N/A	N/A

### Calls to Web Server Endpoints

#	Endpoint URL	Request type (GET, POST, etc.)	Description of why the new call is needed, or why the changes to an existing call is needed
1.	N/A	N/A	N/A

### UI Screens/Components

#	ID	Description of new UI component	i18n required ?	Mock/spec links	A11y requirements
1.	N/A	N/A	N/A	N/A	N/A

## Data Handling and Privacy

#	Type of data	Description	Why do we need to store this data?	Anonymized?	Can the user opt out?	Wipeout policy	Takeout policy
1.	N/A	N/A	N/A	N/A	N/A	N/A	N/A

## Other Requirements

### Will the feature require the use of particular third-party libraries or technologies?

It will rely on existing third-party dependencies such as Bazel build rules, which are already integrated into the project's source code. Listed are the major requirements:

Technology/Tool	Purpose/Description
Bazel 6.2.0	Adds test coverage
Coroutines	For asynchronous code
CI/CD Pipelines	Automated code coverage analysis and reporting
GitHub Actions	Third party GitHub Actions <ul style="list-style-type: none"><li>• Upload Comment by <a href="#">Peter Evans</a></li><li>• Workflow Celler by <a href="#">Style</a></li></ul>
Proto	Represents coverage data
JUnit	Runs tests
Build rules	For compiling code and tests <ul style="list-style-type: none"><li>• rules_java</li><li>• rules_kotlin with <a href="#">bazel#17467</a> commit</li></ul>

---

## Section 3.2: HOW

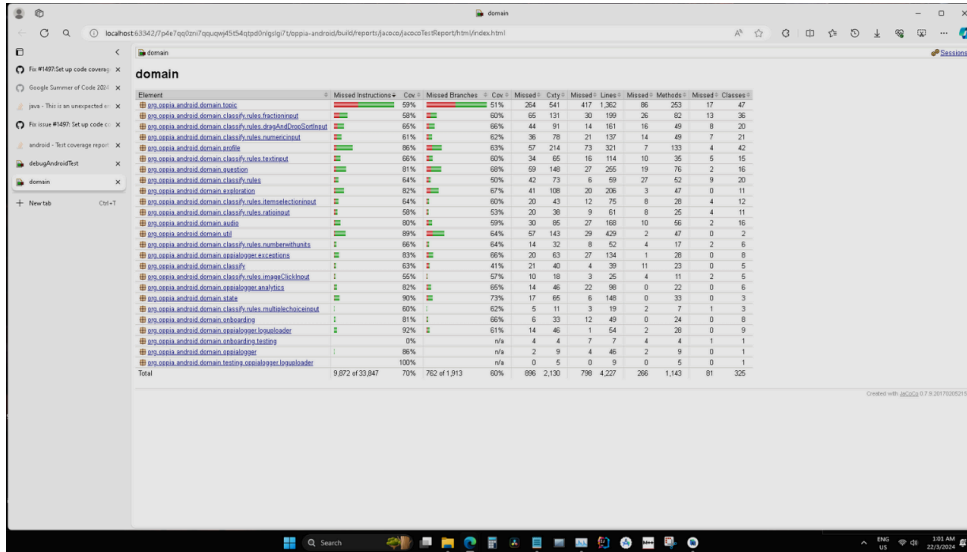
### Existing Status Quo

#### 1. Past Attempts:

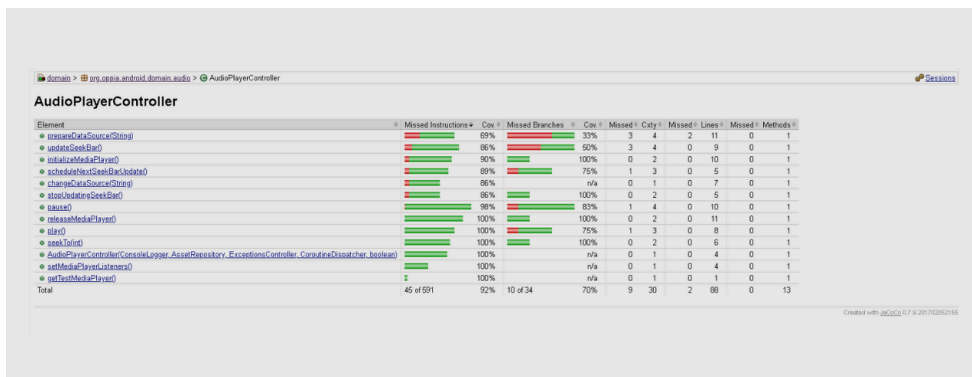
- Gradle Code Coverage: [Previous attempts](#) to incorporate code coverage using Gradle and JaCoCo (version 0.7.9) were unsuccessful.
  - JaCoCo encountered issues in multi-module Gradle projects (refer to <https://github.com/jacoco/jacoco/issues/996>).
  - Observations from pull requests, as of the PR date, reveal discrepancies in code coverage reporting:

- JaCoCo 0.8.0 reported 0% coverage for certain modules, while 0.7.9 managed to provide coverage for the utility module but not for the app.
- The integration between AGP (Android Gradle Plugin) and JaCoCo is intricate, complicating the understanding of why certain configurations yield different results.
  - Considering the team's endeavor to transition to Bazel for code coverage, it appears beneficial in the long term, to align with the broader migration strategy.
- Report: My attempted report generated using JaCoCo 0.7.9 and Gradle is attached below:

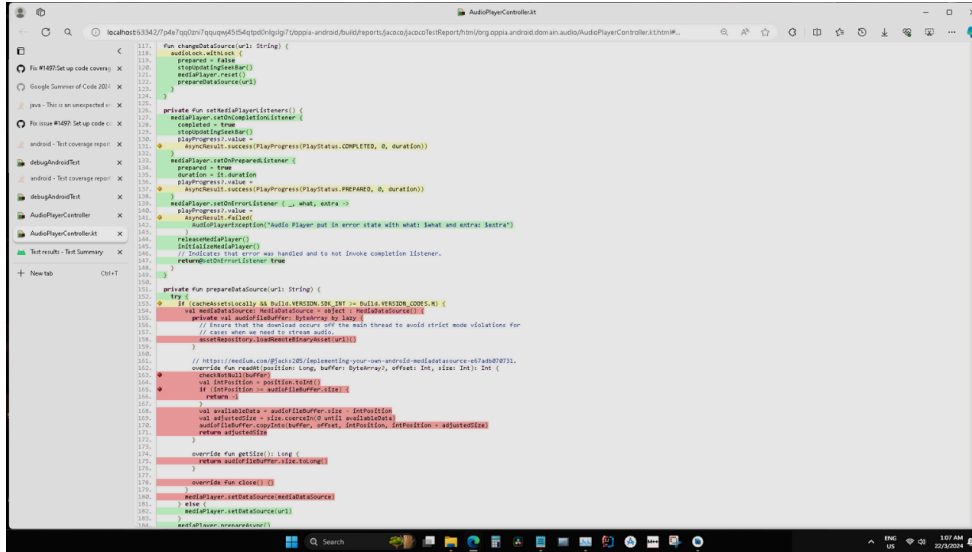
## Domain Module:



## AudioPlayerController:



## File Report:



## Existing Limitations:

These limitations are outlined by the JaCoCo team in their [official FAQ section](#) and in the GitHub issues within the [JaCoCo repository](#).

Library/Tech nology Affected	(Brief) Description	Details	Mitigatable in Oppia (Y/N)	Tracking Issues (in Oppia)
JaCoCo	Source code lines with exceptions show no coverage	JaCoCo does not mark lines covered if an exception occurs, affecting accurate coverage reporting.	No  Stems from JaCoCo's design and Oppia cannot control it.	N/A
JaCoCo	Line coverage figures not shown in the coverage report	JaCoCo relies on class files analysis and requires debug information to calculate line coverage, leading to missing line coverage figures in reports if code is not compiled with debug information.	No  Debug info provision isn't always feasible, affecting JaCoCo accuracy.	N/A
JaCoCo	Reflection usage may cause failures	JaCoCo instruments classes under test, adding synthetic members. Reflection usage may fail if not adjusted to ignore synthetic members, impacting coverage data.	No  Handling synthetic members in reflection is complex and error-prone.	N/A

JaCoCo	Error while instrumenting certain Java classes	JaCoCo can instrument only valid class files, resulting in errors for files with syntactic or semantic errors.	No  Depends on JaCoCo's ability to handle class files, which we have limited control over	N/A
JaCoCo	StackOverflowErr or during code coverage analysis	Misconfiguration or heavy stack usage can cause this error, necessitating correct configurations and potential stack size adjustments.	Yes  Address misconfigurations and increase stack size to prevent StackOverflowError	
JaCoCo	Abstract methods not shown in coverage reports	Abstract and interface methods lack executable code, making direct coverage evaluation impossible, but subclasses implementing them are covered.	No  Inherent to language and JaCoCo design; altering requires fundamental changes	N/A
JaCoCo	Excluded classes may still appear in the coverage report	Classes appear in the coverage report despite exclusion in JaCoCo due to separate report generation where all relevant class files must be explicitly provided.	Yes  JaCoCo advises configuring the relevant report generation tool accordingly.	
JaCoCo (Kotlin Context)	Kotlin inline functions are not reported in code coverage	JaCoCo does not recognize coverage for Kotlin inline functions due to challenges in mapping inline function calls to source code lines. This leads to incomplete coverage analysis, potentially affecting code quality assessment and test effectiveness.	No  JaCoCo's inability to recognize coverage for Kotlin inline functions due to mapping challenges is inherent to its design.  Some propose writing Java unit tests to call Kotlin inline functions, but they also fail with reified types.	In JaCoCo - <a href="#">Issue #654</a> , <a href="#">Issue #973</a>
JaCoCo (Kotlin	Code Coverage Issue with Kotlin	JaCoCo incorrectly marks certain Kotlin suspend	No	In JaCoCo - <a href="#">Issue #1410</a>

Context)	Suspend Functions	functions, particularly those calling WebClient.awaitBody(), as not fully covered by tests. This issue affects code coverage accuracy, particularly when using suspend inline functions with reified type parameters.	These limitations are specific to JaCoCo's handling of Kotlin language features and constructs and require changes at the library level.	
JaCoCo (Kotlin Context)	JaCoCo Coverage Issue with Automatically Generated Parcelable Creator	JaCoCo fails to recognize and calculate coverage for the automatically generated Parcelable Creator class and its methods (createFromParcel and newArray) in Android projects. This discrepancy affects code coverage reporting accuracy, as these methods are not accessed despite being essential for Parcelable implementation.		In JaCoCo - <a href="#">Issue #1364</a>
JaCoCo (Kotlin Context)	Missing branch coverage while using Sealed Classes	JaCoCo incorrectly reports missing branch coverage when using sealed classes within a when expression in Kotlin. Despite the exhaustive nature of sealed classes, JaCoCo identifies a missing branch in the coverage report. This issue affects code coverage accuracy in Android projects utilizing sealed classes.		In JaCoCo - <a href="#">Issue #1219</a>
JaCoCo (Kotlin Context)	JaCoCo Reports Missing Branch for Nested Elvis Operator in Kotlin	JaCoCo incorrectly identifies a missing branch for a nested elvis operator (?.) in Kotlin code. Despite both null and non-null inputs being tested, JaCoCo reports only one branch covered out of two. This issue affects code coverage accuracy, particularly when testing methods with nested elvis operators.		In JaCoCo - <a href="#">Issue #921</a>

**Existing Limitations associated with Bazel:**

Library/Tec hnology Affected	(Brief) Description	Details	Mitigatable in Oppia (Y/N)	Tracking Issues (in Oppia)
Bazel (Oppia-andr oid)	Lack of code coverage support	Bazel 4.0.0 lacks code coverage, resulting in a "no local_android_tests" error; rules_kotlin lacks android support until Bazel 6.2.0.	Yes  This can be mitigated by migrations to Bazel 6.2.0 which was already initiated through the <a href="#">PR #4886</a> .	Oppia-androi d - <a href="#">PR #4886</a>
Bazel (JaCoCo)	Discrepancy between source and JAR paths	Limitation arises from how rules_kotlin constructs -paths-for-coverage.txt, leading to inaccuracies in coverage analysis.	No  This arises from how rules_kotlin constructs the coverage paths file, and would require changes to the library itself.	In Bazel - <a href="#">Issue #12159</a>
Bazel (Android)	Lack of ViewBinding support	Absence of ViewBinding support in Bazel may lead to inaccuracies in coverage metrics for ViewBinding-reliant projects.	No  Limitation lies in Bazel's core functionality, needing to rely on a future update to Bazel.	In Bazel - <a href="#">Issue #13931</a>
Bazel (JaCoCo)	Lack of exclusion support	Inability to configure exclusions for code coverage analysis hampers granularity and accuracy of coverage reports.	No  Fix within the library would be necessary to allow users to define exclusions for a more robust and maintainable approach.	In rules_kotlin - <a href="#">Issue #822</a>

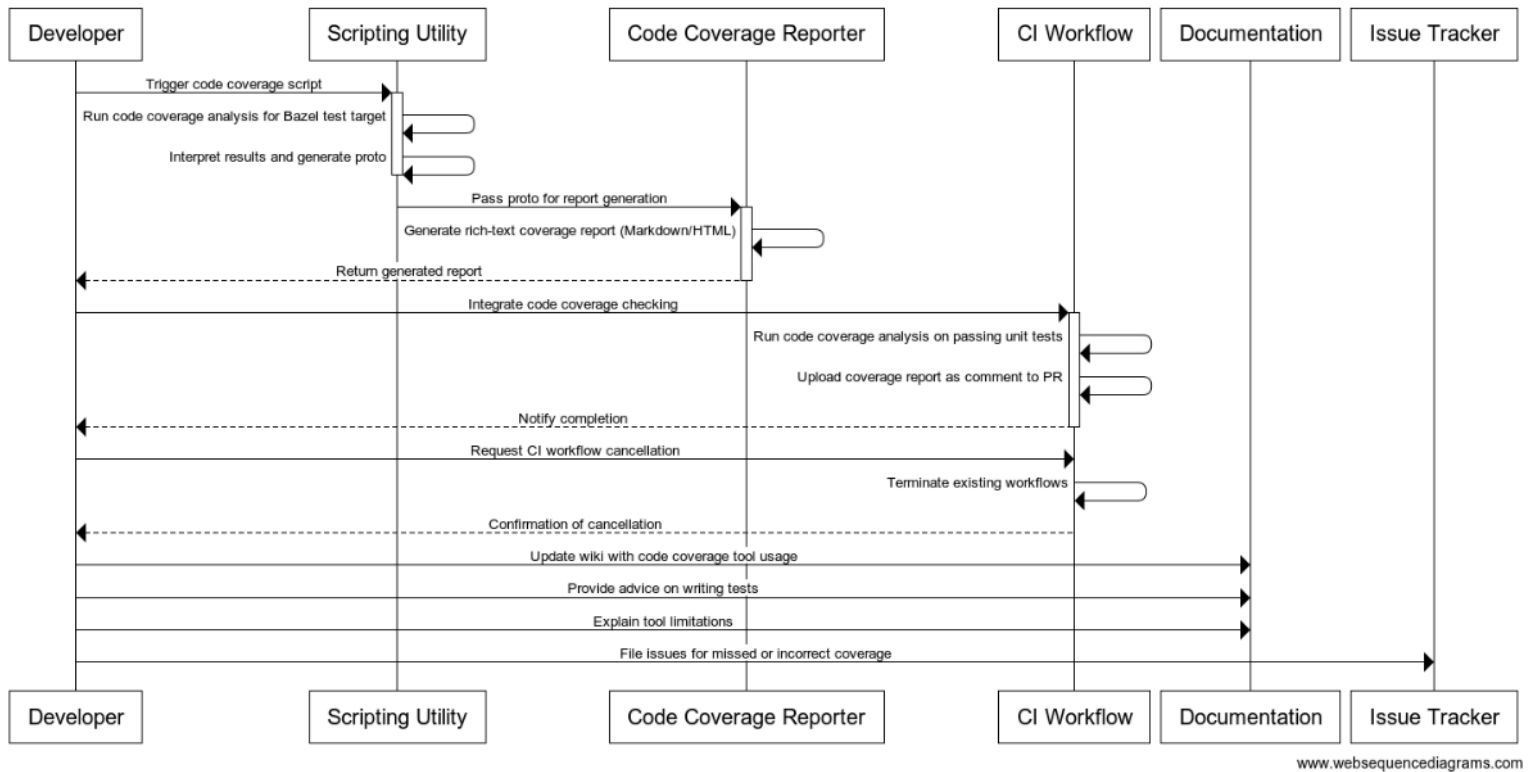
## Solution Overview

### Proposed Solution:

- Introduce code coverage analysis script, generate rich text reports for easy code coverage insights, update test exemption checks.

- Integrate code coverage checking into CI workflow, and ensure proper workflow cancellation for efficient CI/CD operations.

#### : Code Coverage Analysis Workflow



#### Deliverable 1:

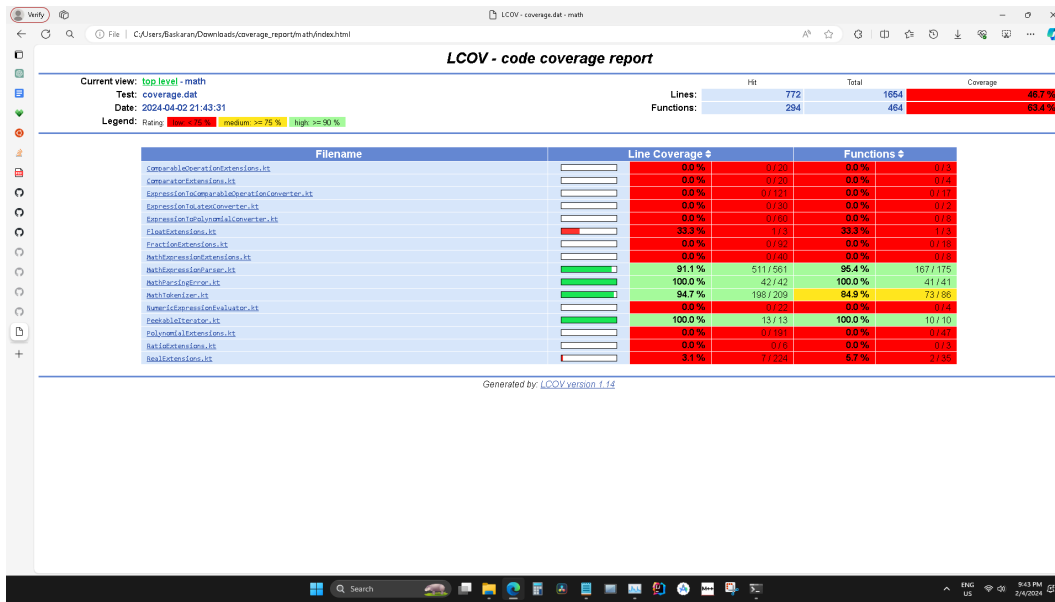
- Developer triggers a script to run Bazel target, generating a lcov / .dat file

#### Run coverage

- As the [PR #4886](#) was blocked I initiated to locally setup the upgrade for **bazel version 7.0.0**. During this process, I successfully generated a code coverage report (.dat file with which I was able to generate html) for the test targets that passed, particularly focusing on those within the math module.
- I have uploaded a separate repository containing the implementation (it may appear messy).
- **Source Link:** [Commit #1](#), [Commit #2](#)
- **Snippet:** ``bazel coverage //:testTarget``
- **Sample coverage.dat report:** [coverage.dat](#)
- **Screenshots:**







## Parse Coverage Data:

Initially, the coverage results obtained from .dat or .info files can be parsed and saved as proto data [[ParseCoverageData function](#)], facilitating the generation of Custom HTML and Markdown (MD) reports.

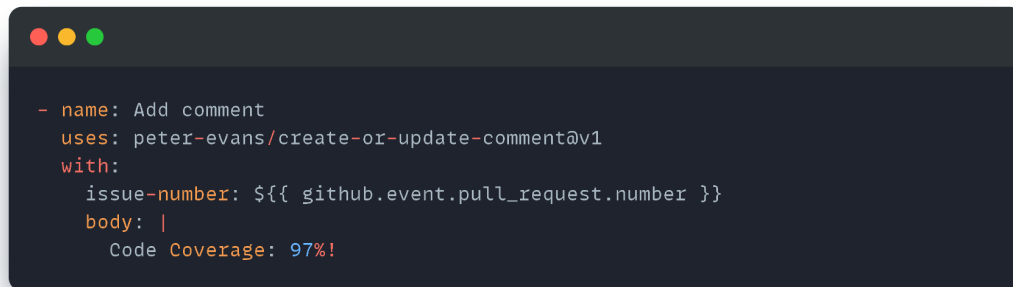
- I attempted to utilize LCOV for generating HTML files, and while it produced valid report results, it should be noted that it only offers line and function coverage by default [[Link to the generated test HTML Coverage Report](#)]. However, if configurable, LCOV could potentially support branch coverage reporting, enhancing its utility for local developer usage. Alternatively, custom HTML reports could be constructed using the parsed data, providing flexibility in report generation.
- Generating MD files directly from .dat files is not feasible and necessitates conversion from either HTML or JSON formats. Despite my attempt to derive MD files from HTML using LCOV, the results were not satisfactory [[Link to the generated test MD Coverage Report](#)], prompting consideration of parsing through custom code. While JSON could serve as an intermediary format, parsing directly from .dat files might offer a simpler approach. I have included sample parser code to illustrate this process as a foundational component [[generate MD Files from the CoverateReport](#)].
- To accommodate both function and branch coverages, appropriate modifications to the proto schema are required. Currently, the function schema with additional necessary fields has been incorporated [[updated proto schema](#)], with plans for further adjustments to support branch coverage.
- Local developers will utilize the HTML report for local coverage analysis, while the CI workflow will utilize the MD report, which will be uploaded as a comment to the respective pull request.

## Unit-Based Coverage Analysis:

To ensure that we enable support for conducting coverage analysis on a per-unit basis, where only the coverage of a specific file, such as Example.kt, is measured when running ExampleTest.kt and no other tests, we have two options available. We can either:

- parse the collected lcov data or
- utilize an [-instrument-filter](#) to extract only the required target coverages.

The automatic Markdown upload is incorporated into the plans for CI. It will be handled in the [deliverable 2 CI/CD workflow](#) by uploading the comment using the following process:



```
- name: Add comment
  uses: peter-evans/create-or-update-comment@v1
  with:
    issue-number: ${ github.event.pull_request.number }
    body: |
      Code Coverage: 97%!
```

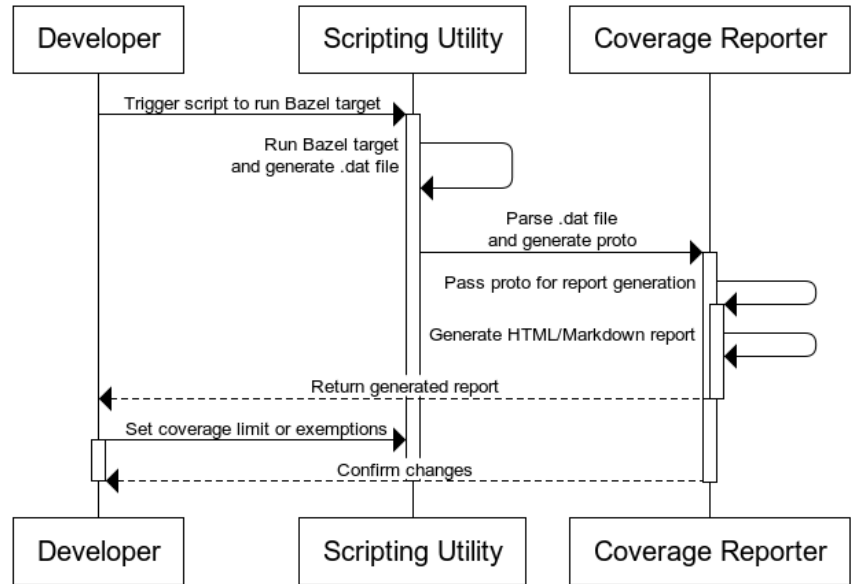
### Textual representation of sample coverage data:

Below is a condensed example of a .dat file and its corresponding coverage proto representation.

.dat file	coverage.proto representation
SF:sample.kt DA:1,1 DA:2,0 DA:3,1 end_of_record	bazel_test_target: "sample" covered_file { file_path: "sample.kt" covered_line { line_number: 1 coverage: FULL } covered_line { line_number: 2 coverage: NONE } covered_line { line_number: 3 coverage: FULL } lines_found: 3 lines_hit: 2 }

I've generated the factual .dat file [[dat file permalink to the test file](#)] for the math module and included their .dat and sample textual representations in this [repository folder](#).

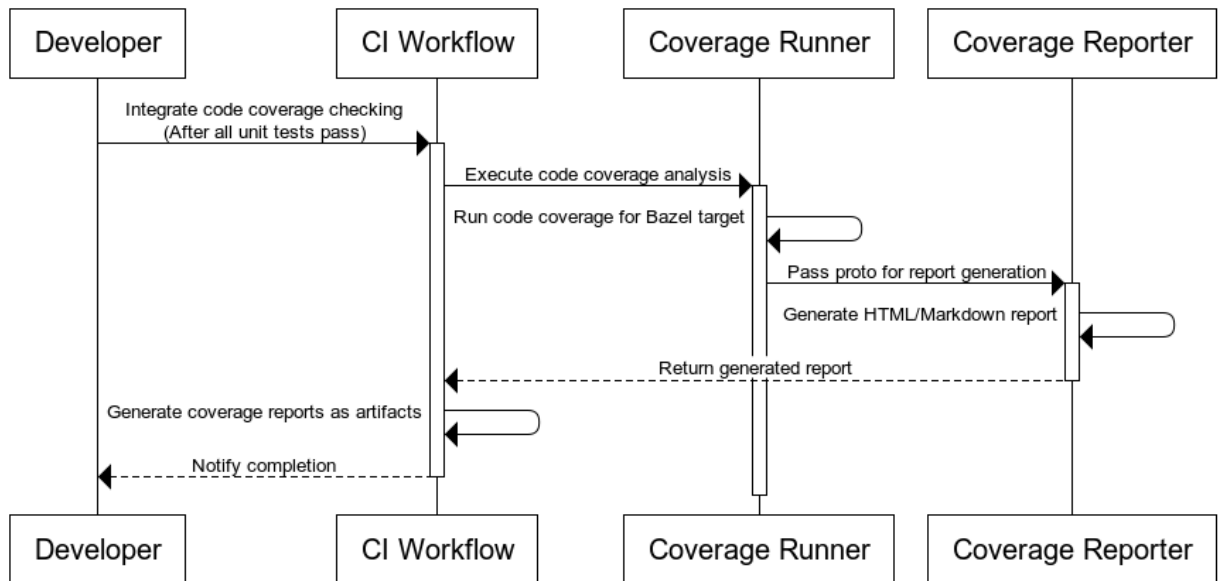
- Developer sets coverage percentage limits via Coverage Runner.



www.websequencediagrams.com

## Deliverable 2:

- CI Workflow starts after passing unit tests.
- Script runs code coverage analysis on target, generates reports, uploads reports as comments.
- Developer configures coverage percentage limits via CI Workflow.



www.websequencediagrams.com

### Current CI workflow requiring rectification:

I aim to delve deeper into the issues surrounding the previous workflow canceller and its relevance to our project during the community bonding period.

In addressing the challenge of canceling all dynamic jobs within workflows, one approach is to leverage the `workflow\_run` event with all workflows listed, as outlined in the [cancel workflow actions repository's documentation](#). This was implemented in [PR #2890](#), and initial focus will be on executing it to assess its effectiveness.

Another potential solution involves utilizing the GitHub API to fetch all currently running workflows and subsequently canceling each related run individually. This alternative method is yet to be tested.

Source: [GitHub API Documentation](#)  
**workflow\_canceller.yml**

```
name: Automatic Workflow Canceller

on:
  workflow_dispatch:
  pull_request:

jobs:
  cancel:
    name: Cancel Dynamic Matrix Jobs
    runs-on: ubuntu-latest
    steps:
      - name: Get currently running workflow runs
        id: get_runs
        run: |
          response=$(curl -s -X GET \
            -H "Authorization: Bearer ${ secrets.GITHUB_TOKEN }" \
            "https://api.github.com/repos/${ github.repository
          }}/actions/runs")
          echo "::set-output name=response::$response"

      - name: Cancel related workflow runs
        if: ${ github.event_name == 'workflow_dispatch' ||
          github.event_name == 'pull_request' }}
        run: |
          runs=$(echo "${ steps.get_runs.outputs.response }" | jq -r
            '.workflow_runs[] | select(.head_repository.full_name ==
            env.REPOSITORY) | .id')
          for run in $runs; do
            curl -X POST \
```

```
-H "Authorization: Bearer ${ secrets.GITHUB_TOKEN }" \
"https://api.github.com/repos/${ github.repository }
/actions/runs/${run}/cancel"
done
```

### Third-Party Libraries

No.	Third-party library name and version	Link to third-party library	Why it is needed	License (if third-party library)	[Android only] Min / target / max SDK version that the library supports
1	Bazel 6.2.0	<a href="#">bazelbuild/bazel</a>	Bazel 4.0.0 lacks support for test coverage for android_local_test. However, with the introduction of android test coverage in Bazel 6.2.0 [Link], upgrading to this version is essential to establish basic code coverage, marking a crucial starting point for this project.	Apache-2.0 license	
2	Create-or-update-comment - v1	<a href="#">peter-evans/create-or-update-comment</a>	To automate the process of uploading the generated coverage report to the relevant pull requests.	MIT license	
3	Cancel-workflow-action - 0.8.0	<a href="#">styfle/cancel-workflow-action</a>	GitHub Action that will cancel any previous runs that are not completed for a given workflow	MIT license	

### “Service” Dependencies

No.	Dependency name	Why it is needed	What our plan is, if the dependency fails under us
1	GitHub CI	Essential for automated testing and code coverage reporting	<b>Investigate:</b> If it's a temporary glitch, we can investigate the issue with GitHub CI and

		and to automate upload of reports as comment to the PR.	<p>retry the pipeline execution.</p> <p><b>Manual execution:</b> If the failure persists, we might need to manually trigger the build and code coverage analysis on a local machine or another CI platform (if available) to ensure coverage is not compromised.</p> <p><b>Alert team:</b> For critical failures, we may need to alert the development team to address potential issues within the codebase or the CI configuration.</p>
--	--	---	--

## Impact on Other Oppia Teams

N/A

## Key High-Level and Architectural Decisions

### Decision 1: Code Coverage Tool

We have considered the following alternatives:

- **Bazel Coverage:** Utilizes built-in coverage analysis features of Bazel.  
(Note: Bazel creates coverage data with Jacoco for lines (and functions) and uses internal Jacoco classes for branches)
- **JaCoCo:** Dedicated code coverage tool for JVM-based projects.

Among these, we believe that Bazel is the best approach because:

1. **Tight Integration:** Bazel provides seamless integration with the build system, enabling efficient coverage analysis as part of the build process.
2. **Consistency and Simplicity:** Using Bazel for both build and coverage ensures consistency in the development workflow, simplifying the toolchain and reducing overhead.

### Feature Evaluation Table:

The above approaches are contrasted in detail in the following table:

Aspect	Bazel Coverage	JaCoCo
Integration	Seamlessly integrates with Bazel	Requires configuration in Gradle
Language Support	Supports multiple programming languages	Primarily focused on JVM-based projects
Build System	Integral part of the Bazel build	Requires separate setup in the build script

Flexibility	Limited customization options	Offers extensive configuration capabilities
Community Support	Well-supported within Bazel community	Widespread adoption in Java ecosystem

Risks and mitigations

Potential Risk	Mitigation
Older Bazel versions may lack compatibility with code coverage tools, leading to inaccurate analysis	Stay updated with stable Bazel releases for improved compatibility and features. Monitor release notes and conduct thorough testing before updating.
Bazel's unique dependency management may pose challenges in handling dependencies, versioning, and conflict resolution.	Define clear dependency management practices aligned with Bazel's model. Regularly review and update configurations to ensure consistency, stability, and security

Decision 2: Coverage Report Generator

We have considered the following alternatives:

- Custom HTML Code: Develop a custom solution to generate coverage reports using a custom proto format and HTML code. [\[Custom HTML report prototype\]](#)
- LCOV's Genhtml: Utilize LCOV's genhtml functionality to generate coverage reports from the .dat files. [\[lcov genhtml report prototyp\]](#)

While lcov genhtml may appear straightforward and convenient, prioritizing custom HTML report generation aligns better with long-term scalability and future prospects. Below, we outline the reasons and evaluations supporting this choice:

- 1. No Additional Configurations Needed:**  
Custom HTML reports require no additional setup or configurations, offering a straightforward solution for coverage reporting without the need for complex tooling or installations.
- 2. Flexibility:**  
With custom HTML, teams have complete control over the layout, styling, and content of coverage reports, allowing them to tailor the presentation to match project requirements and branding guidelines precisely.
- 3. CI/CD Integration and Output Management:**  
Generating final HTML reports using the generated protos is much more straightforward than aggregating genhtml reports. This approach ensures seamless integration into CI/CD pipelines, automating and maintaining consistent report generation without relying on external tools.
- 4. Scalability:**



Custom HTML reports are scalable and adaptable to projects of any size, accommodating diverse requirements and providing insights into code coverage across multiple targets or modules with ease.

**Feature Evaluation Table:**

The above approaches are contrasted in detail in the following table:

Aspect	LCOV Genhtml	Custom HTML Code
Integration	Seamlessly integrates with Bazel	HTML itself requires no additional configuration or integration efforts
Maintenance	Low maintenance burden, standard tooling	Higher maintenance effort for custom solution
Standardization	Ensures consistency across projects	May vary based on implementation and practices
Flexibility	Limited customization options	Provides full control over report format and styling
Reliability	Stable and well-supported	Reliability depends on implementation quality
CI/CD Integration	Requires additional tools and configuration	Seamless integration after initial setup
Output Management	May generate separate assets per target	Optimized for single file outputs
Extendability and Consolidation of reports	Harder to manage with multiple files	Easily extendable with custom features and proto data

Risks and mitigations

Potential Risk	Mitigation
Necessitates manual implementation and ongoing maintenance, demanding continuous oversight to ensure reliability and accuracy.	Facilitate easier management by offering clear documentation and regularly verifying the produced data.

Implementation Approach

Domain Objects

N/A

## User Flows (Controllers and Services)

N/A

## UI changes

This project will primarily entail command-line interface (CLI) operations on the developers' end and will not directly modify the UI of the application or its user content. Although it may include generating reports for analytical purposes in various formats, this process does not directly affect the UI of the application.

This approach ensures alignment with project objectives while minimizing disruptions to the user experience.

## Test data changes

This project will not include any test data changes, but it will involve creating temporary test files similar to how it was done with `ComputeAffectedTestsTest`, where we utilize `testBazelWorkspace's` `createBasicTests` function. Similarly, we can create temporary test files with basic test cases to ensure test coverage works well.

```
/**
 * Function to add two integers.
 * @param a The first integer.
 * @param b The second integer.
 * @return The sum of the two integers.
 *
 * Expected coverage results:
 * - Line Coverage: 100% (all lines executed)
 */
fun add(a: Int, b: Int): Int {
    return a + b
}

fun testAdd() {
    assert(add(1, 2) == 3)
    assert(add(-1, 1) == 0)
    assert(add(0, 0) == 0)
}

/**
 * Function to perform integer division.
 * @param a The dividend.
 * @param b The divisor.
 * @return The result of the division or an error message if division by zero.
 *
 * Expected coverage results:
```

```

* - Branch Coverage: 100% (both branches executed)
*/
fun divide(a: Int, b: Int): Any {
    return if (b != 0) {
        a / b
    } else {
        "Cannot divide by zero"
    }
}

fun testDivide() {
    assert(divide(4, 2) == 2)
    assert(divide(4, 0) == "Cannot divide by zero")
}

/**
 * Function to check if an integer is positive.
 * @param num The integer to check.
 * @return True if the integer is positive, false otherwise.
 *
 * Expected coverage results:
 * - Conditional Coverage: 100% (all conditions tested)
 */
fun isPositive(num: Int): Boolean {
    return when {
        num > 0 -> true
        num == 0 -> false
        else -> false
    }
}

fun testIsPositive() {
    assert(isPositive(5) == true)
    assert(isPositive(0) == false)
    assert(isPositive(-5) == false)
}

```

These test cases are provided as samples and will evolve based on testing needs.

## Testing library changes

There will be no Testing library changes implemented in the project.

## Script & CI changes

### RunCoverage :

The project entails the incorporation of the core RunCoverage script, which acts as the central script responsible for executing Bazel coverage for the test targets. Subsequent enhancements will provide functionalities allowing users to specify minimum coverage percentage thresholds and desired output formats.

Please find the KDoc provided below:

### RunCoverage.kt

```
/**
 * Entry point function for running coverage analysis.
 */
* @param args Command-line arguments.
*/
fun main(vararg args: String)

/**
 * Class responsible for analyzing target files for coverage and generating
 reports.
 */
class RunCoverage {

    /**
     * Analyzes target file for coverage, generates chosen reports accordingly.
     *
     * @param targetFile Path to the file to analyze.
     * @param outputFormats Output formats for the coverage reports.
     * @throws IllegalStateException if computed coverage is below min required.
     */
    fun runCoverage(targetFile: String,
                    outputFormats: List<CoverageReporter.ReportFormat>
                    )

    /**
     * Runs coverage analysis on the specified target file asynchronously.
     *
     * @param targetFile Path to the target file to analyze coverage.
     * @return A deferred result representing the coverage report.
     */
    fun runCoverageAnalysis(targetFile: String): Deferred<CoverageReport>
```

```

/**
 * Generates coverage reports in the specified output formats.
 *
 * @param coverageReport Coverage report obtained from coverage analysis.
 * @param outputFormats List of output formats to generate coverage reports.
 */
fun generateCoverageReports(coverageReport: CoverageReport,
                           outputFormats: List<ReportFormat>
)

/**
 * Computes the coverage percentage based on the given coverage report.
 *
 * @param coverageReport Coverage report obtained from coverage analysis.
 * @return percentage The computed coverage percentage.
 */
fun computeCoveragePercentage(coverageReport: CoverageReport): Int
}

```

#### CoverageRunner.kt:

This addition will introduce the CoverageRunner class, responsible for implementing the Bazel code coverage runner. Its role is to analyze the coverage and provide coverage reports.

```

/**
 * Class responsible for running coverage analysis asynchronously.
 */
class CoverageRunner {

    /**
     * Runs coverage analysis asynchronously for the Bazel test target.
     *
     * @param bazelTestTarget Bazel test target to analyze coverage.
     */
    suspend fun runWithCoverageAsync(bazelTestTarget: String)

    /**
     * Parses coverage data and returns a coverage report.
     *
     * @param coverageDataFilePath Path to the file containing coverage data.
     * @return A coverage report parsed from the coverage data.
     */
}

```

```

    */
    private fun parseCoverageData(coverageDataFilePath: String): CoverageReport
}

```

### CoverageReporter.kt:

Utility class for generating rich-text coverage reports and computing coverage ratios.

```

/**
 * Class responsible for generating coverage reports.
 */
class CoverageReporter {

    /**
     * Generates Rich-text reports based on coverage report and format.
     *
     * @param report The coverage report to be formatted.
     * @param format The format in which the report should be generated.
     * @return The generated rich-text report.
     */
    fun generateRichTextReport(report: CoverageReport,
                               format: ReportFormat): String

    /**
     * Computes the coverage ratio based on the provided coverage report.
     *
     * @param report The coverage report containing coverage data.
     * @return The coverage ratio with the given coverage data.
     */
    fun computeCoverageRatio(coverageReport: CoverageReport): Float

    /**
     * Enum representing different formats for coverage reports.
     */
    enum class ReportFormat {
        MARKDOWN,
        HTML
    }
}

```

### BazelClient.kt:

This will introduce a new function in the BazelClient to runCoverage.

```

class BazelClient() {
    // Existing methods and properties...

    /**
     * Runs code coverage for the specified Bazel test target.
     *
     * @param bazelTestTarget Bazel test target for which code coverage will be
    run.
     * @return generated coverageResult output
     */
    fun runCoverage(bazelTestTarget: String): List<String>
}

```

#### coverage.proto:

This project will entail the addition of coverage.proto for standardized representation and exchange of coverage data between components, enhancing maintainability and extensibility of the coverage analysis tool.

From the Tracking Issue #5343:

```

message CoverageReport {
    string bazel_test_target = 1; // The Bazel test target that was run.
    repeated CoveredFile covered_file = 2; // Files with coverage in this report.
}

message CoveredFile {
    string file_path = 1; // Relative to the project root.
    string file_sha1_hash = 2; // SHA-1 hash at the time of report
                             // (to guard against changes).
    repeated CoveredLine covered_line = 3; // Lines of code covered in the report.
    int32 lines_found = 4; // New field for total lines found in the file.
    int32 lines_hit = 5; // New field for total lines hit in the file.
    repeated CoveredFunction covered_function = 6; // Functions with coverage.
    int32 functions_found = 7; // New field for total functions found in the file.
    int32 functions_hit = 8; // New field for total functions hit in the file.
}

message CoveredFunction {
    string function_name = 1; // Name of the function.
    Coverage coverage = 2; // Detected coverage.
}

message CoveredLine {

```

```

int32 line_number = 1; // 0-starting line number of the covered line.
Coverage coverage = 2; // Detected coverage.

enum Coverage {
  FULL = 0; // This line was fully covered by the test.
  PARTIAL = 1; // This line was partially covered by the test.
  NONE = 2; // This line was not executed during the test.
}
}

```

### script\_exemptions.proto:

From the Tracking Issue #5343:

```

message TestFileExemptions {
  repeated TestFileExemption test_file_exemption = 1;

  message TestFileExemption {
    string exempted_file_path = 1;
    oneof exemption_type {
      bool test_file_not_required = 2;
      int32 override_min_coverage_percent_required = 3;
    }
  }
}

```

### unit\_tests.yml:

Existing CI workflow will be updated in order to support code coverage.

```

jobs:
  ...

  generate_coverage_report:
    name: Generate Coverage Report
    needs: [bazel_run_test]
    if: ${ needs.bazel_compute_affected_targets.outputs.have_tests_to_run ==
'true' && needs.bazel_run_test.result == 'success' }}
    runs-on: ubuntu-20.04
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

```



```
- name: Set up JDK 9
  uses: actions/setup-java@v1
  with:
    java-version: 9

- name: Generate coverage report
  # run code coverage script

- name: Add comment
  # upload comment
```

## Documentation changes

As this targets developers, it necessitates adjustments to the documentation, I intend to introduce four key additions to the project's wiki page. To incorporate the described documentation steps into your project proposal, you can outline them as follows:

### **Introduce a Wiki Page for Code Coverage Tool Usage:**

- Create a comprehensive Wiki page within the project repository to provide detailed instructions on how to utilize the code coverage tool effectively.
- Include step-by-step guides, code snippets, and examples to demonstrate how developers can integrate the tool into their workflow and interpret the coverage reports.

### **Provide Guidance on Writing Tests for Good Behavioral Coverage:**

To ensure the creation of effective tests, the [Android-wiki presentation](#) outlines several crucial aspects. Additionally, I aim to enhance the Developer Workflow section by incorporating additional samples and tutorials on testing methodologies, including classic testing, Test-Driven Development (TDD), and my approach. TDD plays a pivotal role in enhancing test quality by fostering a mindset focused on testability during code writing, ensuring comprehensive testing of new code prior to deployment.

Code coverage massively improves with thorough testing practices and meticulous attention to various aspects of code functionality and behavior. I would like to initially start off with the following tutorials/docs explaining their vitality for code coverage to the wiki page.

A sample document has been compiled, featuring code snippets and tests. Further enhancements are underway to include additional illustrations, aiming to enrich comprehension and provide an interactive learning journey. - [\[Link to the Sample document file\]](#)

### **1. Boundary Value Testing :**

Boundary Value Testing ensures that test cases cover input values at the edges of acceptable ranges, helping to uncover potential edge case issues in software.

**2. Condition Coverage :**

Condition Coverage validates that both true and false conditions in the code are thoroughly tested, ensuring comprehensive assessment of code behavior under different conditions.

**3. Decision Coverage :**

Decision Coverage ensures that all possible outcomes of conditional statements are tested, guaranteeing comprehensive evaluation of code behavior for different decision paths.

**4. Loop Testing :**

Loop Testing involves testing loops with both minimum and maximum iterations to ensure correct functionality and handling of loop boundaries and conditions. This ensures robustness and reliability in loop-based code structures.

**5. Exception Handling :**

Exception Handling testing focuses on ensuring that exceptions are handled appropriately within the code. This involves verifying that the code reacts correctly to exceptional situations, such as invalid inputs or unexpected conditions, by throwing or catching exceptions as required.

**6. Overflow Testing :**

Overflow Testing ensures software can handle situations where variables exceed their limits, like numeric values reach the maximum or minimum representable values

**Explain Limitations of the Code Coverage Tool:**

- Detail the limitations and edge cases where inaccurate coverage metrics are reported.
- Provide examples and explanations to help developers understand why certain code may not be correctly counted in coverage reports.

**File Issues for Code Coverage Tool Limitations:**

- Identifying and document cases where the code coverage tool fails to accurately count coverage.
- Create issue tickets in the repository for each identified limitation or missed coverage scenario.
- Include detailed descriptions, reproducible examples, and proposed solutions or workarounds for addressing these issues.

## Metrics Plan

Event (see PRD)	Event parameters (see PRD)	Do we already record the event + parameters? <ul style="list-style-type: none"><li>• If so, please link to the corresponding code on GitHub.</li><li>• If not, describe the changes needed to do so.</li></ul>
N/A	N/A	N/A

# Testing Plan

## Acceptance tests for core user flows

#	Test name	Initial setup steps	Steps	Expectations
1.	RunCoverageTest: minimum_coverage_successful, minimum_coverage_fail	Ensure project setup	Set minimum coverage for pass and fail cases	The script executes without errors
			Run RunCoverage script	Coverage percentage meets or exceeds the minimum coverage specified
				Coverage percentage fails the minimum coverage and fails
2	RunCoverageTest: no_test_target_found	Ensure project setup	Run RunCoverage script with available file	The script passes if file available
			Run RunCoverage script with unavailable file	The script fails with an IllegalArgumentException if file not found
				Error message indicates if no corresponding test target found
3	CoverageReporterTest: invalid_coverage_arguments	Ensure project setup	Run the RunCoverage script with invalid arguments	The script fails with appropriate error messages indicating the nature of the invalid or missing arguments
			Run the RunCoverage script with missing arguments	
4	CoverageReporterTest: coverage_report_generation	Ensure coverage data available	Call the generateRichTextReport with coverage report and specified format	The report is generated in specified format
			Call the generateRichTextReport with coverage report and no format specified	Coverage reports are generated in default format

5	CoverageReporter Test: coverage_ratio_computation	Ensure coverage data is available	Call the computeCoverageRatio method with coverage report	The method executes without errors
				Coverage ratio is computed accurately

## Implementation Plan

### Community Bonding Period:

I consider the community bonding period to be crucial, as it allows me to analyze the project, identify areas where I need to improve my skills, and begin building a prototype once the necessary blocked PRs are merged.

The tasks that must be predominantly finished before the GSoC period commences should encompass:

- Making sure the bazel tests work properly as mentioned in the discussion as that is hindering the main process of testing many features.
- Additionally, getting the [PR #4886](#) merged is crucial.

If those were done right then the project flow should be fairly smooth. I would like to spend my community bonding period

- Initially to get cleared up on any parts of the proposal.
- Followed by assisting in the mentioned 2 key steps before the GSoC starts.

## Milestone Table

NB: This includes both PRs and other actions that need to be taken prior to launch.

### Milestone 1

#### Key objective for this milestone:

No	Description of PR / action	Prereq PR numbers	Target date to start working on the PR	Target date for PR creation	Target date for PR to be merged
	<b>MILESTONE 1</b>				
1.1	Introduce RunCoverage script, CoverageRunner utility to execute Bazel coverage command for a single test target '//:test_target' and add tests for script execution.	None	27-05-2024	01-06-2024	03-06-2024
1.2	Update Test Exemption Check scripts and add tests for	None	01-06-2024	03-06-2024	05-06-2024

	TestFileCheckTest and related scripts as required.				
1.3	Update / Implement RunCoverage script to run code coverage for a specific file 'filename.kt' replacing test target argument and add test for the Bazel command execution with filename..	1.1	03-06-2024	08-06-2024	10-06-2024
1.4	Extract and Parse related the coverage data to store them in proto and add tests to validate them.	1.3	08-06-2024	13-06-2024	15-06-2024
1.5	Introduce CoverageReporter utility to generate code coverage reports in Markdown and HTML, compute coverage ratio and add tests to validate them.	1.4	13-06-2024	20-06-2024	22-06-2024
	<b>Evaluation</b>		<b>Starting Date</b>	<b>Creation Date</b>	<b>Merge Date</b>
	<b>Buffer Time</b> To work on reviews and code changes.		22-06-2024	26-06-2024	28-06-2024
	<b>Midpoint Evaluation</b> Code Reviews and Evaluations		28-06-2024	08-07-2024	12-07-2024

## **Milestone 2**

**Key objective for this milestone:**

<b>No .</b>	<b>Description of PR / action</b>	<b>Prereq PR numbers</b>	<b>Target date to start working on the PR</b>	<b>Target date for PR creation</b>	<b>Target date for PR to be merged</b>
	<b>MILESTONE 2</b>				
2.1	Introduce new CI workflow for code coverage	1.4	12-07-2024	22-07-2024	24-07-2024
2.2	Upload generated markdown report as comment	2.1	22-07-2024	26-07-2024	28-07-2024
2.3	Fix/replace cancellation workflow	None	26-07-2024	01-08-2024	03-08-2024

2.4	Create wiki page explaining code coverage usage, limitations, file issues for coverage gaps, and test writing tips		01-08-2024	04-08-204	06-08-2024
	<b>Evaluation</b>		<b>Starting Date</b>	<b>Creation Date</b>	<b>Merge Date</b>
	<b>Buffer Time</b> To work on reviews and code changes.		06-08-2024	10-08-2024	12-08-2024
	<b>Final Evaluation</b> Code Reviews and Evaluations		12-08-2024	26-08-2024	03-09-2024
	Fix any further issues			Spare intervals	Spare intervals
	<b>End of GSoC period</b>				03-09-2024

## Future Work

### Continual Documentation and Test Improvement:

1. Provide ongoing documentation updates on best practices for writing effective tests to achieve better code coverage.
2. Regularly review and update documentation to incorporate new insights and improvements in testing methodologies.

### Addressing Limitations and Filing Missing Code Coverage Issues:

Continuously monitor and address any limitations or gaps in code coverage analysis.