

Section 1: About You

Goals and Objectives

Why do you want to do a GSoC project with Oppia?	I want to learn more about contributing and gain experience on real-life coding, while also contributing to a good cause like Oppia.	
What do you hope to learn/achieve during GSoC?	1. While I have some experience with backend unit tests, I hope to learn much more about this especially since we will be reducing backend unit test times through interesting topics like mocking/optimization. 2. Impact measuring, I hope to learn more about how to measure impact and how to know if my solutions have improved developer experience.	
Which Oppia teams have you collaborated with, and what have you done on those teams?	LaCE	N/A
	Developer Workflow	N/A
Contact information	jn-nguyen@outlook.com	
Preferred method of communication	Email	
Which timezone will you primarily be in during the summer?	PST	
(If you are a student) When are your school holidays?	Mainly just summer which starts after June 14th.	
What other obligations might you need to work around during the summer?	None, except will be less active from May 1-June 14th due to school.	
Planned time commitment	I plan to commit to this project 5-7 days per week, around 6-8 hours per day.	

Section 2: About Your Project

Project Details

Project title <i>(should match the one on the Project Ideas list)</i>	3.2 Make CI and pre-push hooks more efficient.
Project size <i>(should match the options on the Project Ideas list)</i>	Large (~350 hours)
Why did you choose this project?	First of all I have had many personal problems with developer experience and this mainly consisted of long frontend tests running for minimal changes. I would always be frustrated when I make small changes to commit and this results in waiting a long time for frontend tests to run even when they aren't needed. I also believe that this project will greatly improve developer experience in general by reducing pre-push hook times through running certain tests (frontend) on changed files only. It will also ensure code stability for PRs since we now require backend unit tests to be run. Furthermore the idea of running only certain tests for PRs that only have frontend changes is also great since it will reduce the time developers take to analyze their changes and also reduce CI resources.

Required Skills

WEB	
I can write Python code with unit tests.¹	https://github.com/oppia/oppia/pull/19877
I can write TS + Angular code with unit tests.²	https://github.com/oppia/oppia/pull/19549 https://github.com/oppia/oppia/pull/19877 https://github.com/oppia/oppia/pull/19587
I have good UI judgment and attention to detail.³	N/A, I have done various frontend bug fixes though. https://github.com/oppia/oppia/pull/19549 https://github.com/oppia/oppia/pull/19532 https://github.com/oppia/oppia/pull/19678
I can debug and fix CI	https://github.com/oppia/oppia/pull/19725

¹ To develop this skill: Some [LaCE](#) and [Contributor Dashboard](#) issues have a backend component, and many [Dev Workflow](#) issues do too. Focus on issues that aren't marked as "backlog".

² To develop this skill: Most [LaCE](#) and [Contributor Dashboard](#) issues have a frontend component. Focus on issues that aren't marked as "backlog".

³ To develop this skill: Tackle a non-backlog responsiveness issue from the [LaCE team](#).

failures / flakes. ⁴	https://github.com/oppia/oppia/pull/20035 https://github.com/oppia/oppia/pull/20021
I can write or modify e2e or acceptance tests. ⁵	https://github.com/oppia/oppia/pull/19877 https://github.com/oppia/oppia/pull/19587 https://github.com/oppia/oppia/pull/19938
I can communicate effectively using debugging docs . ⁶	N/A
I can write or modify Beam jobs . ⁷	N/A
I have participated in QA testing. ⁸	N/A, but I have filed various bugs that I encounter and have done testing when trying to track console errors. https://github.com/oppia/oppia/issues/19545 https://github.com/oppia/oppia/issues/19607 https://github.com/oppia/oppia/issues/19704

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	I will adhere to the above deadlines.
---------------	---------------------------------------

Communication Channels

Note: The Oppia team places a high emphasis on communication, and we have found that daily contact between contributors and mentors is important for helping keep projects on track. This is why we ask that contributors send short daily updates to their mentors explaining what they have done, where they are stuck, and what they plan to do next.

I can commit to sending daily updates to my mentor by email, each day I work during the GSoC period.	Yes
In addition to the above: how	I plan to contact my mentor through Google Chat whenever I

⁴ To develop this skill: See issues labelled "[CI breakage](#)" on GitHub, or look at [CI failures in develop](#) (or cross-signs [here](#)) and figure out how to fix them.

⁵ To develop this skill: Solve part of issue [#17712](#) by covering the CUJs for one or more sets of users.

⁶ To develop this skill: Tackle an issue that requires creating a debugging doc that you share with the broader team. Fixing CI failures counts.

⁷ To develop this skill: See [this doc](#) for some examples of issues to work on.

⁸ To develop this skill: Join the release testers mailing list at oppia-release-testers@googlegroups.com, and look out for calls to help with release testing.

often, and through which channel(s), do you plan on communicating with your mentor?	need help and also to have at least one meeting per week to discuss the progress.
---	---

Section 3: Proposal Details

Problem Statement

Target Audience	Oppia Developers
Core User Need	Right now whenever developers push their code they are forced to go through a long pre-push hook to run frontend tests that their code might not even affect . This causes distress for developers whenever they push their code. Alongside this, we have a lack of backend unit tests running on pre-push hooks, resulting in excess commits and drainage of CI resources on github as simple failing backend tests are not detected . The backend tests and also E2E tests take a long time , I usually see these E2E tests in the CI running for around 30 minutes for a build and 10-15 minutes for parallel E2E/acceptance tests and for lighthouse tests it takes around 30 minutes per shard. However, this is if all tests run in parallel, which isn't always guaranteed since runners aren't always free. Reducing the time that backend and E2E tests take will also reduce CI resources and improve developer experience.
What goals do we want the solution to achieve?	The goals we want to achieve with this solution are to introduce backend unit tests for files changed and also reduce the time backend unit tests take. We also want to make frontend tests run for only changed files and to overall reduce the time pre-push hooks take. The E2E/acceptance/lighthouse tests time should also be reduced by introducing a way to run partial tests depending on the PR and its changed files. Alongside all of this, we should ensure that code is still stable by making sure that code is still fully checked in merge queues.

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 that"</i>	List of tasks needed to achieve the goal (this is the "User Journey")
1	Running Backend Unit Tests	As a developer, I need backend unit tests that are well developed and	Run the backend unit tests.

		don't have unnecessary calls to the backend, so that I can ensure that my tests work while still running quickly.	Observe that the backend unit test runs quickly so I can push my code or merge my PR.
2	Reporting long backend unit tests.	As a developer and developer work team member, I want to be able to see any long backend tests so that I can note any new long backend tests and coordinate to improve times of the particular backend test.	<p>The backend unit tests run on the github CI.</p> <p>Observe that any tests that took longer than 150 seconds, are reported at the end of running the backend unit tests inside the CI log.</p>
3	Running pre-push hooks.	As a developer, I need pre-push hooks to catch simple errors that can be caught by lint checks and frontend/backend unit tests, so that I can get quick feedback on my code. However, at the same time the pre-push hook shouldn't take more than 5 minutes because then the SSH connection I'm using to push will time out.	<p>Push code to the github repository.</p> <p>Observe that the pre-push hooks run the following:</p> <ul style="list-style-type: none"> • Lint checks (HTML/pylint/eslint/stylelint) • Other file based checks (Typescript, MYPY). • Backend unit tests • Frontend unit tests <p>These tests should only be run on files that are changed in the particular commit I am pushing.</p> <p>Observe that the pre-push hook completes within 5 minutes and that any overflow backend unit tests that run over this 5 minute limit are forcefully stopped and a log should show with the tests that were stopped and the commands to run them. The shortest backend tests should also run first so the pre-push hook can fit as many tests as possible within 5 minutes.</p>
4	Run lighthouse/acceptance tests	As a developer, I expect lighthouse/acceptance tests to run only when necessary (i.e. when the code in my PR actually changes something occurring in the test), so that the CI can check the effects of the changes I made faster.	<p>Push a new commit to a PR</p> <p>Observe that if my PR code only contains frontend changes, then only tests which my code affects are run.</p> <p>Observe that if my PR code contains any backend changes, then the whole suite of tests are run. Below is some examples of file changes and what tests will run:</p> <ul style="list-style-type: none"> *.py - All tests will run. README.md - no tests will run. Specific test - only that test will run. *.ts - Only tests that are dependent on that file

			will run.
5	Running tests in merge queues.	As a developer or maintainer, I expect all tests (acceptance/lighthouse/backend/frontend) to pass prior to any of my PRs being merged, so that I have assurance that I'm not breaking develop for other developers.	<div>Click the "Merge when ready" button on github.</div> <div>Observe that all tests are run regardless of what the PR code changes.</div>

Section 3.2: HOW

Existing Status Quo

Pre-Push Hooks (Takes ~7 minutes on my computer):

- Runs lint checks on changed files (The lint checks include HTML/pylint/eslint/stylelint and other custom linters like Code Owner linter).
- Runs all Typescript/MYPY checks on all files (only on commits that have Typescript changes for Typescript and Python changes for MYPY).
- Runs all frontend tests regardless of changed files (only on commits that have frontend changes with Typescript or JavaScript).
- Does not run any backend unit tests at all, including for changed files.

Backend Unit Tests (15-24 minutes per shard, a total of 5 shards):

- Backend unit tests take quite long and many backend unit tests can be optimized or mocked to decrease the time it takes.

Pull Request CI (30 minutes for a build and 5-15 minutes for each E2E/acceptance test, 28 minutes per lighthouse shard):

- Currently all acceptance/lighthouse tests run for every PR, which wastes PR resources on small changes like frontend bug fixes.

Solution Overview

Optimizing and running partial frontend/backend unit tests on the pre-push hook:

1. Shorten the install-and-build setup script, and remove the install-and-build step completely from the pre-push hook.

Here from analyzing the code, I believe that the install-and-build step primarily stems from the *run_lint_checks* install step which ensures that all third party libs are installed. From there other pre-push checks use the *-skip-install* flag to skip installation as the *run_lint_checks* has already ensured that those packages are installed. This step is quite vital to ensure that the developer's environment is correctly set up and ready to be used for tests like frontend/backend/lint. However, since Oppia is completely moving towards using a docker setup, we can safely remove the install-and-build from the pre-push hook since docker build already ensures that all requirements are installed/built.

However, in general we can shorten the install-and-build step. From testing, this initial step actually only runs for about 18 seconds (given that the environment is actually set up, e.g. all libs are already installed, which is usually the case).

Now, this might not be the case on other computer systems and OS, I looked at the profiling of the *install_third_party_libs* and it looks like the **longest calling function is the *setup.py* script**. Further investigation shows a comment on how **recursive chown and chmod take long on some machines**. While it only **takes ~5 seconds on my computer**, this comment leads me to believe it could **take much longer on other hardware or OS**. One potential solution to this is just **using *os.system* to call *chown* and *chmod* (we can use *chown -R* or *chmod -R* to recursively set permissions)**. My best guess would be that the time complexity of the existing *chown* and *chmod* come from walking through directories and files and calling *chown/chmod* multiple times. **The built in linux *chown/chmod* recursion could be faster** and doing this on my computer brings the runtime of *install_third_party_libs* from ~13 seconds to ~6.5 seconds on my computer (and could potentially fix the long running times of *chown/chmod* on other hardware/OS).

NOTE: Most of my testing came after my environment was already set up (e.g. already ran *install_third_party_libs*, which should be the case in most situations, though deleting *node_modules* to simulate the *chown* and *chmod* was significantly faster using *os.system*).

PRE-PUSH HOOK MIGRATION TO DOCKER:

When we completely migrate to docker, the install-and-build step can be safely removed from the pre-push hook since all requirements are installed and built by docker through the *devserver* command.

EXPECTED CHANGES:

1. In *common.py*, remove the *os.walk* for loop inside both *recursive_chown* and *recursive_chmod*, and instead use *os.system* with the recursion flag (*-R*).

2. Remove the install-and-build step completely from the pre-push hook since we are using docker now, which ensures that all requirements are installed/built when creating images.

2. Reduce the time frontend tests take by running only the frontend tests that are needed (changed files).

Problem: The main problem here is that when running the pre-push hook, **all** frontend tests are run and this can be quite frustrating as flakes might pop up and the **full** suite of frontend tests is generally quite long (from my **own** computer it takes **around 5 minutes for the frontend tests**, though I have observed that **other computers may take much longer like the M1 Mac** from looking at pre existing issues/discussions on github), leading to a hindrance in developer workflow.

PROPOSED SOLUTION:

1. FINDING CHANGED FILES:

First off we have to find what files have changed, this is quite straightforward as there is already a git command to do this "*git diff*". Below I highlight two approaches that we can take regarding this and I believe that **using the current commits is the best approach in this situation**. Given this, if we choose to go **with the current commits approach** we can **utilize the already existing functionality** inside `pre_push_hook.py` to get changed files (since **they are already utilized for lint checks**). There are two approaches to using *git diff* in this case, we can use *git diff* to get the changes from the most recent commits or we can use it to get the difference between our **feature** branch and the upstream **develop** branch.

Current Commits (Local Branch and Remote Branch Changes)	Full Change List (Local Branch and Upstream Develop Branch Changes).
The benefit of using current commits would be that there are shorter push times. In the context of unit tests (backend/frontend), I believe that this would be a better approach than using the changes from the develop branch since unit tests are designed to be isolated . Essentially any changes that you make to a certain file should be completely covered by its corresponding unit test and shouldn't affect other files that are changed.	The downside of using the changes from the develop branch is that if you commit a lot and then let's say you have some <i>trivial</i> name change that you need to make, it will run all frontend/backend unit tests (which could be a further downside if you made a change to a backend file with a large backend unit test). Overall, it shouldn't be necessary to run unit tests like this since frontend/backend unit tests are isolated and running it on all commits that are changed from the upstream develop branch is not needed, especially since these tests will be run on the PR again.

2. ALTERING KARMA TO RUN PARTIAL TESTS:

Currently, when we execute frontend tests, we utilize a `combined-tests.spec.ts` file which runs all the frontend tests based on a regex. However, this approach is no longer desired as we don't want all

tests to execute within the pre-push hook, and sometimes developers may only want to run tests for a specific file.

To address this issue, I propose using Webpack's `ContextReplacementPlugin` (<https://webpack.js.org/plugins/context-replacement-plugin/>). This plugin allows us to override Webpack's `require` method, which we use to bundle all the tests together. This solution **enables** us to significantly **reduce** memory usage and bundling time when running *partial* frontend tests, as we directly alter how Webpack collects files instead of filtering them at runtime.

Below is the steps we would take to implement the above solution:

1. Introduce a **new** `-specs_to_run` argument inside `run_frontend_tests.py` which takes a space/comma delimited list of files.
2. Inside `pre_push_hook.py` we can **alter** the run command for `run_frontend_tests` to pass in the list of *git diff* files we get (that are frontend, e.g. have `.ts` or `.js` extensions), we will append the `.spec.ts` extension for frontend files that aren't tests since we want karma webpack to require the test files and not the changed frontend files. To get the *git diff* files we can get the *git diff* using the most recent commits (which is already implemented in the pre push hook script) and we can add a way to filter for frontend files.
3. In `run_frontend_tests.py` with this **new** argument, we should **forward** the list of files to the karma runner with an argument like `-specs_to_run` where we can specify the files that we want karma to run. **Inside** `karma.conf.ts` we will introduce the webpack plugin called `ContextReplacementPlugin`. This plugin allows us to override the default functionality of `require.context`. First we should get the comma delimited list of files **passed** through `-specs_to_run` and in `karma.conf.ts` we will **parse** these files into an array. If there are any present specs from the argument to run we will add a **new** context that maps the file (making webpack only be able to require these files) and push a new `ContextReplacementPlugin` with this information. I also moved the default regex to the `karma.conf.ts` file so we can test against files in both the situations where we provide a list of specs and where we don't provide a list of specs, since we want certain files to be **excluded** regardless of whether we run all specs or run specific specs. Here if there are no *specs* specified then the context will simply be defaulted to our default regex and if there are *specs* then we will map the provided files and also check it against the default regex just in case any of the files should be excluded (the default regex includes exclusions for folders we don't want to test).
4. This solution allows *partial* tests to run depending on the argument passed to `run_frontend_checks` and also allows us to leave the **combined specs** file mainly alone, with

only minimal changes to `karma.conf.ts`. Below is a proof of concept and image of it working:

```
1  var specsToRun = [];  
2  if (argv.specs) {  
3    specsToRun = argv.specs.split(',');  
4  }  
5  
6  const SPECS_PATTERN = /(\\.|s|S)pec\\.ts$|(?<!services_sources)\\/([w\\d\\.\\-])* (component|controller|direct  
7  const webpackPlugins = [];  
8  
9  let newContext = SPECS_PATTERN;  
10 if (specsToRun.length) {  
11   newContext = specsToRun.reduce((context, file) => {  
12     if (!SPECS_PATTERN.test(file)) {  
13       return context; You, 1 second ago • Uncommitted changes  
14     }  
15     const relativeFile = `.${file}`;  
16     context[relativeFile] = relativeFile;  
17     return context;  
18   }, {});  
19 }  
20  
21 webpackPlugins.push(  
22   new webpack.ContextReplacementPlugin(  
23     /(:?)/,  
24     path.resolve(__dirname, '..', '..'),  
25     newContext  
26   )  
27 );
```

Above is the implementation I **created to run partial frontend tests** and the code above alters the `karma.conf.ts` file. Like I explained I **created a local constant to hold the old regex which encapsulates all spec files in the code base**. If there are no specs to run (e.g. no `-specs_to_run` option is passed through to jasmine), then we simply use the local constant to show how webpack should collect required files using `ContextReplacementPlugin`. **If there are specs provided**, we would instead **create a mapping that webpack can use to require files, essentially pointing webpack to what files it should collect using `ContextReplacementPlugin`**.

```
jnguyen@JNLAPTOP:~/oppia$ ../oppia_tools/node-16.13.0/bin/node --max-old-space-size=4096 ./node_modules/kar  
ma/bin/karma start core/tests/karma.conf.ts --specs=core/templates/pages/maintenance-page/maintenance-page.  
component.spec.ts,core/templates/pages/about-foundation-page/about-foundation-page.component.spec.ts  
Seed for Frontend Test Execution Order 421
```

```
Chrome Headless 121.0.6167.85 (Linux x86_64): Executed 7 of 7 SUCCESS (0.242 secs / 0.207 secs)  
TOTAL: 7 SUCCESS  
TOTAL: 7 SUCCESS  
jnguyen@JNLAPTOP:~/oppia$
```

```
TOTAL: 7 SUCCESS  
jnguyen@JNLAPTOP:~/oppia$ ../oppia_tools/node-16.13.0/bin/node --max-old-space-size=4096 ./node_modules/kar  
ma/bin/karma start core/tests/karma.conf.ts  
Seed for Frontend Test Execution Order 335
```

```

0F4VG2AAAB with id 795783
Chrome Headless 121.0.6167.85 (Linux x86_64): Executed 53 of 8949 SUCCESS (0 secs / 0.231 secs)
22 03 2024 14:48:03.927:WARN [web-server]: The 'customFileHandlers' is deprecated and will be removed in Ka
rma 7. Please upgrade plugins relying on this provider.
22 03 2024 14:48:03.928:WARN [web-server]: 404: /assetsdevhandler/exploration/expId/assets/image/img_202107
Chrome Headless 121.0.6167.85 (Linux x86_64): Executed 55 of 8949 SUCCESS (0 secs / 0.255 secs)
22 03 2024 14:48:03.951:WARN [web-server]: 404: /assetsdevhandler/exploration/expId/assets/image/img_202107
Chrome Headless 121.0.6167.85 (Linux x86_64): Executed 142 of 8949 SUCCESS (0 secs / 1.857 secs)

```

3. Introduce backend unit tests to pre-push hooks and maintain a list of “long” backend unit tests by creating a reporter and skip these long tests when running the pre-push hook.

Problem: The main problem here is that backend unit tests are not run inside pre-push hooks which can lead to many errors prior to pushing code to the PR. Alongside this many backend unit tests take quite long due to needing to test data scalability and therefore if introduced in the pre-push hook would cause long wait times for the developer.

PROPOSED SOLUTION:

CREATING A REPORTER:

First off we should introduce a new reporter which detects which individual tests take too long given some threshold constraint (**say 150-200 seconds**). This long backend test reporter will only run on the Github CI, so it isn't a reliable source to decide what tests to skip in the pre-push hook.

Use the long backend test reporter to maintain a list of backend unit test times as an artifact to decide what order pre-push hook tests should run in.	Use the long backend test reporter to maintain a list of long backend tests to skip in the pre-push hook.
This is a more reliable approach, since Github CI backend test times will be not similar in scale to the pre-push hook but will be relatively the same in terms of sorting. We can have a script which combines all the test times, uploads the test times sorted as an artifact and in the pre-push hook we will use this to sort what tests we should run (tests will be run in groups for concurrency).	This is an unreliable approach since the Github CI backend test times are much longer than the ones in the pre-push hook. Many tests that run over 120 seconds in the Github CI actually run less than 120 seconds in the pre-push hook, even without optimizations. This is mainly due to the slow nature of Github runners and sharding at maximum concurrency (25). Therefore if a test takes long on the Github CI it doesn't necessarily mean it will cause the pre-push hook to timeout at the SSH limit of 5 minutes, so the above would be an unreliable approach.

Therefore the best approach would be to not maintain any list of long backend tests to skip in the pre-push hook and instead use the long backend test reporter to make a list of sorted backend test times

and report any tests that take over 150 seconds on the Github CI. Therefore instead the pre-push hook's backend tests will be grouped and run in concurrent threads based on which tests take the shortest and will cancel any running tests near the end, leaving a message with the tests that were canceled so the developer can run them on their own time (though all tests will be run on the Github CI).

Time estimation

Situation

- **A developer makes changes in the following files:**
 - 10 typescript files (*.ts),
 - 8 Python (*.py) files.
 - 3 HTML (*.html) files.

Estimated time occupied:

Check name	Min time (mins)	Max time (mins)
Lint check	1	2
Frontend test	0.5	1
Backend test		4
Total		7

In situations like this, the lint checks and frontend checks will run first so they occupy max 3 minutes. From there, we will run the backend checks and we should keep the pre-push hook under 5 minutes so in this scenario the backend tests will need to run for around 2 minutes. While we considered create a long test list which we will exclude any tests that run more than 2 minutes in the Github CI, as highlighted above this would be unreliable so instead we should order the backend tests by time (download an artifact from the Github CI) and any tests that potentially go past this 5 minutes mark, we will cancel them, outputting a message to developers. Next, to create a **reporter** for long backend unit tests, it is quite straightforward as the `run_backend_tests` script already includes a way to get a test's time. We can **gather all the tests and their times and filter all tests** that are greater than the time threshold and report them at the end of running backend unit tests in the PR CI. This report should just be in the CI log, but we should also gather the test times and sort them by shortest to longest so the pre-push hook can download and decide which tests to run first.

CREATING A LONG BACKEND TEST REPORTER:

The long backend test reporter is basically a script which will combine all the different backend shards tests and their times in the Github CI and output a message for any tests that are more than 150 seconds. This script will also combine all tests and their times and output an artifact containing a sorted list from shortest to longest.

Initially, we will use this long backend test reporter to get a list of tests we should look into and see if we can **reduce the time through mocking or optimizing**. The long backend test reporter will also be useful for future development so reviewers/developers can see if a PR adds any new long backend tests.

RUNNING BACKEND UNIT TESTS INSIDE THE PRE PUSH HOOK:

To run backend unit tests in the *pre-push hook* we should add a new backend unit test check in the `pre_push_hook` script. This script already contains a way to get the differential or changed files as lint files rely on this. Using this existing functionality we can filter the **changed** files by only python or backend files using the `.py` extension. After this we should first check that the file indeed does have a corresponding backend unit test (e.g. `_test.py`) and run the test. We are going to run these in concurrency (5 tests at a time instead of 25, since higher concurrency threads can cause a decrease in performance and generally if tests are short enough they won't be canceled, so we can leave any "extreme" long backend tests to be skipped).

Below are the steps we would take to implement the above solution:

1. LONG TEST REPORTER

- Since the times are already output by the `run_backend_tests` script, we can upload these as a github artifact and parse the times from the output. We can do this by outputting the times into a file after running the script (maybe have a flag called `-generate_time_report`, similar to `-generate_coverage_report`) and uploading them as a github artifact for each backend test shard in the backend test workflow.
- After that we can get the github artifacts as an extra step inside the workflow (similar to check backend coverage) and then run a script called `long_backend_tests_reporter.py`. This script will **simply combine the github artifact files from each shard into a list of dictionaries (each containing a test name and its corresponding time)**. If there are any backend unit tests which are long (e.g. more than 150 seconds), we will output a message in the CI log.
- The log that will be outputted by the CI should have info on which backend tests were long and should guide a developer on how to fix it through a wiki page. Below is an example of what the log message will look like:

The following backend test suites ran for a long time:

- `{{backend_test_1}}`: took `{{backend_test_1_time}}` seconds
- `{{backend_test_2}}`: took `{{backend_test_2_time}}` seconds
- ...
- `{{backend_test_n}}`: took `{{backend_test_n_time}}` seconds

We want to try and reduce each backend test suite runtime to less than 150 seconds, so that the pre-push hook runs fast and we use up less GitHub CI resources. If you'd like to help with this, please follow the guidelines at [wiki page DEF](#).

- We should also have a way to sort tests in the pre-push hook so this long backend test reporter should also output a sorted list of backend tests and their times. Each backend test shard will output a file called `backend_test_shard_X_times.txt` and upload the file as an artifact that looks like this:

```
{{backend_test_1}}, {{backend_1_test_time}}  
{{backend_test_1}}, {{backend_2_test_time}}  
...  
{{backend_test_n}}, {{backend_n_test_time}}
```

We will combine all of the backend test shards' times files (which are imported by Github and locally accessible), and then we will parse this list by splitting by new line to get each test dictionary, then splitting by a comma to get the test's name and its time. Using this information we will format the list to look like this [log info](#) and output it into the CI as a log (just printing).

Alongside this we should also use the combined backend test shards' times to output an artifact containing the sorted list of tests and their times, which will be used later in the pre-push hook.

2. ADDING TO PRE PUSH HOOK

- In the `pre_push_hook` script we should **add** a new function to check if there are **changed** backend files by checking if any of the changed files ends with a `.py` (python extension).
- If there are backend files with changes we would first detect if that file has a corresponding test file and if it does we would pass them to `run_backend_tests.py` (we should alter the argument `--test_target` inside `run_backend_tests.py` to **take more tests by allowing it to take in a list of space delimited files to run in concurrency**). The `run_backend_tests` will run only the tests that are provided in the argument. We will run tests which are shorter first by downloading the artifact of sorted test times from the Github CI shown in this [step](#).
- There may still be some cases where the pre-push hook timeouts at the SSH limit of 5 minutes, so we should have a runtime check that cancels any running backend unit tests and gives a clear output that shows that their tests didn't run. Below is what the comment will look like:

The pre-push hook timed out since it exceeded 5 minutes, the following backend unit tests were canceled:

- `{{backend_test_1}}: {{backend_test_1_command_to_run}}`
- `{{backend_test_2}}: {{backend_test_2_command_to_run}}`
- ...
- `{{backend_test_n}}: {{backend_test_n_command_to_run}}`

NOTE: There may be a discrepancy between runtimes of backend unit tests on the actual github CI (which we are using to report long tests) and developer's work machines. The way I thought about this is that the pre-push hook's purpose in this case **is to get some immediate feedback on code** (through unit tests and linters). Therefore one of the **primary goals of the pre-push hook is for it not to run for too long**.

Therefore, I can use the Github CI as a **baseline** to sort which tests should run first since the order of shortest to longest should be fairly the same on the Github CI and locally and this allows us to fit as many backend tests since the shortest will run first and any long ones that we don't necessarily want to run on the pre-push hook will run last.

4. Ensure that all the lint checks are run for only changed files in the pre-push hooks.

From looking at the existing code right now, this seems to be already implemented for lint checks and `run_lint_checks` script is run inside the *pre-push hook* for files that are **changed** and have an ACRMT status (added, changed, copied, modified, renamed). For checks like Typescript and MYPY we should continue to run this for all files since running them partially might result in undetected errors as types can be spread across multiple files. For example: File A uses File B, we change File B, File A is not updated (if we run a type check here, it will only result in type checking File B, while File A might have an undetected error if it uses changes in File B). Most of the lint checks in `run_lint_checks` run only on specific files such as the *python_linter*. Other lint checks such as the codeowner lint checks rarely run since it only checks these files when the file extension isn't *js*, *ts*, *html*, *css*, or *python*.

Analyze and reduce backend unit tests which take a long time:

1. Analyze backend unit tests and reduce the tests that take a long time (where needed and safely).

Problem: Many backend unit tests at the moment take a **long** time when they don't necessarily need to (essentially many parts of backend unit tests that take long can be **mocked**).

PROPOSED SOLUTION:

1. First get a list of backend unit tests and their times (this can be **obtained** from the "long" backend reporter).
2. Using this list of backend unit tests, go down the list of backend unit tests that can be considered long and **analyze** the code to see what can be **refactored to reduce the time the test takes** (while also ensuring that the test functions like it needs to). Long tests will usually take long since it makes **expensive** calls to the backend such as checking **many files or database/api calls**, to fix this we should **mock** in situations where tests can be **mocked** (while still ensuring they still test what they are supposed to).

EXAMPLES:

EXAMPLE OPTIMIZING *scripts.check_backend_associated_test_file_test* (~244.8 SECONDS):

Here I believe is a **great example to implement mocking**, from looking at the profile stats each test case spends **around 47 seconds on the function *check_if_path_ignored*** which is a function which simply calls ***git check-ignore***, which checks if a certain path is ignored by **git**. I feel like this isn't generally needed in the scope of this backend unit test as we are **primarily trying to test if the script can check for associated backend test files correctly**. Note that the time complexity of the *check_if_path_ignored* function primarily **comes from checking the whole codebase as we walk through all files in the topmost path level, calling this function hundreds of times** (e.g. the whole Oppia codebase). To alleviate this time complexity I suggest that we **mock the actual *TOPMOST_LEVEL_PATH* constant to only scope into the temp directory that we initialize when doing the test cases**. I implemented this in my own repository by simply making a swap with the *TOPMOST_LEVEL_PATH* constant to **the relative path of the temp directory**. Doing these changes, the time of the *scripts.check_backend_associated_test_file_test* goes from ~244.8 seconds to ~0.9 seconds. Note that this solution **can be extended similarly to other tests which run this *git check-ignore* on the whole codebase** (e.g. *scripts.codeowner_linter_test*, which time complexity once again comes mainly from running *git check-ignore* on ~6000 files).

GENERAL APPROACH TO OPTIMIZING TESTS:

1. Use *cProfile* to profile a test, which shows how long each function call takes (and how every function's time builds on a test case).
2. With this information, find what calls take the longest and see how we can optimize it. One method to do this is by mocking, we can mock certain database calls or network calls to shorten

test cases. Another general thing I thought about was running test cases in parallel, which I don't know if we do now, or migrating to another test framework like pytest.

Stay with Unittest	Migrate to PyTest
Unittest is more of an old testing framework and is missing some core features, but it is easier to learn and easier to understand overall as a beginner. It doesn't have any built in system to run test cases in parallel, but it would be hard to change all the existing code to PyTest.	PyTest is a testing framework which is well supported and more "complete" as it includes many useful tools like parameterization, fixtures and also comes with a parallel test case running out of the box. However, migrating to PyTest from Unittest is difficult and complex as their API is different and PyTest also takes more time to learn.

Since it is quite complex to migrate to PyTest from Unittest, it would be better to stick with Unittest for now.

Here is a google sheet containing the test times on the Github CI and general time complexity/optimizations for the long running tests: [📊 Oppia Backend Tests](#)

Generating a dependency graph and using it to run partial tests on pull request CI.

1. Create a script that generates a dependency graph which can be used to find what Angular root modules changed files are a part of.

Problem: We need a way to see what Angular root modules are **changed** by certain files so we can use them to determine **which acceptance/lighthouse tests to run**.

Angular Root Module: Here when I mention a root module I am talking about a Angular page module which usually ends with `.module.ts` and AngularJS modules which end with `.import.ts` and are referenced inside `webpack` (since the codebase is currently split between AngularJS and Angular), **high level page modules**.

PROPOSED SOLUTION:

I suggest we implement this in *Typescript/JavaScript* as it will allow us to get imports of frontend files quite easily using Abstract Syntax Trees.

Before trying to scrape for declarations at all, we should do an initial sweep which will collect various information on the frontend files in the codebase. One major thing that we will check for first is if the file is a component, module, directive, or pipe (using the `@Component`, `@Module`, `@Directive`, `@Pipe` decorators). We should also get the selector argument that is passed through (e.g. `oppia-thumbnail-uploader`. Below is an example of what this presweep information object would look like ("..." means deeply nested):


```

{
  'core/templates/.../thumbnail-uploader.component.ts': {
    type: 'component',
    selector: 'oppia-thumbnail-uploader',
  },
  'core/templates/.../apply-validation.directive.ts': {
    type: 'directive',
    selector: 'applyValidation'
  },
  'core/templates/.../teach-page.module.ts': {
    type: 'module'
  }
}

```

LOGIC FOR TYPESCRIPT/JAVASCRIPT FILES:

To get all of the imports of typescript/javascript files we can use the library “**typescript**” which is already installed on the Oppia codebase. We should create a new TSConfig with the files that we want to “scrape” for declarations. Then inside our script we will create a new TS compiler host using the `ts.createCompilerHost` function. Using this TS host, we have access to a variety of useful functions in order to extract the imports of a file. In order to do this, we should first **loop through every file path inside the host files** and get the source file of the file path that is passed in through `ts.getSourceFile`, which returns a “Node” which is basically an AST (Abstract Syntax Tree) of the TypeScript or JavaScript file. This allows us to **loop through each of the children nodes which are syntax groups** using `ts.forEachChild` and check if that syntax node is a corresponding import using `ts.isImportDeclaration`. Using this, we can **easily get all of the imports that are defined in a JS/TS file** and we can also exclude certain files like libs using the TSConfig.

LOGIC FOR EXTRACTING NODES FROM TYPESCRIPT/JAVASCRIPT FILES:

1. Extracting required information for generic imports:

- path = undefined;
- if `ts.isImportDeclaration(node)` then
 - `moduleSpecifier = node.moduleSpecifier.getText();`
 - if moduleSpecifier starts with ‘.’ then
 - add the dependency with the relative path to root; **stop**
 - if `fs.existsSync(node_modules/{{first part of path}})` then return undefined;
 - look for path by alias under ts config;
 - if the alias exists then add the dependency with the relative path to root;
 - path = path from logic above...
- return path;

2. Extracting required information from Component/Directive/Module/Pipe

- `angularInformation = [];`
- if not `ts.isClassDeclaration(node)` **stop**
- if there are **no decorators** then **stop**
- for decorator in decorators:

- if is not a call expression then **stop**
 - decoratorText = decorator.getText()
 - if decoratorText is not in ['Component', 'Directive', 'NgModule', 'Pipe'] or decoratorText is not in IGNORED_DECORATORS then **throws an error**.
 - if decoratorText is 'Component' or 'Directive' then get the selector and templateUrl argument property;
 - if decoratorText is 'Pipe' then get the name argument property;
 - angularInformation.push(angular information from above logic...);
- return angularInformation;

LOGIC FOR PARSING AND FINDING HTML FILE DATA:

First off I will use a library that is called cheerio which can be installed using yarn, this is basically the *DOMParser* class or JQuery but it can be used through NodeJS instead of the browser. This is the simplest way to find dependencies in HTML files since the selectors that come from components and directives are "selectors", like for components it is a tag element and for directives it is attributes. Therefore to find out if a HTML file uses a certain component or directive we can basically find it using the corresponding selector with cheerio. We also need to scrape for directives, pipes and components in HTML files. In order to scrape for these we can use the pre-swept Angular information of each file. Simply filter for only files that are directives/components/pipes and use cheerio to search for its specific selector in the HTML file. For @load functions we can simply check the element's text using cheerio and if it includes any @load we can find the import module between the function's parentheses. Regarding CSS files, most are imported through components and will be captured when we [look](#) for import nodes. For the special cases where there are CSS files in the HTML files, we can check if the element's tag is a link and resolve the href as a dependency.

Pseudo code for extracting info from html files:

- filesAngularInformations {
 [fileName as key]: [list of angular class informations in file];
 } = this.fileAngularInformations <- depends on this which is scraped beforehand.
- dependencies = [];
- for htmlFile in htmlFiles:
 - \$ = cheerio.load(html);
 - For each element in \$:
 - Remove any [] or () that are wrapped around attributes, which are used for binding attributes in Angular. We do this since later on when we look for specific directive attributes these will prevent us from selecting attributes using the [attribute] selector.
 - Check if the text of the element includes @load and if it does substring the load function to its () and resolve the module import inside.
 - if element is link then resolve(element href) and dependencies.push(resolved);
 - loop through every file info entry in the preswept angular information mapping:
 - if fileAngularInformations[file].type is 'component' or 'directive':
 - if \$(fileAngularInformations[file].selector) then dependencies.push(file);
 - ~~using the angular information if it's a pipe then search the attributes and the text of every element for the | text and if it is present check if the text/attribute value also includes the pipe's selector.~~
 - if fileAngularInformations[file].type is 'pipe':

- for element in \$('*'):
 - if element.attributes contains | and fileAngularInformations[file].selector then:
 - dependencies.push(file);
 - if element.text contains | and fileAngularInformations[file].selector then:
 - dependencies.push(file);
- return dependencies;

General computations:

For all files **we have to do some general computation to ensure that every file that is declared as a declaration is absolute and not relative**. Many imports **use relative paths** like starting with './', '../' or TSConfig paths like 'domain/...', 'pages/...'. However, in order to **backtrack later with precision**, we will have to **have absolute paths** so 'domain/...' should be 'core/templates/domain/...', **where all files should be a reference from the codebase's root path**. In order to do this, whenever we get a file path **we should check if there are any relative paths** (e.g. starting with .) and if there are we should **make them absolute by concatenating the path with the current directory's path from the root**. For TSConfig paths like 'domain/...' or 'pages/...', we can **check if the first path is apart of the TSConfig paths option** (which is parsed for when we create the *compilerHost*) and if it is **we can just substitute the corresponding path** (e.g. 'domain/...' will correspond to 'core/templates/domain/...' according to the TSConfig).

Other Edge Case Files:

While the above contains the logic for extracting TypeScript/JavaScript and HTML files which covers a lot of the main files in the codebase, there are also some edge case files like README.md. When changing a file like this, we should run no tests so in order to do this we can just add the '.md' extension as an included extension when gathering the files using `host.readDirectory()`, we can also do this similarly with files like CODEOWNERS. The extensions we will need to use to generate the dependency graph are: [.html, .ts, .js, .md, .css, CODEOWNERS, AUTHORS, CONTRIBUTORS].

We will only scrape files with the above file types. Within these filetypes, there are also several files that we want to exclude as we don't want to scrape their dependencies and in cases that these files are changed, we should run all tests. By default we should exclude all files in .gitignore and have some additional custom exclusions. Below is a list of the exclusions during this step: [

```
...all files in .gitignore,
'types',
'typings',
'scripts',
'core/tests/test-dependencies'
'core/templates/services/UpgradedServices.ts',
'core/templates/services/angular-services.index.ts',
'core/tests/build_sources',
'core/tests/data',
'core/tests/load_tests',
'core/tests/release_sources',
'core/tests/services_sources',
]
```

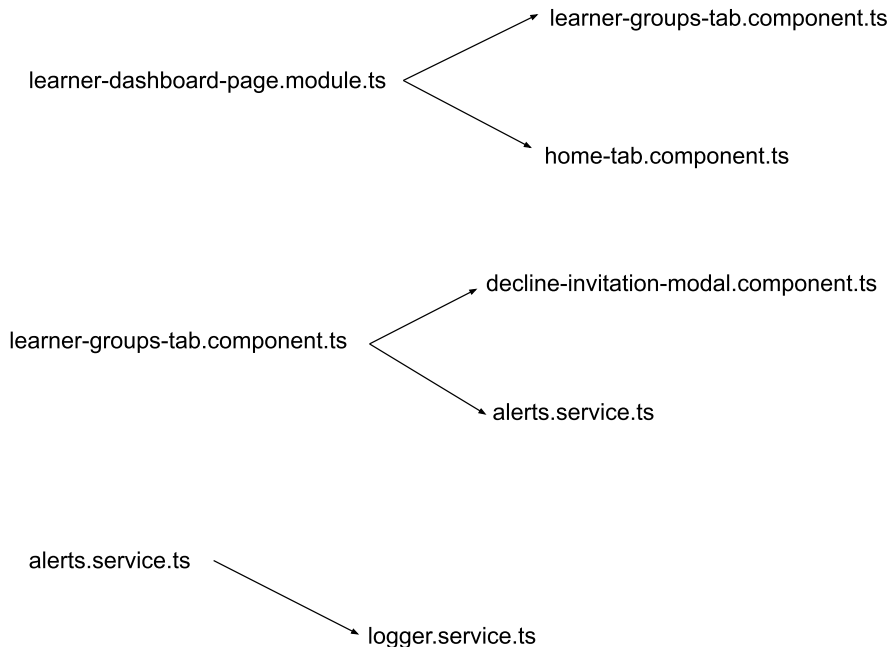
EXPECTED CHANGES:

FILE: *dependency-generator.ts*

- `host = ts.createCompilerHost(TSConfig);`
- `jsAndTsFiles = host.readDirectory(root, ['.html'], EXCLUSIONS, []);`
- `filesAngularInformations {`
 `[fileName as key]: [list of angular class informations in file];`
 `} = {};`
- `depedencyGraphDict {`
 `[fileName as key]: [list of dependencies];`
 `} = {};`
- `for file in jsAndTsFiles:`
 - `filesAngularInformations[file] = logic for extracting angular information like module, component, pipe, directive and add it to a mapping`
- `for file in host.readDirectory(root, ['.html'], EXCLUSIONS, []):`
 - `fileContent = fs.readFileSync(file, 'utf8');`
 - `document = cheerio.load(fileContent);`
 - `depedencyGraphDict[file] = extract component tags directives and pipes`
- `for file in jsAndTsFiles:`
 - `sourceFile = host.getSourceFile(file, ts.ScriptTarget.ES2020);`
 - `sourceFile.forEachChild(node =>`
 - `if ts.isImportDeclaration(node) then`
 - `if type is a module only extract module imports;`
 - `depedencyGraphDict[file].push(logic for extracting imports...)`
- `fs.writeFileSync(depedencyGraphDict, 'dependency-graph.json');`

MAP THE FILES BACK TO THE ROOT MODULES THROUGH THEIR DECLARATIONS:

Now once we have all these declarations, we should associate every file with its corresponding root modules. In order to do this we should loop through all of the files in the codebase once again and **use the declaration list we made above to recursively find the file's root module(s)**. Let's say we are analyzing a component file, we should **find all components/modules that declare this component and declare those as new stems**. We recursively do this until all of the stems reach a root module (which should be a module which **isn't referenced anywhere as a declaration**), and these modules/files that are reached should be considered the root module(s) that a file changes.



The above diagram is a visual representation of the list I was talking about and each class name and its file is associated with a list of declarations it has (which once again is created using some node parser algorithm). Now once we have a list of all the files and classes alongside their declarations like above we would go **through all the files again and associate them with their root module(s)**. For the purpose of simplicity I have made the above modules and components quite **simplified** (the actual tree will be much more complex with multi dependency trees). Let's say we want to find the root module of the *logger.service.ts* file, to do this we would **first see if any files declare this service** and in this case, there is one, **that being the *alerts.service.ts***. We would declare *alerts.service.ts* as **our new "stem"** and **see if any files declare *alerts.service.ts***. In this case there is one, that being the *learner-groups-tab.component.ts* file. Once again, we look to see if any files declare this component and in this case there is one, that being the *learner-dashboard-page.module.ts* file. Once we **look up the declaration tree, and in this case we can see that no more files declare this module** and we can **safely assume that this module is the one and only root module of *logger.service.ts* file**. We would repeat this step for all files and get a list of the root module(s) that the file is associated with in some database file like *JSON*.

To **make this process faster** we can also **cache paths of declarations** that are common between different files. When we run the **algorithm to go up the declaration tree**, we will **first see if the stem that we declared already has its root modules defined** (e.g. are already ran through the algorithm), if it does then we can automatically **assume that the path with that particular stem will end up with those root modules**. In order to cache **more efficiently** we can also **cache and define the root modules of a stem** everytime we get a **stem without any cached path**. For instance *SharedModule* is a widely used module and **therefore will end up as a stem of many files**. However this algorithm **will actually not run through the *SharedModule* path more than once** since we will **cache the declaration path and declare it** when we get it for the first time (essentially making **any shared stems be declared first and cached to be used for other files**). After this algorithm is run, there will be a *JSON* file that will **contain the list of all the frontend files (TS/JS/HTML/CSS) and their corresponding root modules**. Below is an example of what a few of the lines would look like (**note that all the paths would be absolute, so *learner-dashboard-page.module.ts* below would be *core/templates/pages/learner-dashboard-page/learner-dashboard-page.module.ts***):

```

"community-lesson-tab.component.html": ["learner-dashboard-page.module.ts"],
"merge-skill-modal.component.ts": [
  "collection-editor-page.module.ts",
  "topics-and-skills-dashboard-page.module.ts",
  "topic-editor-page.module.ts"
],
"editability.service.ts": [
  "topic-editor-page.module.ts",
  "skill-editor-page.module.ts"
]

```

EDGE CASES WHEN BACKTRACKING FILES:

One main problem when backtracking files is that many shared modules hold many imports like *shared-component.module.ts* or *base.module.ts*. When backtracking this will result in many files ending up at these modules, then eventually resolving to nearly all root page modules since most pages rely on these modules. The main way to remedy this is like so:

1. We will first backtrack files while ignoring modules, so all files will ignore a module if it comes across one when backtracking. When the references reach 0, essentially we reach a "root" file which in this case won't actually be a page module since we ignored all modules.
2. From there we will not ignore modules and backtrack from that "root" file to its corresponding module, we should also check if any module on the way is a page module and if it is, just declare the page module as the final root (this is to remedy large import modules like *app.routing.module.ts*).
3. Now there are some cases where we actually want to backtrack to some shared module (this is usually the case when the actual file is a module and when we change a module we want any tests referencing this module to run). Therefore we will also have a revision stage of generating this dependency graph, where we loop through each key in the dependency graph and if there aren't any page modules present in its "root" files, then we will backtrack while not ignoring modules.
4. There will always be some holes as the whole codebase is quite complex, but doing the above will ensure that we have a good mapping which ensures that tests aren't run when unnecessary, while also not causing lots of flakiness.

Another general problem when backtracking files is that some files will end up at misc files that aren't linked to any tests such as *src/main.ts*, *src/index.html* or *core/templates/pages/oppia-root/index.ts*. While some of these files are remnants of AngularJS and will get removed when we completely migrate, there are some cases where files are loaded outside of modules. This will cause some files like *index.html*, where someone could make a breaking change, but it doesn't actually run any tests. Furthermore, there are also some "tricky" pages such as the error page and maintenance page. These pages don't have any real route as the maintenance page is enabled through a flag and the error page occurs when a status error shows up. Therefore while some files will map to these modules, these modules/files don't actually map to any test, which causes problems. To solve this problem, we can have a list of root files which when changed, will run all tests. When the partial test python script receives a list of changes, we will cross check it with this list and if the file is in there, it will output all tests.

Testing Strategy:

There will be two main strategies that will be used to test the generated dependency graph.

1. Look at opened/closed frontend PRs and cross-check the files changed with the dependency graph and see if what modules being changed makes sense.
2. Manually trace files to their page modules and compare that with the dependency graph.
3. Take a look at the root modules and see if any of the root modules don't match up or don't map to anything.

2. Maintain this mapping of Angular modules and use it to determine which acceptance/lighthouse tests they affect and run them only on the PR CI (for only frontend affecting PRs).

Problem: We need a way to maintain a mapping of acceptance/lighthouse tests and their corresponding Angular files to modules and use it to determine which acceptance/lighthouse tests to run on PRs that only affect frontend files.

PROPOSED SOLUTION:

AUTOMATE & MAINTAIN A LIST OF ANGULAR MODULES TO TESTS:

Firstly we should maintain some list of Angular modules to their corresponding tests. As noted this should be done in some automated way and there should be some easy way to update this list when needed. **We should first create a utility file called `test-to-angular-modules-matcher.ts` inside that `main core/tests` folder that can be used by any test easily.** The purpose of this utility file is **to take URLs in through a function `registerUrl()` and simply build a list of the modules that the URLs correspond to** and therefore as the test continues, this list completes with the **root modules that a test is affected by**.

1. **We first need a mapping of Angular routes to Angular modules** so that we can execute a match on a specific route and a url coming into `registerUrl`. This mapping will look like this:

```
{ path: '', pathMatch: 'full' } =>
'core/templates/pages/splash-page/splash-page.module.ts',
{ path: 'collection/:collection_id', pathMatch: 'full' } =>
'core/templates/pages/collection-player-page/collection-player-page.module.ts',
{ path: 'voiceover-admin', pathMatch: undefined } =>
'core/templates/pages/voiceover-admin-page/voiceover-admin-page.module.ts',
{ path: 'blog/:blog_post_url_fragment', pathMatch: 'full' } =>
'core/templates/pages/blog-post-page/blog-post-page.module.ts'
```

Above is a subset of the mapping of where the key is an Angular route and the value is an Angular module's file path.

2. We should have some way to automate this, but at the moment, there is **no definite list of URLs** and therefore no straightforward way to maintain this list through automation since the Oppia codebase is currently migrating from AngularJS to Angular (resulting in some routes being in the backend and chunked by the webpack and some routes being inside `app.routing.module.ts`). In the case that we do finish the migration from AngularJS to Angular, **one simple way to automate this process would be to scrape `app.routing.module.ts` and get the URLs & imports that are defined in that file.** One way around this is to have one automatically mapped URLs to Angular modules and have a manual mapping for URLs to AngularJS modules. It would be best to go with

this approach especially since AngularJS modules aren't going to change and we can automatically scrape any new URLs to modules. Here is a prototype for the above scraping: [link](#).

3. When the `registerUrl()` method of the utility file is called with a URL (these URLs will come from the tests whenever a page is visited), we need to **match that URL to a specific module** and add that module to a list. One way to know if we have mapped all URLs is to **give an error whenever a URL which is passed through doesn't have a corresponding module mapped** (only for URLs which **originate from localhost**).

For creating this utility file, **it will be quite straightforward once we have a mapping of URLs to modules** and the steps for creating this matcher would be like so:

1. Create a new file in the tests directory in JavaScript called *test-to-angular-modules-matcher.ts*.
2. Have some sort of local array which we can append modules that we match with.
3. Create a function *registerUrl* which **takes in a URL and matches it to a module in the URL-to-module mapping**. In order to actually match URLs called through the function *registerUrl* with the URL to module mapping is by using the `@angular/router` library. This library will make it simple to take in the URLs from *app.routing.module.ts* (which use the router library to actually create the frontend router) and match it with a URL called from *registerUrl()* and if there is a **matching URL, append it to the local array if it isn't already present**.
4. Now that we have this local array, which is simply an array of modules like *topics-and-skills-dashboard-page.module.ts* or *creator-dashboard-page.module.ts*, **we should add a function to compare the array with the preexisting "golden" text file and show the diff in the console**. This function will create and save a text file under the path *test-module-mappings/{{test_type}}/{{test_name}}-generated.text* to the local file system, which will contain the newly generated modules array as strings with newlines and this function will also compare it to the existing *test-module-mappings/{{test_type}}/{{test_name}}.txt* and if there are any differences, error the script and output the diff. The workflow will then upload this file as an artifact so the developer can easily change the "golden" text file to the correct one, if needed.

In order to pass the URLs that a test visits into this utility file while running tests, we can alter the lighthouse/acceptance tests to do this. However, there are **many different test types** (e.g. Lighthouse/Acceptance) which use various testing **engines like Webdriverio and Puppeteer**. Therefore in order to input the URLs that a test visits through the function *registerUrl* inside the utility file mentioned above, we will have to alter them differently like so:

- **Lighthouse Tests (Puppeteer):** Lighthouse tests are slightly different as they use puppeteer instead of Webdriverio, but the process remains mostly the same. For puppeteer tests we can use the *page.on("framenavigated")* event which **detects when a user visits a new page and we can get the url and call the *registerUrl* function**.
- **Acceptance Tests (Puppeteer):** This process will be exactly the same as the approach for lighthouse tests except we will incorporate it into acceptance tests. Therefore, the syntax for getting the new page URLs should be exactly the same since both tests use puppeteer.

For all of the tests we will have a corresponding text file under the path *test-module-mappings/{{test_type}}/{{test_name}}.txt* which stores an array of all the modules that the test depends on. This text file will be the source-of-truth for determining which tests to run in the PR CI. This text file might become outdated, so the best way to remedy this would be to **compare the text file obtained from running the utility file with the pre existing "golden" text file after running any test**. When a test runs in the PR CI, the tests' text files that will be used to determine which tests to run are the PR's

feature branch (the one that the developer checks in for the PR and not the main Oppia develop branch). Therefore, when a test runs on the PR CI and it generates the JSON file mentioned above (will be a file called *test-module-mappings/{{test_type}}/{{test_name}}-generated.txt*), we should compare the output with the JSON file that is already locally existing (*test-module-mappings/{{test_type}}/{{test_name}}.txt*) and see if there are any changes. If there are, we should error the test and show what exactly needs to be updated so the developer can manually update the JSON file on their repository.

Every test suite in the *check_ci_test_suites_to_run* has a suite name and its module path, which simply points to the file that is used for that test. Whenever a test or a test's dependency is modified, then the root module that is mapped in the dependency graph will be the module path. So in the script when we collect the various ci test suites to run, we will also check if any of the modified modules correspond with a test's module path and if it does we will add that test suite to the test suites that will run. When we remove a test, this test will just simply not run since it is no longer present in the *check_ci_test_suites_to_run* script (which should be enforced by cross checking the test directories and the constants present in the script [here](#)).

In the case that the developer removes a test, but forgets to remove it from the config files, then they will be notified by the ci when the cross check [here](#) happens. However, this would be quite cumbersome as if a developer decides to remove a test or adds a test and pushes it, the CI would throw an error, preventing their PR from progressing. To prevent this we can run the cross check [script](#) every time a test change is pushed.

LIGHTHOUSE TESTS STRUCTURE CHANGE:

We also want to run lighthouse tests dependent on what modules are modified and the current structure does not allow us to do so. Therefore we will have to make a few changes to the overall structure of the lighthouse testing infrastructure. Below are all of the changes we will need to make:

1. Introduce a new *-pages* argument to the *run_lighthouse_tests* script which takes in a delimited list of page names. These page names will each correspond to a lighthouse URL to test and this mapping will be stored in *.lighthouserc-base.js*. At runtime we will pass the *pages* arg as an environment variable and map it to what lighthouse URLs we should run and this will be the URLs that are used in the performance/accessibility shards.
2. Our new *check_ci_test_suites_to_run* will also contain a mapping that maps modules to lighthouse page names. The lighthouse accessibility and performance will generate their "golden" files from all URLs in their corresponding shards in *.lighthouserc-base.js*. So when any module that runs a shard is changed, that particular shard will run. However, alongside this, the *check_ci_test_suites_to_run* script will also map the modified modules to lighthouse page names using the mapping mentioned above. From there it will pass this to the workflow where the workflow will pass it as the *-pages* argument.
3. Therefore, whenever a module is modified, if a shard consists of this module, it will run and dynamically generate the URLs to run on that shard. If there is no *-pages* argument then it will simply run all URLs.

EDGE CASES:

At the moment, **the above algorithm will match all URLs for tests, so some tests will run even when it is not really needed**. For example, let's say we alter the login module (some file connecting to it), **all acceptance/lighthouse tests which include the login page will run**, even when **it doesn't really need to**

run. This is one of the **downfalls of trying to completely automate this process**, some **edge cases** like this will **slip** through. Here, I will describe a few solutions to this problem:

Create an exclusion mapping for certain URLs (like the login URL), which when the URL is passed through we will exclude it from the module to test mapping and instead manually map the login Z~module to a test.	Do time analysis on tests, when a URL is passed through to the utility file, we will time how much a test spends on that page and at the end of the test we will combine the times to get the total amount of time the test spent on a specific module. From there we can exclude certain modules which were not used a lot during the test (through time analysis, e.g. only include modules which a test spends a majority of time on).
An exclusion mapping will allow us to be more precise, but it will require more manual alteration.	The time analysis solution will allow much more automation, but could cause more flakiness.

EXPECTED CHANGES:

- `core/tests/angular-route-to-module-generator.ts`
 - This file simply contains a class that allows mainly the *test-to-angular-modules-matcher* to generate a mapping of Angular routes to modules by scraping the `app.routing.module.ts` and also adding on manual routes mainly from webpack.
- `core/tests/test-to-angular-modules-matcher.ts`
 - `setGoldenFilePath(goldenFilePath)`
 - Sets the golden file path of the script, so it can use it to find inclusions and in the end compare and output a text file containing the tests' modules.
 - `matchUrl(url: string, route: Route)`
 - Matches a specific Angular route to a url by using the route's params and url segments.
 - `registerUrl(url):`
 - Matches a URL to a module using the `matchUrl` function on all routes from the *angular-route-to-module-generator* and stores that module in an array if it is not excluded. If it is excluded, check if the golden file is an inclusion of that specific exclusion.
 - `compareAndOutputModules():`
 - Uses the `goldenFilePath` to find which file to compare with and uses the local root modules array generated from `registerUrl` to compare and output the diff in the console along with outputting a generated text file containing all of the modules.
- `core/tests/test-module-mappings/{{test_type}}/{{test_name}}.txt`
 - This text file contains the specific test to module mapping. Every test will have one of these files and they essentially contain a list of modules that the test depends on.

TESTING STRATEGY:

- First we will run on all tests, so we will have a fully populated `core/tests/test-module-mappings/{{test_type}}/{{test_name}}.txt`.

- Scan all of the different golden files and make sure that it aligns the pages that the test tests. Make sure that if any is missing or any are extra, that the specific test errors and shows a list of what modules are missing/extra.

ALTER THE EXISTING WORKFLOW TO RUN PARTIAL TESTS WHEN NEEDED:

Now we should create a new script and within this script the following functionality will be implemented:

- The script will first use **git diff between branches to check the difference between the develop branch and the feature branch**. We should first exclude all '.py' files and automatically run all tests if there is any '.py' file included. From there we will check if any of the files are not included in the dependency graph, in this case it is considered an exclusion and we should run all tests. However, if we do find the file in the dependency graph then we will use those modules to find which tests to run. If there are no modules corresponding with that file (in cases where it is a file like README.md), then no tests will run. The file extensions and exclusions for the dependency graph and therefore this step are listed in the [dependency graph section](#).
- First off if the **changes include non-frontend** files like .py, **we should run all suites so the script should in this case just output all suites**. If there are no backend changes, we will look up the files **inside the generated dependency file** (which in this case will return the root page modules). We will combine all of the different test files under the specific test type directory and look for the `{{test_name}}-test-module-mapping.txt` into a dictionary and find which tests depend on the root page modules that we got from the files inside the generated dependency file. From there we will output these tests and in the workflow run the tests depending on this output.
- We will also move all suite names to a new file since we no longer compute the matrix inside the workflow yaml file. Therefore, all test type configs will now go under `core/tests/ci-test-suite-configs/{{test_type}}.json` where it will contain all of the specific test type's suite names and the location of its module or script.
- There are two ways we can incorporate this into our workflow:

<p>We can use outputs and jobs to run a script and output the JSON for the subsequent tests (acceptance and lighthouse). This would be a job that is run at the start of running all of the tests and it would only be run once and instead of passing all the suites through to the matrix we should use this new job output instead.</p>	<p>Create a script that is run every time a suite is run to determine if that suite should be skipped or not.</p>
<p>This approach allows us to determine which tests to run before running the tests, so suites that don't need to run will be skipped, saving CI resources.</p>	<p>While this method does seem simpler it would introduce unnecessary overhead when running tests and it beats the purpose of making the CI faster since all tests would have to run some script which will take PR resources.</p>

CHECK THAT ALL E2E/ACCEPTANCE TESTS ARE CAPTURED IN CI:

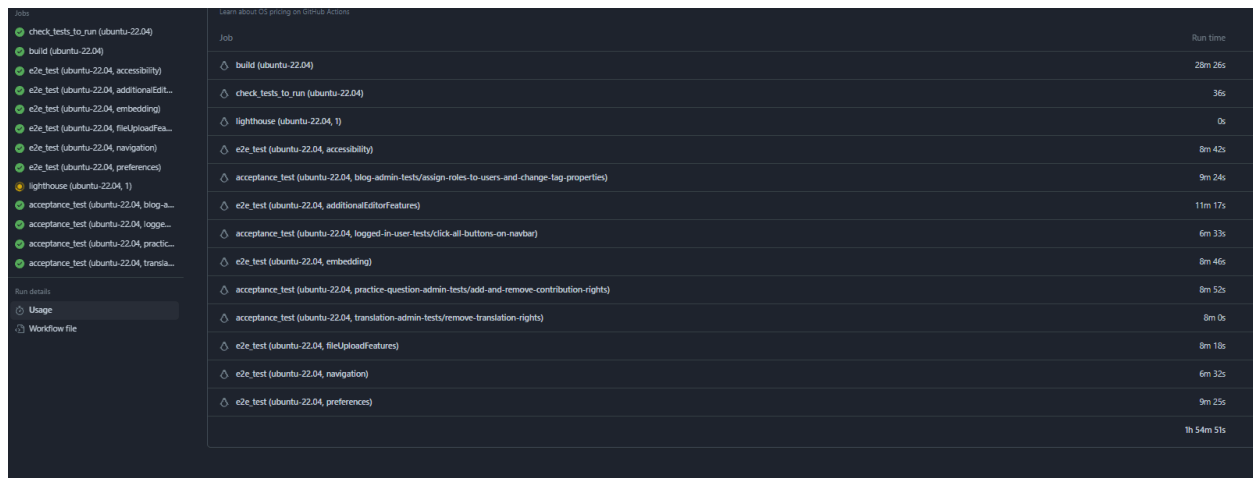
Currently, the codebase has a `check_e2e_tests_are_captured_in_ci` which simply cross checks the workflow file's matrix suites with the local e2e folder to see if all the e2e tests are captured in the workflow. However, given the above, we have to now take all of the suites in the workflow matrix to a new

script called `check_ci_test_suites_to_run`. Using the existing `check_e2e_tests_are_captured_in_ci`, we can simply alter it to instead use the `ALL_E2E_TEST_SUITES` array inside the partial test script, which should be simpler than parsing the workflow file. Similarly, we should implement something like this for acceptance tests, get all tests captured in the `core/tests/puppeteer-acceptance-tests/spec` and compare this with the `ALL_ACCEPTANCE_TEST_SUITES`.

We should also run the above script in the pre push hook whenever a test file is added/removed since if a developer makes test changes, they will not get the error until the cross check happens in the CI.

PROOF OF CONCEPT:

In my own repository, I have made a proof of concept of the conditional suites running using a script and job outputs to determine which of the suites to run beforehand. In this script, I didn't make any "real" mapping but instead **the script just outputs some tests** to show that **job outputs can be used in matrices to render partial tests within the CI**. However, implementing the actual matcher will be quite straightforward since the data with files to modules to tests is already present. Here is the pull request that I made to show these changes: <https://github.com/jnvtnguyen/oppia/pull/17>.



The screenshot shows the 'Jobs' section of a GitHub Actions workflow run. The job 'check_ci_test_suites_to_run' is expanded, showing a matrix of test suites. The 'Run time' column indicates the duration of each test suite. The 'Run details' section on the left shows the workflow file and usage.

Job	Run time
build (ubuntu-22.04)	28m 26s
check_ci_test_suites_to_run (ubuntu-22.04)	36s
lighthouse (ubuntu-22.04, 1)	0s
e2e_test (ubuntu-22.04, accessibility)	8m 42s
acceptance_test (ubuntu-22.04, blog-admin-tests/assign-roles-to-users-and-change-tag-properties)	9m 24s
e2e_test (ubuntu-22.04, additionalEditorFeatures)	11m 17s
acceptance_test (ubuntu-22.04, logged-in-user-tests/click-all-buttons-on-navbar)	6m 33s
e2e_test (ubuntu-22.04, embedding)	8m 46s
acceptance_test (ubuntu-22.04, practice-question-admin-tests/add-and-remove-contribution-rights)	8m 52s
acceptance_test (ubuntu-22.04, translation-admin-tests/remove-translation-rights)	8m 0s
e2e_test (ubuntu-22.04, fileUploadFeatures)	8m 18s
e2e_test (ubuntu-22.04, navigation)	6m 32s
e2e_test (ubuntu-22.04, preferences)	9m 25s
	1h 54m 51s

Now that we have implemented the main method to run conditional suites based on file changes, we should ensure that **all tests are run in the merge queue** and that **core maintainers have an easy way to turn this feature (i.e. running partial tests on the PR CI) on or off**. To do this, we will set a variable `RUN_ALL_SUITES` that is only true if the event is in the merge queue (**e.g. merge_group event**) and some environment secret or variable like `RUN_SUITES_ONLY_ON_CHANGED_FILES` is false. If `RUN_ALL_SUITES` is true (**e.g. RUN_SUITES_ONLY_ON_CHANGED_FILES is false or the event is a merge queue event**), then the **script would output all suites** (essentially **running all suites on the workflow**). This allows for all tests regardless of changed files to run in the merge queue and it also allows core maintainers to turn off and on the functionality through `RUN_SUITES_ONLY_ON_CHANGED_FILES` just in case the merge queue becomes flaky from this change.

General Problems:

1. Measure impacts of pre-push script and CI changes.

Problem: To ensure that our solutions to the pre-push check and CI tests are actually improving our developer workflow, we need a way to measure the impact that our changes have.

GENERAL SOLUTION:

1. One simple way we can measure the change the pre-push checks is by asking developers, maybe through github discussions, if they like the improvements to the pre-push check or not and if it made their experience with pushing changes easier/faster.
2. A way to measure the impact on the CI pipeline would be to analyze the workflow runs and check the beforehand and afterhand times. We should also be able to observe that many PRs that have trivial frontend changes should only run a subset of lighthouse/acceptance tests, thus resulting in an overall dip in PR CI usage (thus allowing tests to be ran quicker as github runners are freed more often). There are a few ways we can do this, we can sample PRs which have backend changes (e.g. running all lighthouse/acceptance tests) and compare them with frontend PRs where only partial tests are run. We can also observe an overall decrease in github workflow usage (e.g. through git action usage summary). In order to observe impact on the backend test times, we can compare the different individual backend test times in the CI right now and compare them after we optimize tests.

STRATEGIES IN COLLECTING TIMES:

To calculate the impact before and after on the Github CI, we can collect two statistics, the total time and the concurrent time. The total time that a certain test type takes tells us how resources are being utilized and how often github workers can be freed. On the other hand, concurrent times are more important in terms of getting things done, as usually tests run in concurrency so the "maximum" time that any concurrent test takes is when the CI is complete for that specific task. Below are some of the times for backend tests and lighthouse/acceptance tests. Each test type records 10 different Github runs with the time of each concurrent task (for backend it would be the 5 shards and the coverage reporter), and adds them together. From there we divide by 10 to get the average maximum time. To get the concurrent time we get the max time of all concurrent tasks of each of the Github runs and average that.

BEFORE TIMES ON GITHUB CI:

BACKEND TESTS:

1. $17+24+16+20+18+5 \rightarrow 100$ minutes
 2. $17+23+18+20+15+5 \rightarrow 98$ minutes
 3. $18+24+17+20+17+5 \rightarrow 101$ minutes
 4. $17+27+19+19+16+5 \rightarrow 103$ minutes
 5. $17+24+20+20+16+5 \rightarrow 102$ minutes
 6. $17+24+19+19+18+5 \rightarrow 102$ minutes
 7. $17+24+20+20+17+5 \rightarrow 103$ minutes
 8. $17+24+19+21+17+5 \rightarrow 103$ minutes
 9. $17+24+18+20+16+5 \rightarrow 100$ minutes
 10. $17+26+18+21+17+6 \rightarrow 105$ minutes
- Average: 101.7 minutes

Average Concurrent Time: 24.4 minutes

LIGHTHOUSE ACCESSIBILITY TESTS:

1. 29+29=58 minutes
2. 28+28=56 minutes
3. 28+28=56 minutes
4. 28+29=57 minutes
5. 28+29=57 minutes
6. 29+29=58 minutes
7. 28+29=57 minutes
8. 28+29=57 minutes
9. 28+29=57 minutes
10. 28+29=57 minutes

Average: 57 minutes

Average Concurrent Time: 28.8 minutes

LIGHTHOUSE PERFORMANCE TESTS:

1. 27+28=55 minutes
2. 27+27=54 minutes
3. 27+28=55 minutes
4. 27+28=55 minutes
5. 28+28=56 minutes
6. 27+28=55 minutes
7. 28+28=56 minutes
8. 27+28=55 minutes
9. 27+29=56 minutes
10. 27+28=55 minutes

Average: 55.2 minutes

Average Concurrent Time: 28 minutes

ACCEPTANCE TESTS (ONLY CONCURRENT TIME):

1. 12 minutes
2. 12 minutes
3. 13 minutes
4. 12 minutes
5. 12 minutes
6. 13 minutes
7. 15 minutes
8. 12 minutes
9. 14 minutes
10. 24 minutes

Average Concurrent Time: 13.9 minutes

Third-Party Libraries

For this proposal, I used mainly libraries that were already installed on Oppia, notably **Webpack** and **TypeScript**. However to scrape the HTML pages I am using cheerio since *DOMParser* is not available on NodeJS since it is a browser API. I am also using ts-morph which allows easier AST manipulation.

Impact on Other Oppia Teams

This project should impact all Oppia developer teams since it alleviates a lot of PR CI usage (leading to better productivity as tests will be executed quicker) and it also helps a developer's workflow locally through faster pre-push hooks and better code quality assurance. Here when I talk about code quality assurance, I am talking about ensuring that developers get optimal feedback on their code through automated tests before actually pushing to their PR. This project seeks to do this by implementing backend unit tests into the pre push hook and a more timely pre push hook will lead developers less likely to force push or *-no-verify* their pushes.

Documentation changes

Regarding the documentation, there would be a few modifications we would have to make, notably:

- Change the running [frontend/backend](#) unit tests documentation to include the new `-files` arg, which allows developers to run multiple frontend/backend unit tests depending on the files they pass through.
- Update the [backend unit tests documentation](#) to include the new information about the reporter and what a developer should do if the CI complains about long tests.
- Update the tests documentation for [lighthouse/acceptance](#) to guide the developer to update the new test output script with new test names and their modules as we no longer put test suite definitions in `.github/workflows/e2e_lighthouse_performance_acceptance_tests.yml`.

Conflicts with other projects

Regarding conflicts with other projects, there may be a potential conflict with the acceptance tests project since adding new acceptance tests will need to be subsequently added to the mapping for tests to root modules. However, I plan to do this project in milestone 1 and it should be fairly simple for developers to update this list as there will be an error on the CI if there is a change to the mapping and they can see the difference between the "golden" JSON file and the new JSON file. To update it they can do it manually using this diff or they can simply run the test once to generate this mapping automatically.

List of Artifacts:

- `dependency-graph.json`: Stores the information about files except for backend files and the modules that they affect (This takes a short time to generate, this is also not stored permanently since it will be generated every time we run the CI). See <https://raw.githubusercontent.com/jnvtnguyen/oppia/gsoc-planning/dependency-graph.json> for an early prototype.
- `core/tests/url-to-module-mapping.json`: Stores the information about URLs and the modules that correspond with that URL This is generated automatically by scraping the `app.routing.module.ts` and combining it with a manual mapping of AngularJS modules since they won't change.
- `test-module-mappings/{{test_type}}/{{test_name}}.txt`: Stores the information about tests and what modules are used in that test (This takes a long time to generate since it has to run the whole test in order to generate).

COMMON ISSUES AND HOW TO HANDLE THEM:

The main problem that could show up when implementing **Milestone 1** is that subsets of tests might not run on PRs even when the changes should initiate them. This can cause frustration as developers expect

feedback on their PRs during the review phase, but as their PR starts to get merged it starts failing. Furthermore, this can also cause flaky code as a missing test means that it will only run once in the merge queue and in some cases the merge queue can pass “once” and later on it might not. Below is the steps that the developer workflow team and maintainers should follow to remedy the above issue:

1. First if you observe many PRs having the problem of passing while reviewing, but ultimately not in the merge queue then make a collective decision and ask a maintainer to disable the `RUN_SUITES_ONLY_ON_CHANGED_FILES` environment variable through the Github settings. This will make all tests run in the Github CI regardless of what files are changed.
2. The good thing is that this project is mainly modular and the most problems I expect will stem from the dependency graph. The dependency graph is quite large (~120000 lines) which will be quite hard to audit and in some cases a changed file might not connect to any test when it should. Disabling the `RUN_SUITES_ONLY_ON_CHANGED_FILES` will disable the subset test system and will allow developers to detect exactly where the issue is (most likely in the dependency graph). From there the developer workflow team should look at the PR that is failing in the merge queue and the files that are changed, try searching them up in the dependency graph and see why the files aren't getting resolved to tests. If the dependency graph looks fine, then look at the specific test that is failing and its “golden” file. This golden file will contain all the modules that should make the test run and if an obvious module of that PR is missing, then there is most likely a problem with the [test modules analyzer](#).

Overall, this system is quite complex and some problems might arise despite trying to cover all cases. Therefore the system has an easy off and on switch through the `RUN_SUITES_ONLY_ON_CHANGED_FILES` and turning off the subset test system, will allow relief to developers, allowing the developer workflow team to fix the problem following the steps above.

Milestone Table (include both PRs and other actions that need to be taken prior to launch)

Milestone 1

Key objective for this milestone:

- Allow the Github CI to run partial acceptance/lighthouse tests for frontend changes so developers can get faster feedback on the tests that matter, while also ensuring that all tests run in the merge queue.
- Reduce Github CI resource usage by running tests partially on frontend changes, which will allow more Github CI resources to be free and therefore more tests will be able to run.

For product demos we should have one after PR 3 and then after we ensure that the proposed solution works through Task 4's approach we can start to measure impact by sampling different PRs and their test times.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	Create a utility file which takes in a test's URLs and maps it to Angular root modules. This utility file should also create a url to module mapping by automatically scraping the <code>app.routing.module.ts</code> file for Angular declarations and manually adding all AngularJS mappings.	N/A	28 May	2 June


2	<p>Create a script which will output a dependency graph JSON file which contains files and the modules that depend on that file.</p> <p>TASKS:</p> <ul style="list-style-type: none"> • Create a typescript file whose purpose is to scrape frontend files for their declarations and associate them with Angular root modules. • Create an algorithm that scrapes frontend files for their declarations. • Create an algorithm which will backtracks the frontend files to Angular root modules using their declarations. • Create a python script which basically compiles the typescript file and runs it on node. 	1.1	5 June	18 June
3	<p>Have text files for each lighthouse/acceptance test that store the modules that the test relies on.</p> <p>TASKS:</p> <ul style="list-style-type: none"> • Create a system to handle edge cases, by creating a manual exclusion mapping. • Ensure that the above text files are maintained by adding a check on all tests which simply compares the mapping it gets at runtime and the preexisting text file mentioned above. 	1.1	8 June	18 June
4	<p>Create a python script which maps changed files in a PR to Angular root modules and then to tests, so that the github CI will run tests depending on the files changed. In order to do this we can have the above script in PR 1 run as a step beforehand and then use this script to map the changed files to modules then to tests using the existing maintained JSON file from PR 2.</p> <p>TASKS:</p> <ul style="list-style-type: none"> • Move all the test suites from the github workflow yaml config to new json files under core/tests/ci-test-suite-configs/{{test_type}}.json and add a README. • Create a python script that gets changed files in a PR and map it to Angular root modules then to tests. • In this script we should first get the 	1.1,1.2,1.3	18 June	24 June

	<p>changed files of a PR by running <i>git diff</i> on the PR branch and develop branch.</p> <ul style="list-style-type: none"> • Match the changed files to Angular root modules using the JSON file in PR 1. Then map those modules to tests using the JSON file from PR 2. • Have this script output the corresponding tests to the github environment through a job and alter the existing github workflow to run tests depending on the output of the job. 			
6	Update the tests documentation for lighthouse/acceptance to guide the developer to update the new test output script with new test names and their modules as we no longer put test suite definitions in <code>.github/workflows/e2e_lighthouse_performance_acceptance_tests.yml</code> .	N/A	18 June	24 June
7	In order to ensure stability and that my proposed solution actually works, we should test and measure impact before completely committing to these changes. Luckily the <code>RUN_SUITES_ONLY_ON_CHANGED_FILES</code> allows us to do exactly this. We can get PR 3 merged then disable the <code>RUN_SUITES_ONLY_ON_CHANGED_FILES</code> environment variable to run all suites regardless of changed files. From there we can perhaps have the python script from PR 3 emit the tests even if <code>RUN_SUITES_ONLY_ON_CHANGED_FILES</code> is false (for debugging) and observe if there are any discrepancies in PRs by seeing if any failed tests fall in the changed files subset (only for frontend PRs). We can do this by looking at natural runs or sampling certain PRs and running them multiple times in a test PR.	1.1,1.2,1.3	24 June	28 June
8	If there are any issues that pop up after deploying the above to the Github CI or while testing, create a wiki page on how to debug common issues following this outline .			

Milestone 2

Key objective for this milestone:

- Make frontend tests run only on changed files, which will reduce the overall pre-push hook time.
- Optimize the pre-push hook by reducing the install-and-build step and look to remove it completely once the docker migration is stable.
- Create a backend unit test reporter, which will report any long backend unit tests inside the PR CI to ensure that all backend unit tests can run in a timely manner.
- Optimize any long-running backend unit tests and run backend unit tests in the pre-push hook, which will allow developers to get feedback on their backend code before they push.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	<p>Make frontend tests be able to run only for changed files, and add this functionality to the pre-push hook by only inputting changed files to the frontend tests, while also ensuring that all tests run in the PR CI.</p> <p>TASKS:</p> <ul style="list-style-type: none"> • Make frontend tests be able to run partially through this solution, using webpack context plugin to collect files depending on a mapping. • Add this functionality to the pre-push hook by simply passing the collected frontend files already existing to the <i>run_frontend_tests</i> script, which will be altered to add a param for this. 	N/A	9 July	14 July
2	<p>After PR 1 and other impacting PRs get merged, we should measure the impact it has on developer experience by collecting feedback from developers.</p> <p> Pre-Push Hook Impact Form</p>	N/A	14 July	12 August
3	Optimize the pre-push hook by reducing the time the install-and-build step takes using this solution .	N/A	10 July	16 July
4	Change the existing lint to run only on changed files.	N/A	10 July	16 July
5	Create a wiki page to guide developers on how to optimize their backend unit tests (e.g. via profiling with cProfile).	N/A	10 July	20 July
6	Create a reporter which reports any long running backend unit tests in the github CI. This reporter should also upload an artifact which contains all of the backend test names and times in sorted order.	N/A	16 July	22 July

	TASKS: <ul style="list-style-type: none"> • Create the base reporter script, which will take in backend shard files which contain test names and times. After combining these, detect if any of the tests are above 150 seconds and if so, report them at the end of the script. • Alter the <i>run_backend_tests</i> script to have a new flag which can output test names and test times. Alter the existing backend github workflow to upload github artifacts of these files (similar to coverage reports). Add a step which will run the base reporter script and log the long backend tests (>150s) in the CI and also upload a sorted list of all backend test names and times. 			
7	Update the backend unit tests documentation to include the new information about the reporter and what a developer should do if the CI complains about long tests.	2.6	20 July	26 July
8	Optimize any long-running (> ~100 seconds) backend unit tests, using the analysis from here (also with further analysis during this period).	N/A	24 July	4 August
9	Create the functionality to run backend unit tests in the pre-push hook and run them sorted by times from PR 2.6. Make sure that the pre-push hook doesn't timeout at 5 minutes by adding a runtime check which will cancel any running tasks if the pre-push hook were to exceed 5 minutes.	2.6	30 July	8 August
10	Change the running frontend/backend unit tests documentation to include the new <code>-files</code> arg, which allows developers to run multiple frontend/backend unit tests depending on the files they pass through.	2.1,2.6	30 July	8 August