GSoc’22 Proposal
Improving the lesson creation experience [part(a) + part(b)]

- Soumyajyoti Dey

Section 1: About You

What project are you applying for?

Improving the lesson creation experience [part(a) + part(b)].

Why are you interested in working with Oppia, and on your chosen project?

When I started with web development and spent some months on it, I wanted to gain some real world experience along with the normal hobby projects that I was into. So, I contacted some seniors in my college for some advice and they suggested that I should start contributing to open source. While searching for some good and systematic organizations to contribute, I came across oppia and started contributing right away. One of the best things I admire about oppia is that there are plenty of beginner friendly issues which are well organized so that anyone with any amount of experience can easily get started and learn new things along the way. I also admire the vision of oppia which is to provide quality education to children who don't have access to it. The mentors at oppia are also very helpful whenever someone gets stuck at something. All these things inspire me to work harder and keep contributing at oppia.

The reason for me choosing this project is because I wanted to work on a new feature and I think this project suits me the best. Sometimes, when there are many collaborators creating a lesson (exploration) in oppia, it becomes very difficult to keep track of who made the previous changes and the exact changes made by them. This project would enable this feature and help the creator and collaborators to take the necessary actions if something does not behave as expected. Also, currently, there are no means for a lesson creator to know about the changes made in exploration metadata (title, goals, tags etc.). Keeping track of metadata changes is important because sometimes, creators might accidentally make some unwanted changes on the exploration metadata and knowing these changes will make it easier for them to fix them if needed.

Prior experience

I have been involved in web development since November 2020 and got exposure to technologies such as React, Django, Angular, Unit testing, Firebase etc.. I am also a part of the development group of our college where I have contributed to many open source projects such as Hackalog, Institute App (Backend) and also worked for a project related to Training and Placement Cell IIT BHU. I have also made some personal projects along my journey. Some of them are:

a) Evader: It is an event management app written in react and django.

b) Group chat application: A group chat application made with react and firebase.

I have been contributing in oppia since September 2021 primarily for the Automated QA and LACE teams. Till date, I have merged around 19 PRs in oppia related to frontend testing, angular migration and lace quality team issues. I am currently working on writing beam jobs and backend validation checks for the LACE android team and also involved in a small project: **Identification of stale tabs and informing users about the same**. This project will be finished till mid April and would not hinder with the gsoc coding period.

My open source contributions other than oppia include contributions I made during hacktoberfest where I was able to complete the target of 4 PRs. During this period, I contributed to SUI Components (added the shape property in their input component), The New Boston Developers (converted some sass styles into styled components) and other organizations.

My list of PRs in oppia

PR	**Status**	**Topic**
Migrate contribution and review service and write the remaining frontend tests for it	Merged	Angular Migration.
Add frontend tests for some files	Merged	Unit testing.
Fix issues related to skill editor component in the exploration editor	Merged	Lace quality team issue.
Add frontend validation checks for some components	Merged	Front-end validation.
Add backend validation check for story description	Merged	Beam jobs and backend validation.

Project size

medium (~175 hours)

Project timeframe

I would be coding for an extended coding period (18 weeks) from June 13 to October 31. This is because I will be in college this summer to complete a few courses. Hence, the default coding period might be too rushed for me.

Contact info and timezone(s)

Email: deysoumyajyoti2017@gmail.com (This is also my Google Chat email).

Phone number: +91-8721979265
Preferred methods of communication:

- Google chat
- Google meet
- Email (I usually respond to important emails within the same day)
- Whatsapp (on the same phone number provided above)

Timezone(s): India Standard Time (GMT+5:30)
Github Profile: soumyo123-prog

Time commitment

During the coding period (June 13 - October 31), I would be able to commit at least 2.5 - 3 hours (flexible) per day which according to me is enough for completing the 175hr long project within the specified timeframe. However, this is flexible as I can increase my working hours depending upon the situation of the project.

Hence, the following is my time commitment schedule for the coding period:

- **Daily**: 2.5 - 3 hours. Can be increased if required.
- **Weekly**: 17.5 - 21 hours. Can be increased if required.

Essential Prerequisites

Answer the following questions (for Oppia Web GSoC contributors):

- I am able to run a single backend test target on my machine. (Show a screenshot of a successful test.)

```
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Mar 13, 2022 6:08:35 PM io.gapi.emulators.grpc.GrpcServer$3 operationComplete
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Mar 13, 2022 6:08:35 PM io.gapi.emulators.netty.HttpVersionRoutingHandler channelRead
[datastore] INFO: Detected HTTP/2 connection.
12:38:45 FINISHED core.domain.skill_domain_test.SkillDomainUnitTests: 23.3 secs
Stopping Redis Server(name="sh", pid=29208)...
Stopping Cloud Datastore Emulator(name="sh", pid=29108)...

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   core.domain.skill_domain_test.SkillDomainUnitTests: 44 tests (9.3 secs)

Ran 44 tests in 1 test class.
All tests passed.

Done!
```

- I am able to run all the frontend tests at once on my machine. (Show a screenshot of a successful test.)

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE                                                bash
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/@mapbox/node-pre-gyp/node_modules/.bin): Error: ENOSPC: System limi
t for number of file watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/@mapbox/node-pre-gyp/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/@lhci/utils/node_modules/.bin): Error: ENOSPC: System limit for num
ber of file watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/@lhci/utils/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/tslint/node_modules/.bin): Error: ENOSPC: System limit for number o
f file watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/tslint/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/d3-fetch/node_modules/.bin): Error: ENOSPC: System limit for number
 of file watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/d3-fetch/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/d3/node_modules/.bin): Error: ENOSPC: System limit for number of fi
le watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/d3/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/firebase-tools/node_modules/.bin): Error: ENOSPC: System limit for
number of file watchers reached, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/firebase-tools/node_modules/.bin'
Error from chokidar (/home/spunky/Coding/Projects/oppia/oppia/node_modules/.bin): Error: ENOSPC: System limit for number of file watchers reac
hed, watch '/home/spunky/Coding/Projects/oppia/oppia/node_modules/.bin'
Chrome Headless 96.0.4664.110 (Linux x86_64): Executed 7344 of 7344 SUCCESS (2 mins 42.193 secs / 2 mins 18.713 secs)
TOTAL: 7344 SUCCESS
TOTAL: 7344 SUCCESS
13 03 2022 17:54:59.074:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!
```

- I am able to run one suite of e2e tests on my machine. (Show a screenshot of a successful test.)

```
 be used
[18:22:26] W/element - more than one element found for locator By(css selector, .protractor-test-state-content-editor) - the first result will
 be used
[18:22:53] W/element - more than one element found for locator By(css selector, .protractor-test-merge-skills-button) - the first result will
be used
[18:22:53] W/element - more than one element found for locator By(css selector, .protractor-test-merge-skills-button) - the first result will
be used
[18:22:53] W/element - more than one element found for locator By(css selector, .protractor-test-merge-skills-button) - the first result will
be used
[18:22:53] W/element - more than one element found for locator By(css selector, .protractor-test-merge-skills-button) - the first result will
be used
.    ��� should merge an outside skill with one in a topic



4 specs, 0 failures
Finished in 299.726 seconds

Executed 4 of 4 specs SUCCESS in 5 mins.
[18:23:10] I/launcher - 0 instance(s) of WebDriver still running
[18:23:10] I/launcher - chrome #01 passed
Stopping Protractor Server(pid=33871)...
Stopping Webdriver manager(name="sh", pid=33811)...
Stopping GAE Development Server(name="sh", pid=32327)
```

Other summer obligations

I am not applying to any other jobs during this summer. Also, I am only applying to oppia for GSOC this year. Regarding classes, my summer vacations are from mid May to mid July and after that, my classes will resume. However, in any circumstance, 3-3.5 hours can be easily committed per day towards this project.

Communication channels

I plan to connect with my mentors two times a week through google meet or discord in order to give my weekly updates. However, this is flexible and can be adjusted by discussion with the mentors.

Section 2: Proposal Details

Problem Statement

Link to PRD (or N/A if there isn't one)	N/A
Target Audience	Lesson creators
Core User Need	- As a lesson creator, before publishing a new version of the exploration, I will need to make sure that all the changes made to migrate into the new version of the exploration are valid and there are not any unnecessary or wrong changes (both consciously and unconsciously). - As a lesson creator, it is difficult for me to keep track of changes made to an exploration properly. This is essential for me so that I can figure out any unnecessary or wrong changes to a state which are responsible for the exploration not behaving in the expected way so that I can fix the issue. The current system of comparing versions with the help of history tab does not help very much in this case because I will often have to keep comparing consecutive versions of the explorations in order to see if that particular state has changed and if so, then examine and find the changes made into it. This often consumes a lot of my time and effort.
What goals do we want the solution to achieve?	- During version comparison, above the comparison graph, the user would see a link saying "View metadata changes". Clicking this would open a modal showing the changes made to the exploration metadata between the two selected versions. - In each state card, the user would see a link just below the state name saying "Latest commit by XXX at version YYY". Clicking this would open a modal and show the changes made to the state card while migrating from version YYY to version YYY + 1. On the modal, the user will see another link reading "Previous commit by PPP at version QQQ" and another one reading "Next commit by ABC at version PQR". The user will also be able to explore the commit history of exploration metadata properties.

Section 2.1: WHAT

This section enumerates the requirements that the technical solution outlined in "Section 2: HOW" must satisfy.

Key User Stories and Tasks

#	Title	User Story Description (role, goal, motivation) "As a ..., I need ..., so that ...."	Priority[1]	List of tasks needed to achieve the goal (this is the "User Journey")	Links to mocks / prototypes, and/or PRD sections that spec out additional requirements.

[1] Use the **M**o**SC**ow system ("Must have", "Should have", "Could have"). You can read more here.

1	Exploration metadata diff between selected versions.	As a lesson creator, I need to be able to see the changes made to the exploration metadata along with changes in each state card so that I can inspect any unwanted changes made into any exploration property (title, goals, tags etc.) and fix them later if needed.	Must have	- Select two versions for comparison. - Click on metadata changes in the visualization graph. - See the changes in the exploration metadata between the two selected versions.	Metadata Diff : Prototype
2	Changes made on a state card.	As a lesson creator, I need to be able to see the latest changes made to a state card and the creator of those changes. This would help me to track the exact changes in case of any unexpected behavior of the exploration and take the necessary actions to fix them.	Must have	- Open any exploration and select a state card. - Look for the link reading "Latest commit by XXX at version YYY " placed at the bottom right corner of the state editor. Click on the link. - See the modal showing changes made to the state card from version YYY to version YYY + 1.	Version history explorer: Prototype

Technical Requirements

Additions/Changes to Web Server Endpoint Contracts

#	Endpoint URL	Request type (GET, POST, etc.)	New / Existing	Description of the request/response contract (and, if applicable, how it's different from the previous one)
1.	/explorehandler/init/<exploration_id>?v=<version>	GET	Existing	This request is used by **ReadOnlyExplorationBackendApiService** to fetch version specific exploration data from the backend. This data is used during version comparison by the history tab. The response dict of this request will be slightly modified to

				include an extra property called **'exploration_metadata'**. This new property will be a dict and its structure will be as follows: ● title: str. ● category: str. ● objective: str. ● language_code: str. ● tags: list[str]. ● blurb: str. ● author_notes: str. ● param_specs: dict[str, dict of ParamSpec domain object]. ● param_changes: list[dict of ParamChange] ● init_state_name: str. ● auto_tts_enabled: boolean. ● correctness_feedback_enabled: boolean. ● states_schema_version: number. ● edits_allowed: boolean. **URL Parameters:** exploration_id: str
2.	/version_history/<exploration_id>/<version>/<state_name>	GET	New	This request will be used to fetch the previous version history for a particular state of an exploration at a particular version. **URL Parameters:** exploration_id: str version: int state_name: str
3.	/version_history/<exploration_id>/<version>/metadata	GET	New	This request will be used to fetch the previous version history for the exploration metadata at a particular version. **URL Parameters:** exploration_id: str version: int

Calls to Web Server Endpoints

#	Endpoint URL	Request type (GET, POST, etc.)	Description of why the new call is needed, or why the changes to an existing call is needed
1.	/explorehandler/init/<exploration_id>?v=<version>	GET	The changes to this existing call are required to include the exploration metadata in the response dict of this request.
2.	/version_history/<exploration_id>/<version>/<st	GET	This new request is needed to fetch the previous version history of a particular state of an exploration at a particular version.

	ate_name>		
3.	/version_history/<exploration_id>/<version>/metadata	GET	This new request is needed to fetch the previous version history of the exploration metadata at a particular version.

UI Screens/Components

#	ID	Description of new UI component	i18n required?	Mock/spec links	A11y requirements
1.	Additional link to show metadata changes.	It will be placed above the comparison which, when clicked, will show the changes in exploration metadata between the two selected versions.	No	Additional link to show changes in Metadata	No
2.	Metadata diff modal.	It will be a new modal which will show the diff in exploration metadata between the two selected versions.	No	Metadata diff modal	No
3.	“Latest commit by XXX at version YYY” annotation at the state cards.	It is an information box placed next to the state name in each state editor which would contain information about the latest commit and also contain a link which, when clicked, would open a modal showing the diff.	No	The new annotation in the state cards	No
4.	“Latest commit by XXX at version YYY” annotation at the exploration settings tab.	It is an information box placed at the top of the exploration settings tab which would contain information about the latest commit and also contain a link which, when clicked, would open a modal showing the diff.	No	The new annotation in the settings tab	No
5.	Modal that shows the difference between the versions YYY and YYY + 1.	As the name suggests, this modal will show the differences in the state card between versions YYY and YYY + 1 of the exploration. It will also be similar to the existing state diff modal.	No	Difference modal	No
6.	Interstitial loading screen.	It will be a simple loading screen which will be shown when the diff data between 2 versions of a state is being fetched.	No	Interstitial loading screen	No

Data Handling and Privacy

#	Type of data	Description	Why do we need to store this data?	Anonymized?	Can the user opt out?	Wipeout policy	Takeout policy
1	Previous commit data (version number, state name and committer id) of each state in each version of the exploration.	The structure of the data is explained in detail in the section: Method 3: Precomputation approach (b) (Efficient and scalable approach).	We need this data to allow the user to explore the commit history of an exploration.	No. Because this data is explicitly tied to the user.	No.	Locally Pseudonymised once the user deletes their account.	N/A. Because this data does not contain relevant data corresponding to users.

Section 2.2: HOW

Existing Status Quo

Currently, there is a history tab in the exploration editor page which facilitates the version comparison between two selected versions of the exploration. However, the user cannot see the changes in exploration metadata during the version comparison. Also, currently, the user cannot navigate over the version history of a state.

Meaning of some phrases and terms used in the proposal

- **Previous version history of a state**
  - It indicates the previous version of the exploration on which the state has been edited along with the information of the state name at the previous version and the username of the user who committed those changes.
- **Previously edited version OR Previously edited version number of a state**
  - It refers to the version number of the exploration on which the state was previously edited.

Solution Overview

Subproject (a)

- Currently, the following properties are considered as the exploration metadata:
  - Title
  - Category
  - Objective

- Initial state name
- Language code
- Correctness feedback enabled status
- Auto text to speech enabled status
- Tags
- Blurb
- Author notes
- Param specs
- Param changes
- Edits allowed
- States schema version

Representation of exploration metadata

Frontend: ExplorationMetadata domain object

- In the frontend, there already exists an ExplorationMetadata domain object. However, it represents only three properties i.e. id, objective and title. This domain object is used to represent exploration search results. Hence, this will be renamed to something like: **ExplorationSearchResult.**
- After renaming, we will create a new domain object called **ExplorationMetadata.**
- **Properties:**
    - title: string.
    - category. string.
    - objective: string.
    - languageCode: string.
    - tags: string[].
    - blurb: string.
    - authorNotes: string.
    - statesSchemaVersion: number.
    - initStateName: string.
    - paramSpecs: ParamDict.
    - paramChanges: ParamChange[].
    - autoTtsEnabled: boolean.
    - correctnessFeedbackEnabled: boolean.
    - editsAllowed: boolean.
- **Functions:**
    - Basic getters and setters of the above properties.
    - fromBackendDict.
    - toBackendDict.

Backend: ExplorationMetadata domain object

- This will also be newly created.
- **Properties:**

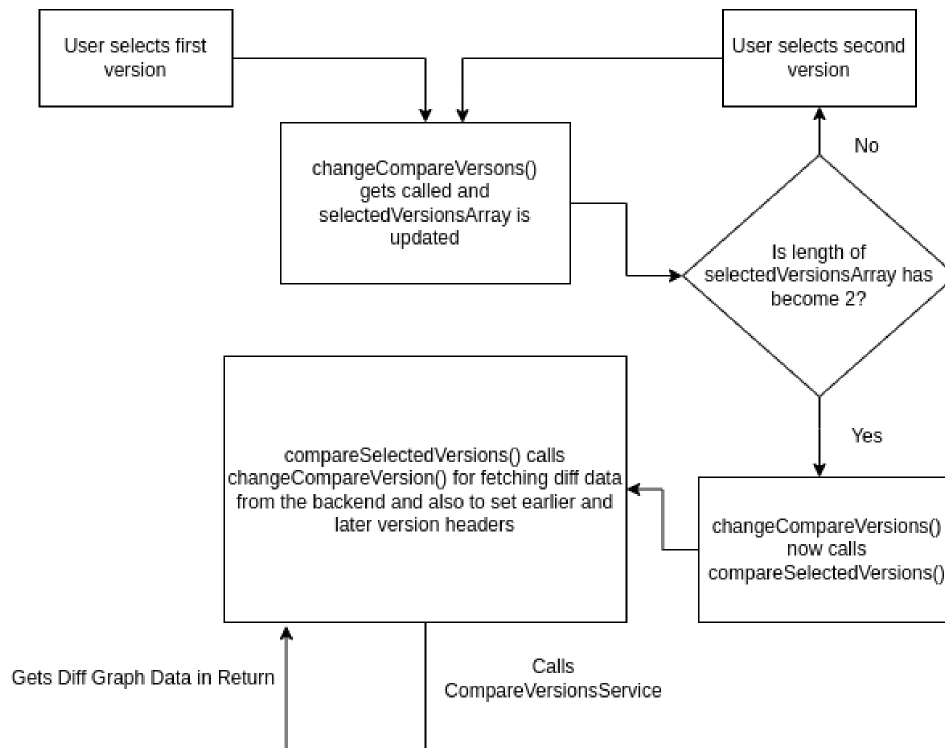
- title: str.
    - category: str.
    - objective: str.
    - language_code: str.
    - tags: list(str)
    - blurb: str.
    - author_notes: str.
    - states_schema_version: int.
    - init_state_name: str.
    - param_specs: dict.
    - param_changes: list(ParamChange)
    - auto_tts_enabled: boolean.
    - correctness_feedback_enabled: boolean.
    - edits_allowed: boolean.
- **Functions:**
    - to_dict

Getting metadata information from the backend

Current system:

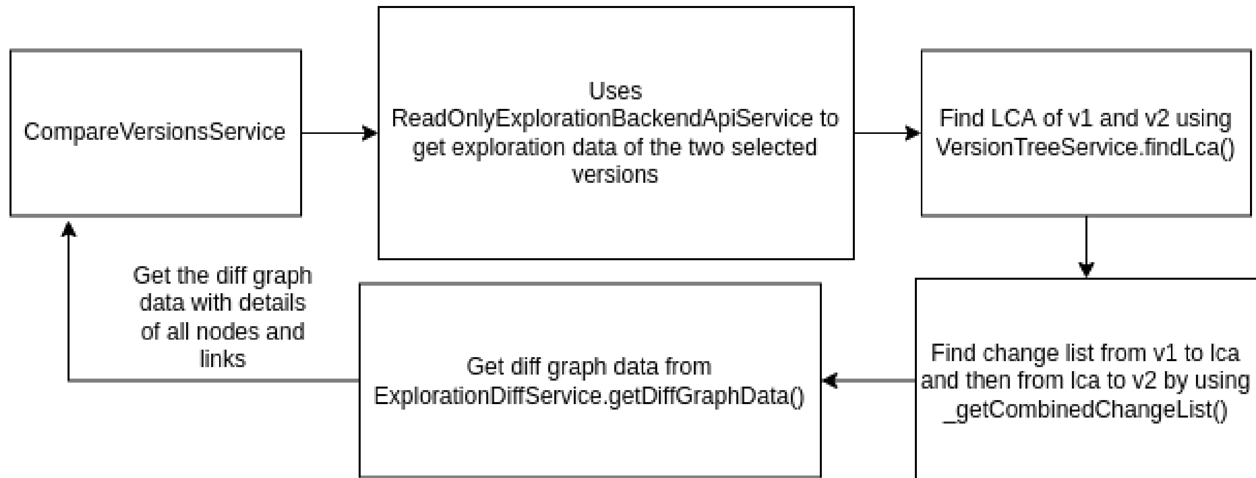
Steps taking place in history tab component while fetching diff graph data

- User selects the first version: **changeSelectedVersions()** of the history tab component gets called. This function modifies the **selectedVersionsArray** to include the first selected version.
- User selects the second version: the same process happens again and **selectedVersionsArray** gets modified.
- Now, the length of **selectedVersionsArray** has become two and **changeSelectedVersions()** now calls **compareSelectedVersions()** automatically.
- **compareSelectedVersions()** in turn, calls **changeCompareVersion()**. This function uses **CompareVersionsService** to get the processed diff graph data from the backend and also sets the **earlier and later version headers** (Eg: Revision #4 by user1 (Mar 12, 3:33 PM)).



Steps taking place in CompareVersionsService.getDiffGraphData()

- Fetching the exploration data for the two selected versions (v1 and v2) using the **ReadOnlyExplorationBackendApiService.loadExplorationAsync()** function.
- Find the LCA of versions v1 and v2 where LCA is the lowest common ancestor of the two versions in the version tree. For this, **VersionTreeService.findLca()** is used.
- Now, the change list is calculated to go from version v1 to v2. First, we calculate changes from v1 to lca and then from lca to v2. For this, **CompareVersionsService** has a function called **_getCombinedChangeList**.
- Now, the diff graph data is calculated by using **ExplorationDiffService,getDiffGraphData()** which takes the v1States (states dict of the first selected version), v2States (states dict of the second selected version) and the changeList as arguments and returns diff graph data.



Required changes:

Modification of the return value of ExplorationHandler

- The request for fetching the exploration data for different versions is handled by **ExplorationHandler** in **reader.py**.
- The return value of the **ExplorationHandler** will be slightly changed to include a new property called **exploration_metadata** which will include the metadata properties for that particular version of the exploration.
- exploration_metadata: exploration.get_metadata().to_dict()

Addition of a new get_metadata function in Exploration domain object

- This function will return an instance of ExplorationMetadata domain object:

```
def get_metadata(self):
        exploration_metadata = ExplorationMetadata(
            self.title, self.category, self.objective, self.language_code,
            self.tags, self.blurb, self.author_notes, self.states_schema_version,
            self.init_state_name, self.param_specs, self.param_changes,
            self.auto_tts_enabled, self.correctness_feedback_enabled,
            self.edits_allowed)
        return exploration_metadata
```

The changes can be tabulated as follows:

File name	**Function name**	**List of changes**
reader.py	ExplorationHandler : get method	Include a new property called **exploration_metadata** in the final return value.
exp_domain.py	Exploration domain object	Add a new function called **get_metadata**

		as mentioned above.

Caching of the fetched exploration versions make the comparison faster

Current system

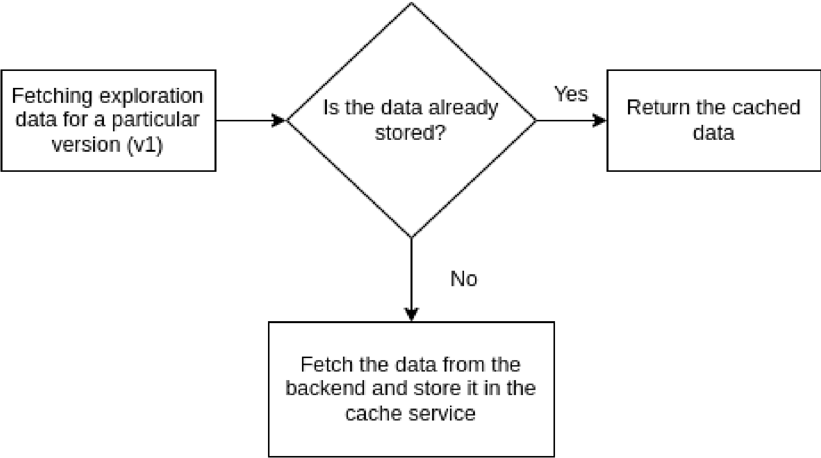
- Currently, in the **read-only-exploration-backend-api-service** there is some sort of caching which stores the latest version of exploration with a particular id.
- However, there is no version specific cache service available to store version specific exploration data. Having a service like this would help in fetching version specific data for previously fetched versions of an exploration.

Required changes

- A new cache service will be created called **ExplorationVersionCacheService.** It will store data in the following format:

```
interface ExplorationVersionCache {
 [explorationId: string]: {
   [version: number]: FetchExplorationBackendResponse;
 };
}
```

The cache service will work as follows:



- While fetching version specific exploration data using **ReadOnlyExplorationBackendApiService.loadExplorationAsync**, a new if-check will be added to check if the data for that particular exploration and that particular version is already cached.
- If so, then the cached data will be returned. Otherwise, the data will be fetched from the datastore and will be cached for further use.

The changes can be tabulated as follows:

File name	Function name	List of changes
exploration-version-cache.service.ts	ExplorationVersionCacheService	Create the new service

Addition of two new properties in the Compare versions service's return value

- These properties will include the values of exploration metadata for the two selected versions.
- The two new properties will be named **v1Metadata** and **v2Metadata**.
- Finally, we have all the information we needed:
  - Metadata node in the diff graph data
  - The metadata information of the earlier and later versions.
  - Later on, while visualizing metadata diff, **v1Metadata** and **v2Metadata** will become the old and new metadata dicts respectively.

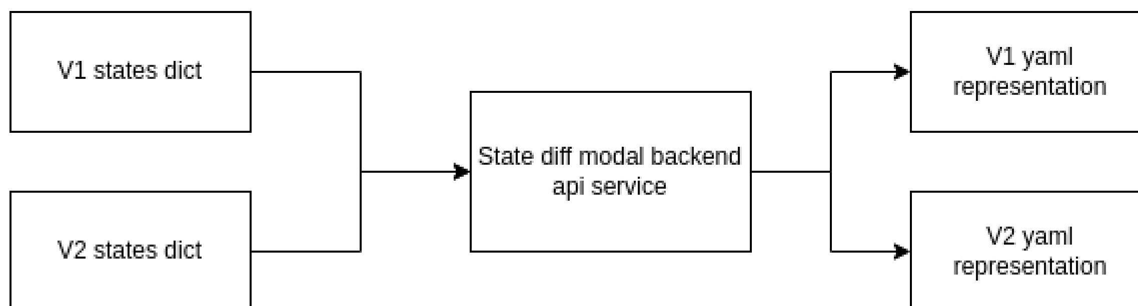
The changes can be tabulated as follows:

File name	Function name	List of changes
compare-versions.service.ts	getDiffGraphData	Add the properties v1Metadata and v2Metadata in the return value of this function.

Creation of a new service for doing yaml conversions

Current system

- Currently, the diff data between different states is visualized by converting the old and new state dicts into yaml strings and visualizing the diff between different strings using codemirror.
- The Codemirror component takes the old state dict (left value) yaml and the new state dict (right value) yaml and visualizes the diff between the states.
- **StateDiffModalBackendApiService** is used for conversion of state dict to yaml which takes the state dict and yaml width as payload and returns the yaml string.

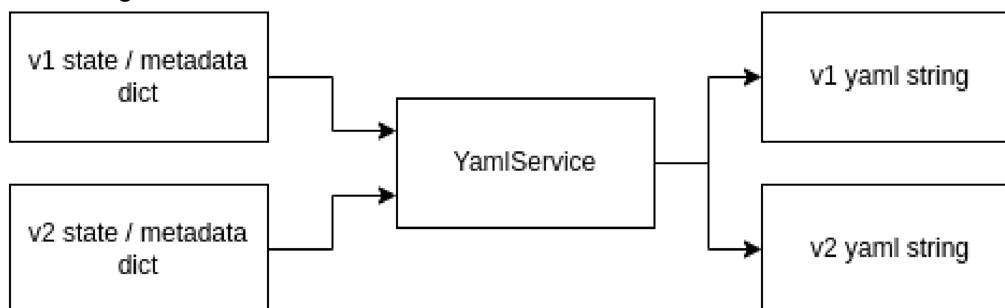


Flaws in the current system

- The current approach is asynchronous and takes some time to finish because of being dependent on the backend server for doing the conversion.
- If the backend api fails for some reason, the version diff visualization would appear broken.
- Hence, using a frontend based approach would help us tackle the cons of the backend approach and also fasten the process a little bit.

Required changes

- The conversion of state or metadata dicts into yaml will be facilitated by using the ‘js-yaml’ library in the frontend itself.
- For doing this, a new service will be created in the frontend called **YamlService**. This service will contain basic methods such as stringify and parse.
- The yaml representation of the metadata dict will be used by CODEMIRROR for showing the changes.



The changes can be tabulated as follows:

File name	Function name	List of changes
yaml.service.ts		Create the service.

Working of the YamlService

The YamlService will have the following structure:

```
import { Injectable } from '@angular/core';
import { downgradeInjectable } from '@angular/upgrade/static';
```

```
import yaml from 'js-yaml';

@Injectable({
  providedIn: 'root'
})
export class YamlService {
  constructor() {}

  stringify(object: unknown): string {
    return yaml.dump(object);
  }
}
```

The function **stringify** will be used to convert the object into yaml string.

Creation of a new YamlConversionService

- This service will be used to share the logic to get left (old) and right (new) yaml strings from the state/metadata dicts. This will help us avoid duplication of the ‘yaml conversion logic’.
- **Functions:**
  - **getYamlStringFromObject(object): Promise<string>**

```
    if (object) {
      setTimeout(() => {
        return this.yamlService.stringify(object);
      }, 200);
    } else {
      setTimeout(() => {
        return '';
      }, 200);
    }
```

The timeout is required to allow CODEMIRROR to fully load.

Example usage of YamlConversionService (in the already existing StateDiffModal)

```
  ngOnInit(): void {
    this.yamlConversionService.getYamlStringFromObject(
      this.oldState.toBackendDict()
    ).then((result) => {
      this.yamlStrs.leftPane = result;
    });
```

```
  this.yamlConversionService.getYamlStringFromObject(
    this.newState.toBackendDict()
  ).then((result) => {
    this.yamlStrs.rightPane = result;
  });
}
```

Showing the metadata diff modal to the user

For this, the following changes need to be done:

Creation of a new Metadata Diff Modal Component

- It will be used to show the metadata diff using codemirror.
- It will have the following properties:
    - **oldMetadata**: The metadata properties of the older version.
    - **newMetadata**: The metadata properties of the newer version.
    - **headers:** The headers to be shown at the top of the modal. (Eg: Revision #4 by user1 (Mar 12, 3:33 PM)). It will be an object with the following structure:
        - leftPane: string. The headers of the first selected version.
        - rightPane: string. The headers of the second selected version.
    - **yamlStrs:** The yaml strings from the old and the new metadata. It will have the following structure:
        - leftPane: string. The yaml string of the oldMetadata dict.
        - rightPane: string. The yaml string of the newMetadata dict.
- The **ngOnInit** function of this component is important as it will be used to do the yaml conversions as follows:

```
ngOnInit(): void {
  this.yamlConversionService.getYamlStringFromObject(
    this.oldMetadata.toBackendDict()
  ).then((result) => {
    this.yamlStrs.leftPane = result;
  });
  this.yamlConversionService.getYamlStringFromObject(
    this.newMetadata.toBackendDict()
  ).then((result) => {
    this.yamlStrs.rightPane = result;
  });
}
```

Changes in History Tab Component

- In the html file of that component, add a new button according to the mocks.

- Create a new function called **showMetadataDiffModal** in the history tab component which will react to the 'click' event of the new button added.
- This function will be pretty much similar to the **showStateDiffModal** function of the version-diff-visualization component.
- It will have the following structure:

```
ctrl.showMetaDataDiffModal = function() {
        let modalRef: NgbModalRef =
NgbModal.open(MetaDataDiffModalComponent, {
          backdrop: true,
          windowClass: 'metadata-diff-modal',
          size: 'xl'
        });

        modalRef.componentInstance.oldMetaData = ctrl.diffData.v1MetaData;
        modalRef.componentInstance.newMetaData = ctrl.diffData.v2MetaData;
        modalRef.componentInstance.headers = {
          leftPane: ctrl.earlierVersionHeader,
          rightPane: ctrl.laterVersionHeader
        };
        modalRef.result.then(function() {}, function() {});
      };
```

Extension of this functionality for a newly added metadata field

- For this, we need to make sure that all the properties present in the Exploration domain object (except id, version and states) are also present in the ExplroationMetadata domain object.
- The to_dict method of exploration domain objects returns all the metadata properties.
- We can call the to_dict method of both the exploration domain object and exploration metadata domain object and compare that we have all the properties in both the dicts (except id and states).
- If any property (Except id or states) is missing in the new domain object, then the test will fail notifying that the new property should be added in the ExplorationMetadata domain object.
- A new backend test can be added for this.

Subproject (b)

Decision on the structure and schema of the version history of a state:

Method 1: Without precomputation

In this method, we will be computing and fetching the whole version history of all the states of an exploration when the exploration editor page loads for the first time and update the version history in the frontend itself when the user saves some changes on the exploration.

Current System:

- Currently, in the **exploration editor page component**, there is a function called **initExplorationPage** which loads the required data from the backend and initializes the exploration editor page along with all the services required by it.
- Also, for generating the version history list of all states, we need access to the changes that were applied to the exploration at all the versions. For this, we have the **ExplorationCommitLogEntryModel.** It has a **get_multi** method that takes the exploration id and a list of versions of the exploration as arguments and fetches the commit logs for all the versions at once.

Required Changes

- A new backend api service will be created named **ExplorationStatesVersionHistoryBackendApiService** which will send a get request to fetch the versions history list of all states in the format explained in [Structure of the version history of a state](#).
- In the **initExplorationPage** function as discussed above, we will use this backend api service to fetch the required data.

In the backend, the request will be processed as follows:

- Initialize the versions history list as:

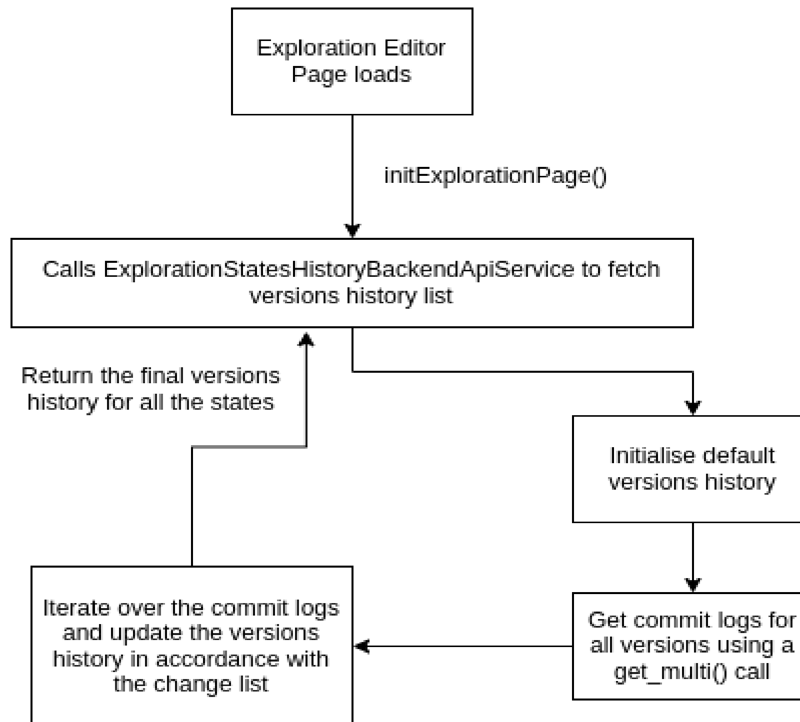
```
versions_history = {
          'Introduction': []
      }
```

- It is initialized like this because when a default exploration is created, it has only one state and its name is 'Introduction'.
    - Fetch the commit logs for all the versions of the exploration so that the versions history can be updated accordingly. As discussed above, it is fetched by **ExplorationCommitLogEntryModel** by using a **get_multi** method.

```
commit_logs = exp_fetchers.get_commit_logs_for_exploration_versions(
            exploration.id, versions)
```

    - Now, we will iterate over the commit logs and in each iteration, do the following:

        - Get the change list
        - Iterate over the change list and update the versions history accordingly. This process is similar to the function **apply_change_list of exp_services.**
    - After all the iterations over the commit logs and change lists, we get the final versions history list of all the states. At last, it is returned to the frontend.



- After fetching the versions history list of all states, we can show the "Last/Previously edited by ...." link to the user. Each time the user clicks on this link, the diff data between the versions will be fetched by CompareVersionsService for visualization.

If the backend API fails for some reason

- The backend api can fail when there is corrupted data in the datastore due to which the generation of version history for the exploration states fails. In this case, the corrupted data will need to be fixed manually.
- When the backend api fails, the rest of the exploration editor page will not become unusable and will work perfectly fine.
- Just the "Last edited by ..." link will be hidden from the user for that particular exploration. The rest of the page will not be affected.

Method 2: Precomputation approach (a)

Storage of the version history

Per exploration

- The full version history of each state of an exploration will be stored together.
- **Generation of the model id:**
  - Since there will be one model for each exploration, the id of the model will be the same as the exploration id.
- **Schema:**
  - **version_history**: dict. The full version history of each state of the exploration.
    - **Key**: str. The state name.
    - **Value**: list. The version history list of a state with each element being an object having the following properties.
      - **version**: int. If version is v, it indicates that the state has been modified while going from version v to v + 1. This list will not contain versions which were associated with only translation commits.
- **Lifecycle of the model:**
  - **[CREATION]:**
    - A new instance of this model will be created each time a new exploration is created.
    - The version history of each state will be initialized as empty lists while creation.
  - **[UPDATION]:**
    - The version_history will be updated according to the change_list when new changes are saved onto the exploration.
    - **Pseudo algorithm:**
      - Fetch the version history.
      - Update version history of all states according to the change list i.e. append the new version of the exploration to each state which is updated in this current change.
      - Save the updated version history.
  - **[DELETION]** The data of the model will be deleted when the exploration gets deleted.

Per state

- Here, the full version history of a particular state of an exploration will be stored.
- **Generation of the model id:**
  - Since the name of the states of an exploration should be unique. Hence, for the “per state” model, the unique id of each model will be (**exploration_id** + **state_name**).

- **Schema:**
    - **exploration_id**: str. The id of the exploration to which the state belongs to.
    - **state_name**: str. The name of the exploration state.
    - **version_history**: list. The version history list of a state with each element being an object having the following properties:
        - **version**: int. If version is v, it indicates that the state has been modified while going from version v to v + 1. This list will not contain versions which were associated with only translation commits.
- **Lifecycle of the model:**
    - **[CREATION]:**
        - A new instance of this model will be created for each state of the exploration each time a new exploration is created.
        - For each state, the version history will be initialized as an empty list.
        - Save the models by using a **put_multi** method.
    - **[UPDATION]:**
        - The version_history will be updated according to the change_list when new changes are saved onto the exploration for each state of the exploration which are modified.
        - **Pseudo algorithm:**
            - Get the state names which have been modified from the change list.
            - Generate a list of model ids by combining exploration_id + state_name from the above list.
            - Get the models for the states which have been modified by using a **get_multi** call.
            - For states that have been removed, delete the models for those states using a **delete_multi** method.
            - For states that have been added, create the models for those states.
            - For states that have been renamed, delete the model with old_state_name and create the model with new_state_name with content the same as the previous one.
            - Lastly, append the new version of the exploration to the version history of the states which have been modified.
            - Save the models by using a **put_multi** method.
    - **[DELETION]** The data for each state of the exploration will be deleted when the exploration gets deleted.

Comparison between the two schemas

	Weight of the point of comparison	Per exploration	Per state

Efficiency while fetching of history data	3 (We will be fetching this data very frequently)	While fetching for the active state, this method will have to fetch the whole version history of all states first and then return the version history for the active state. This means that a lot of unnecessary data is being fetched from the datastore while we just need a chunk of it.	While fetching for the active state, this method will just fetch the required data because data for each state will be separately stored. This means that there will not be any unnecessary data being fetched from the datastore. Hence, this method will be more efficient and fast.
Efficiency while updating the data	3 (This process will also occur very frequently)	While updating, this method will first fetch the full version history for all the states, update the data according to the change list and finally save them in the datastore. If a large portion of the states have been updated during that commit, then this method will be an efficient way. However, if only a small portion of states are updated, then we will be fetching a lot of data unnecessarily.	While updating, this method will fetch the data for the states which have been modified, update their data according to the change list and finally save them in the datastore. This method will involve fetching and saving multiple models. Hence, it might be more efficient when only a small number of states have been modified. However, if a large number of states are modified, it will become inefficient.
Ease of implementation	1 (Ease of implementation will not be a very major decider)	Relatively easier to implement than the second method.	Relatively more difficult to implement than the first method.
Handling of exploration reverts	2 (This can also be a significant decider)	Difficult to implement as one will have to delete all the versions greater than the version to which the exploration has been reverted from the version history list of all states. However, the plus point in this approach will be that the modifications can be done by	Even more difficult and inefficient to implement as one will have to fetch multiple version history models for all the states first and delete the versions greater than the version to which the exploration has been reverted.

		fetching and saving only one model and not multiple.	

Color code:
- **Red:** Not desirable (Score = 0)
- **Yellow:** Mediocre (Score = 1)
- **Green:** Most ideal (Score = 2)

Scores:
- Per exploration: 7.
- Per state: 10.

From the above comparison table and calculating the scores by considering the weights of each point, we can say that using the second approach will be beneficial if we use this method.

Method 3: Precomputation approach (b) (Efficient and scalable approach)

Storage of the version history

Per exploration per version

- The number of version history models for a particular exploration will be equal to the number of versions of the exploration.
- Here, for each version of the exploration, we will store the "previously edited version number" for all the states present at that version.
- **Generation of the model id:**
  - Here, to generate a unique id, we can use (exploration_id + exploration_version). Since exploration_version is an increasing quantity, the id will always be unique.
- **Schema:**
  - **exploration_id:** str. The id of the corresponding exploration.
  - **exploration_version:** int. The version number of the exploration.
  - **version_history**: dict. A mapping of the state names of an exploration in its current version and their version history.
    - **Key:** str. The name of the state.
    - **Value:** dict.
      - The structure of the dict will be as follows:

```
'previously_edited_on_version': int
'state_name_in_previous_version': str
'committer_id': str
```

    - If the value of "previously_edited_on_version" for a state is None, then it would mean that the state was newly added in the current exploration version.

    - **metadata_previously_edited_on:** int. The version number of the exploration when the metadata was previously edited.
    - **metadata_previously_edited_by:** string. The id of the user who committed changes to exploration metadata previously.
- **Lifecycle of the model:**
    - **[CREATION]:**
        - Create a new instance of this model whenever a new exploration is created.
    - **[UPDATION]:**
        - The updation process explained in detail in the below section Updation process of the old version history to get a new one during each exploration save.
    - **[DELETION]:**
        - Delete all the instances of this model when the exploration is deleted.

Per state per version

- In this approach, all the processes will be the same except that we will be storing the data for each state and each version of the exploration.
- The schema of the data will be the same as the schema used for each state in the previous method.
- The id of the exploration will be generated by using exploration_id + state_name + exploration_version.

Comparison between the storage schemas

	Weights	Per exploration per version	Per state per version
Efficiency while fetching of data	3	For this, we will first fetch the 'previous version history' data for all the states and then return the data for the active state. Unlike the previous method, we will not be fetching a lot of unnecessary data while doing this operation as each model will store only one piece of version history rather than storing the full data.	For this, we will just fetch the required data for the active state and return it. Hence, it will again be the most ideal approach for this operation.
Efficiency while updating	3	While updating, we will first fetch the model for the previous version, update the	While updating, we must fetch and save multiple models corresponding to the states

the data		data for the states that have been edited and then save the data. For this operation, this method will be more efficient as the problem of fetching huge amounts of unnecessary data is not present anymore (because each model is storing only a piece of the version history)	which have been updated. Due to fetching and saving multiple models, this method will become less efficient in this case.
Handling of exploration reverts	2	Reverts can be easily handled with this method with the approach explained in [Handling of exploration reverts]() with just fetching a single model.	Here, again multiple models will have to be fetched and updated separately. This will decrease the efficiency of this method.
Ease of implementation	1	Relatively easier to implement than the second method.	Relatively more difficult to implement than the first method.

Color code:

- **Red:** Not desirable (Score = 0)
- **Yellow:** Mediocre (Score = 1)
- **Green:** Most ideal (Score = 2)

Scores:

- Per exploration per version: 15.
- Per state per version: 12.

Hence, the “Per exploration per version” schema would be beneficial for this approach.

Updation process of the old version history to get a new one during each exploration save

Let us suppose that the version of the exploration after saving becomes ‘v’. The updation process will be carried out inside the **_save_exploration** function of **exp_services.** It will be carried out in the following fashion below (sequentially):

Fetching the old version history model (for version ‘v-1’)

```
current_version = exploration.version
prev_version = current_version - 1
```

```
old_version_history_model = exp_models.ExplorationVersionHistoryModel.get(
    exploration.id + str(prev_version))
version_history = old_version_history_model.version_history
```

Calculating the version diff from the change list

- This will be done using the **ExplorationVersionsDiff** domain object.

```
exp_versions_diff = exp_domain.ExplorationVersionsDiff(change_list)
```

- Some general information about exp_versions_diff which is useful for the steps described below: **(Reference: DocString of the ExplorationVersionsDiff domain object)**
  - **added_state_names**: list(str). Names of the states added to the exploration from prev_exp_version to current_exp_version. It stores the newest names of the added states.
  - **deleted_state_names**: list(str). Name of the states deleted from the exploration from prev_exp_version to current_exp_version. It stores the initial names of the deleted states from pre_exp_version.
  - **old_to_new_state_names**: dict. Dictionary mapping state names of prev_exp_version to the state names of current_exp_version. It doesn't include the name changes of added/deleted states.

Handling of exploration state removals

- After getting the deleted state names from **exp_versions_diff**, we will remove the state names from the old version history model.

```
for state_name in exp_versions_diff.deleted_state_names:
        del version_history[state_name]
```

Handling of exploration state additions

- After getting the added state names from **exp_versions_diff**, we will add the state names to the old version history model.
- For each added state, its version history will be initialized as **None** because these states were added for the first time and have no 'previously edited version number'.

```
for state_name in exp_versions_diff.added_state_names:
        version_history[state_name] = (
            state_domain.StateVersionHistory(None, None))
```

Handling of exploration state renames

- After getting the old to new state names mapping, we can iterate over the exp_versions_diff.old_to_new_state_names and update the version history accordingly.

```
for old_state_name, new_state_name in
exp_versions_diff.old_to_new_state_names.items():
```

```
        version_history[new_state_name] = state_domain.StateVersionHistory(
            prev_version, old_state_name)
        del version_history[old_state_name]
```

Handling of changes in state properties (only for those states which are present in both the previous and new versions in the explorations and not have been renamed)

- For this, we first have to calculate those state names.
- Iterate through the old_states of the exploration (states in the older version of the exploration) and if the following conditions are satisfied, we can append those state names:
    - If the state_name does not belong to exp_versions_diff.deleted_state_names.
    - If the state_name does not belong to any key of exp_versions.old_to_new_state_names.

```
    other_modified_state_names = []
    for state_name in old_states:
        if (
            not(state_name in exp_versions_diff.deleted_state_names) and
            not(state_name in exp_versions_diff.old_to_new_state_names)
        ):
            other_modified_state_names.append(state_name)
```

- Now, we will iterate through the change list and flag the states from other_modified_state_names whose properties have been changed:

```
    state_data = {
        state_name: 'unchanged'
        for state_name in other_modified_state_names
    }
    for change in change_list:
        if (
            change.cmd == exp_domain.CMD_EDIT_STATE_PROPERTY and
            change.property_name != (
                exp_domain.STATE_PROPERTY_RECORDED_VOICEOVERS) and
            change.property_name != (
                exp_domain.STATE_PROPERTY_WRITTEN_TRANSLATIONS)
        ):
            state_name = change.state_name
            if state_data.get(state_name) == 'unchanged':
                state_data[state_name] = 'changed'
```

- After that, we have to update the version history of only those states which have been flagged. Also, we need to make sure that the changes made to those states using

EDIT_STATE_PROPERTY are not canceled by later changes in the same commit (similar to what is done in ExplorationDiffService). To achieve this, a new third-party library called **deepdiff** will be used which will deeply compare the old and the new state dicts:

```
for state_name, state_property in state_data.items():
    if state_property == 'changed':
        diff_dict = deepdiff.DeepDiff(
            old_states_dict[state_name], new_states_dict[state_name])
        if diff_dict != {}:
            version_history[state_name] = (
                state_domain.StateVersionHistory(
                    prev_version, state_name
                ))
```

Handing of changes in exploration metadata properties

- For this, we can check the condition: change.cmd == EDIT_EXPLORATION_PROPERTY.
- Here also, we will use deepdiff to compare two metadata dicts of explorations to make sure that the changes were not canceled by later changes in the change list.
- If so, we will record the version in the **metadata_version_history.**

```
metadata_previously_edited_on = prev_version
```

Handling of ‘move backward’ clicks

Each time the user presses this button, the backward diff data will be fetched from the backend and the changes made by the user will be shown. During the fetching of the backward diff data, an interstitial loading screen will also be shown to the user.

Handling of ‘move forward’ clicks

From the above schema, we can see that we are storing only the data required for moving backward. So a concern arises that how will we handle the ‘move forward’ clicks. The following approaches have been considered for this:

Storing the analogous ‘next committed version’ in each of the models

In this case, we will be storing the ‘next committed version’ in each of the models along with the ‘previous committed version’. This will make sure that when the user presses the ‘move forward’ button, the ‘next commit version will be fetched from the backend.

Cons of this approach:

- The updation process will become inefficient as storing and updating the models would be a costly task. During each save, if a state has been updated in that save, we will have to fetch multiple models corresponding to the exploration versions upto the 'previously

edited version number' of that state and update the value of 'next commit version' and username on all of them.

Handling this in the frontend itself

In this case, as the user clicks on the ‘move backward button’, the backward moving data will be fetched from the backend and appended in an array. Due to this, when the user clicks on the ‘forward moving button’, the cached data can be shown. For more explanation, please refer to sections Storage of the version history data in the frontend and Explanation of all the above three sections (fetching, storing and showing) with an example.

How is this approach better than the previous one?

- This approach does not tamper with the performance of the updation process of the model during each save.
- This approach will show the forward moving data quicker than the first one as the first one involves a backend call to fetch the data.

How will I make sure that I ignore the changes solely related to translations

- As explained above, we can use two extra conditions which are related to translations in order to ignore those changes.
- The conditions are:
    - change.property_name != STATE_PROPERTY_RECORDED_VOICEOVERS
    - change.property_name != STATE_PROPERTY_WRITTEN_TRANSLATIONS
- Also, we only need to check these conditions for those states which are present on both the earlier and newer versions of the explorations and were not renamed (explained in above section).

Some example cases to make sure that the above explained updation process is correct

- If a new state has been added and been renamed in the same commit, it will be handled by Handling of exploration state additions. This is because exp_versions_diff.added_state_names contain the latest names of the added states. Since the state has been created for the first time during this commit, it will not have any ‘previously_edited_on_version’ and ‘state_name_in_previous_version’.
- If a new state has been added and some properties of that state have been changed, it will be handled by Handling of exploration state additions. Since the state has been created for the first time during this commit, it will not have any ‘previously_edited_on_version’ and ‘state_name_in_previous_version’.
- If a state has been deleted and before deletion and some changes were made to the state (renames, change state properties etc.), it will be handled by Handling of exploration state removals because the state was ultimately deleted.
- If some properties of a state were changed first by (EDIT_STATE_PROPERTY) and then the state was renamed in the same commit, it will be handled by Handling of exploration

state renames because along with the version, we also need to keep track of the state name in the previous version of the exploration.

- If a state has been renamed first and then some properties of it were changed in the same commit, it will be handled by Handling of exploration state renames because of the same reason as the previous point.
- If only state properties were changed for a state (without any renaming), then it will be handled by Handling of changes in state properties (only for those states which are present in both the previous and new versions in the explorations and not have been renamed).

Handling of exploration reverts

- This task also becomes very easy if we use this method.
- The revert process can be understood in simple terms as follows:
    - Suppose the exploration version was 5 and it was reverted to version 3. Then the updated version of the exploration will be 6 with all the states and settings of the exploration being the same as they were in version 3.
- We can follow the above mechanism to revert the version history model to version 5 to version 3 by following the below pseudo algorithm:
    - Fetch the version history model corresponding to the version to which the exploration is reverted. In our case, the value is 3.
    - Create a new version history model corresponding to the new version of the exploration after the revert. In this case, the value is 6. Make the version_history of the new model to be the one in the older model.
- Following is the pseudocode for better understanding (Here, current_version is the version of the exploration before reverting. After reverting, its version would be updated to current_version + 1):

```
old_version_history_model = exp_models.ExplorationVersionHistoryModel.get(
    exploration_id + str(revert_to_version))
new_version_history_model = exp_models.ExplorationVersionHistoryModel(
    id=exploration_id + str(current_version + 1),
    exploration_id=exploration_id,
    exploration_version=current_version + 1,
    version_history=old_version_history_model.version_history
)
new_version_history_model.update_timestamps()
new_version_history_model.put()
```

Comparison between the three methods explained above

	Method 1: Without precomputation	**Method 2: Precomputation approach (a)**	**Method 3: Precomputation approach (b)**
Query	1 GET: To fetch the	1 GET: To fetch the full	1 GET: To fetch the

complexity	exploration by its id. 1 GET MULTI: To fetch the commit logs for all the versions of the exploration. 1 GET: To fetch the exploration data for the required version in order to visualize the diff between versions.	version history of all the states. 1 GET: To fetch the exploration data for the required version in order to visualize the diff between versions.	‘previous version history’ of all the states. 1 GET: To fetch the exploration data at the ‘previously edited version number’ of the active state. However, this data is mostly cached in the caching_service.
Time complexity	After fetching the data, it does another $O(N^2)$ operation to compute the version history of all the states This can be very inefficient when the value of N (versions) exceeds 1000. $O(N^2)$	After fetching the data, it just returns the data for the active state. O(1)	After fetching the data, it just returns the data for the active state. O(1)
Size of data transferred	The full version history of all the states of an exploration is computed and returned to the client each time. This makes this method undesirable as the version history of each state can have over 100k elements.	The full version history of all the states is fetched from the datastore and the data for the active state is returned to the client. It is more efficient than the first method but still returning the full version history at once is not a very desirable approach as there can be more than 100k elements in the list.	The ‘previous version history’ is fetched for each state and the data for the active state is returned to the client. This will be an ideal approach as we are just fetching a single piece of the version history.
Ease of implementation	Moderately difficult to implement.	Moderately difficult to implement.	Easier to implement all the required features (state addition, removal, renaming and changes in state properties along with

			exploration revert).

Color code:

- **Red:** Not desirable (Score = 0)
- **Yellow:** Mediocre (Score = 1)
- **Green:** Most ideal (Score = 2)

Here, we can see that the third method is indeed the best among all of them even without giving weights to the points of comparison.

Final structure of the version history

- From the above discussions, it is clear that the version history for a particular state for a particular version of an exploration will have the following structure:
    - **previously_edited_on_version**: int. The version number of the exploration on which the state was previously edited.
    - **state_name_in_previous_version**: str. The name of the state in the previously edited version. It is helpful in case of state renames.
    - **committer_id:** str.
- This structure can be represented by a new domain object called **StateVersionHistory.**

Representation of the version history of a single state

Backend: StateVersionHistory domain object

- It will be newly created in the **state_domain** (because it is related to a state rather than an exploration).
- **Attributes:**
    - **previously_edited_on_version**
    - **state_name_in_previous_version**.
    - **committer_id.**
    - These attributes are explained above
- **Functions:**
    - It will just have basic **to_dict** and **from_dict** methods.

Frontend: StateVersionHistory domain object

- **Attributes:**
    - **_previouslyEditedOnVersion:** number.
    - **_stateNameInPreviousVersion**: string.
    - **_committerUsername:** string.
- **Functions:**
    - Basic getters and setters for the above properties.
    - **From** and **to** backend dict methods.
- **Backend dict:**

```
export interface StateVersionHistoryBackendDict {
  'previously_edited_on_version': number;
  'state_name_in_previous_version': string;
  'committer_username': string;
}
```

Populating the new model for already existing explorations

Creation of a new one-off beam job

- For this, a new one-off beam job will be created which will iterate through the change list of every version of the exploration and update the version history of states present at that version of the exploration.
- The structure of the beam job will be as follows:
  - Get all the exploration models.
  - Get all the exploration commit log entry models.
  - Group the exploration models and commit log entry models for each exploration_id by using CoGroupByKey.
  - Now, we iterate over each exploration and in each iteration, we do the following:
    - Create an empty list to store the version history models.
    - Iterate over the commit logs for all versions of that particular exploration. In each iteration, do the following:
      - Get the change list from the commit log.
      - Get the old version history model from the list (version history corresponding to the previous version).
      - Update the version history according to the process explained in the section: Updation process of the old version history to get a new one during each exploration save.
      - Create a new version history model for the current version (in the iteration) with the updated version history. If the model already exists, then we can update that model itself. This can happen when the beam job has been run in the past and could not finish properly.
      - Append the model in the version history models list.
  - After getting all the models, we will save the models into the datastore using ndb_io.PutModels().

Things to check while running the beam job

- There are commit logs present for every version of the exploration i.e. if the number of versions is 'v', then there should be 'v' commit log models. For this, an audit job will be run on the server. If an exploration does not meet this criteria, then we cannot calculate the proper version history of states for this exploration and hence we can ignore that exploration while running the main beam job.

- If the change command is ‘add_state’, ‘rename_state’ and ‘delete_state’, we don’t have to check anything and can move on to the updation process.
- However, for the change command ‘edit_state_property’, we have to check for the following individual property names:
    - If the property name is **‘written_translations’** or **‘recorded_voiceovers’**, then we must ignore them and do not record the changes as they are related to translations.
    - If the property name is **‘content_ids_to_audio_translations’**, **‘widget_handlers’**, **‘widget_sticky’**, **‘gadget_visibility’** or **‘gadget_customization_args’**, we must not record their changes because these are deprecated properties.
- The computation will not be affected by deprecated interactions, customization args etc.. However, the version history will be calculated only upto exploration versions having states schema version greater than or equal to feconf.EARLIEST_SUPPORTED_STATE_SCHEMA_VERSION. In versions less than this value, we will show an information message to the user notifying that “Further version history could not be calculated due to outdated state schema version”.

Since this job will just be computing the version numbers on which the states were edited rather than the actual change, there are no more concerns left to be discussed about an exploration being “valid” while running the job.

Handling the cases where the version history cannot be calculated

- This can happen when the exploration does not have commit logs for all of its versions (i.e. some commit logs are missing).
- In this case, the version history for that particular exploration cannot be computed and hence no version history models would exist for the corresponding exploration.
- From the below section Fetching of the version history data from the backend, we can see that the initial fetching of the data will happen when the page loads for the first time. Now, if the model does not exist for a particular exploration, the backend api will throw an error and we can catch that error in the frontend. After catching the error, an information message will be shown to the user notifying that “Due to missing commit logs, the commit history of the exploration can’t be explored.”

What if the beam job fails midway?

- If it fails, we can check the error logs to understand why the job failed and for which exploration id and version.
- If the failure is due to “bad code” in the beam job, then it will be rectified and the job will be run again.
- Otherwise, the job will be run again without any changes.
- While running, the job will first check if the version history model for a particular version of the model already exists. If so, then the job will just use the existing model and not create a new one. If not, then the job will create the version history model from scratch.

Fetching of the version history data from the backend

What data we will be fetching during each press of the 'move backward' button

For a given state name and a given version of the exploration, we will make a call to the backend to fetch the following data:

- Previous version history at the given version of the given state name.
- State dict at the previously edited version number of the state.
- **DATASTORE CALLS: 3 : GET**
    - One for fetching the version history model.
    - One for fetching the exploration data at the previously edited version number. However, this data is mostly cached by the caching_services.
    - One for fetching the commit log entry model so that we can get the user id of the user who committed at the previously edited version number.

The initial fetching of the data will take place inside the **initStateEditor** function of the exploration editor tab component.

After that, the subsequent fetching of version history data will happen each time the user clicks on the **'move backward button'**.

Structure of the backend response

```
interface VersionHistoryBackendResponse {
  'version_history': StateVersionHistoryBackendDict;
  'state_dict_in_previous_version': StateBackendDict;
}
```

- Here, the StateVersionHistoryBackendDict is explained at [Frontend: StateVersionHistory domain object]().

Structure of the backend handler

- **URL:** /version_history/<exploration_id>/<version>/<state_name>
- **URL Params:**
    - **exploration_id:** The id of the exploration.
    - **version:** Version of the exploration for which we want to fetch the version history.
    - **state_name:** Name of the state for which we want to fetch the version history.
- **Pseudo algorithm:**
    - Fetch the version history model for the given version.

```
version_history = (
            exp_fetchers.get_exploration_version_history(
                exploration_id, version))
```

- Get the “previously_edited_on_version” value from the fetched version history model for the given state. Let’s call it **exp_version_to_fetch.**

```
exp_version = (
            version_history[state_name].previously_edited_on_version)
```

- If the value of **exp_version_to_fetch** is None (i.e. The state has reached the end of its version history), we return the response in the following format:
    - ‘version_history’: version_history[state_name].to_dict()
    - ‘state_dict_in_previous_version’: None

```
        if exp_version is None
            response.update({
                'version_history': version_history[state_name].to_dict(),
                'state_dict_in_previous_version': None
            })
```

- If it is not None, then we fetch the state dict for the given state in the “previously edited version number” of the state and then return the response as follows:

```
            exploration = exp_fetchers.get_exploration_by_id(
                exploration_id, version=exp_version)
            state_name_in_previous_version = (
                version_history[state_name].state_name_in_previous_version

            response.update({
                'version_history': version_history[state_name].to_dict(),
                'state_dict_in_previous_version':
exploration.states[state_name_in_previous_version].to_dict
            })
```

Structure of the backend api service

- It will have a function called **fetchVersionHistory** which will make a request to the backend to fetch the version history for the given state and the given version.
- The return value of this function will have the following structure:

```
interface VersionHistoryResponse {
  versionHistory: StateVersionHistory;
  stateInPreviousVersion: State;
}
```

If the backend API fails for some reason

- When the backend api fails, the rest of the exploration editor page will not become unusable and will work perfectly fine.

- Just the “Latest commit by …” link will be hidden from the user for that particular exploration. The rest of the page will not be affected.

Data structure to store the fetched version histories in the frontend

- We will be using three arrays to store the fetched version numbers, state data and committer data respectively as the user keeps going backward over the version history of a state.
- The arrays will be reset (made empty) each time the active state is changed.
- As the user presses the backward moving button, the previous version history will be fetched from the backend and appended into the end of the arrays.
- As the user presses the forward moving button, the diff data will be shown by the cached values as they will already be stored in the array.

Storage of the version history data in the frontend

For this, a new service will be created called **VersionHistoryService**.

It will have the following attributes:

- **_latestVersionOfExploration:** number.
    - The latest version of the exploration.

- **_fetchedVersionNumbers:** number[].
    - It will store the version numbers from the version history of a state as we keep fetching them one by one as per requirements.
    - It will be initialized as an empty array.
    - This array will be sorted in decreasing order as we insert the versions from the version history of a state.

- **_fetchedStateData**: State[].
    - It will store the state data from the version history of a state as we keep fetching them one by one as per requirements.
    - It will be initialized as an empty array.

- **_fetchedCommitterData:** string[].
    - It will store the committer usernames for different versions in the version history of a state as we keep fetching them one by one as per requirements.
    - It will be initialized as an empty array.

- **_currentPositionInVersionHistoryList:** number.
    - It will be an index pointing to the version in the_fetchedVersionHistory list upto which the user has explored till now.

    - It will be incremented when the user presses the ‘move backward’ button and decremented when the user presses the ‘move forward’ button.
    - For example, if the value of _fetchedVersionHistory is [5, 4, 3, 2, 1] and the user is currently viewing the diff between versions 2 and 3, the value of _currentPositionInVersionHistoryList will be 3 (pointing to the version 2).

It will have the following functions:

- Basic getters and setters for the above mentioned properties.

- Functions for decrementing and incrementing the value of _currentPositionInVersionHistoryList.

- **init(version: number)**
    - It will initialize the value of _latestVersionOfExploration to the latest exploration version.
    - It will be run inside the **initExplorationPage** function of the exploration **editor page component.**

- **reset():**
    - It will reset the _currentPositionInVersionHistoryList to null and the values of the above mentioned arrays to be empty arrays.
    - It will be run each time the function **initStateEditor** of **exploration editor tab component** runs. Hence it will run when the page first loads, each time the active state changes and also when some changes are saved onto the exploration.

- **shouldFetchNewData():**
    - It will return a boolean value which indicates whether the older version history data must be fetched or it is already cached.
    - Suppose that the user has been exploring the version history upto some versions and then closed the modal. When they open the modal again, the data upto that particular version history don’t need to be fetched as they are already cached. The data will be fetched again when we reach the end of the cached data.
    - The structure of the function is explained below. If the _currentPositionInVersionHistoryList is not pointing to the end of the list, then no need to fetch new data as we can use the cached ones.

```
  shouldFetchNewData(): boolean {
    if (this._currentPositionInVersionHistoryList <
this._fetchedVersionNumbers.length - 2) {
      return false;
    }
```

```
    return true;
  }
```

- **insertVersionHistoryData(version: number, stateData: State, committerUsername: string):**
    - It will push the given data into the respective lists.
    - ‘version’ will be pushed to _fetchedVersionNumbers list and so on.

*** Note:**

- If the _currentPositionInVersionHistoryList at any time is v, then:
    - The backward diff data is given by the versions _fetchedVersionNumbers[v] and _fetchedVersionNumbers[v + 1].
    - The forward diff data is given by the versions _fetchedVersionNumbers[v - 1] and _fetchedVersionNumbers[v - 2].

- **canShowBackwardDiffData()**
    - Return true if there are more versions left to be shown in the version history data.
    - It checks the following conditions:
        - _currentPositionInVersionHistoryList >= 0 (i.e. it should not be null).
        - _currentPositionInVersionHistoryList < _fetchedVersionNumbers.length - 1 (This means that we have not reached the end of the version history).

- **getBackwardDiffData()**
    - Returns the diff data required to show the changes in the previous commit.
    - The backward diff data is the diff data between versions **_fetchedVersionNumbers[_currentPositionInVersionHistoryList]** and **_fetchedVersionNumbers[_currentPositionInVersionHistoryList + 1].**
    - It returns the data in the following format:
        - oldState: State.
        - newState: State.
        - oldVersionNumber: number.
        - newVersionNumber: number;
        - committerUsername: string;

```
  getBackwardDiffData(): DiffData {
    return {
      oldState:
this._fetchedStateData[this._currentPositionInVersionHistoryList + 1],
      newState:
this._fetchedStateData[this._currentPositionInVersionHistoryList],
      oldVersionNumber: (
```

```
this._fetchedVersionNumbers[this._currentPositionInVersionHistoryList +
1]),
      newVersionNumber: (

this._fetchedVersionNumbers[this._currentPositionInVersionHistoryList]),
      committerUsername: (

this._fetchedCommitterData[this._currentPositionInVersionHistoryList + 1])
    };
  }
```

- **canShowForwardDiffData()**
  - It returns true if the following conditions are true:
    - _currentPositionInVersionHistoryList >= 2
    - _currentPositionInVersionHistoryList < _fetchedVersionNumbers.length.

- **getForwardDiffData()**
  - It will be similar to the function **getBackwardDiffData.**

```
getForwardDiffData(): DiffData {
    return {
      oldState:
this._fetchedStateData[this._currentPositionInVersionHistoryList - 1],
      newState:
this._fetchedStateData[this._currentPositionInVersionHistoryList - 2],
      oldVersionNumber: (

this._fetchedVersionNumbers[this._currentPositionInVersionHistoryList -
1]),
      newVersionNumber: (

this._fetchedVersionNumbers[this._currentPositionInVersionHistoryList -
2]),
      committerUsername: (

this._fetchedCommitterData[this._currentPositionInVersionHistoryList - 1])
    };
  }
```

Showing the diff data between versions to the user

- For this, a new modal component will be created.

- It will be similar to the StateDiffModal.
- **Properties**:
    - committerUsername: string.
        - The username of the committer in the current diff. If we are showing the diff between versions v and v + 1, then its value will be the username of the user who committed those changes from v -> v + 1.
    - previousVersionNumber: string.
        - If we are viewing diff between v and v + 1, then its value will be v.
    - newState: State.
        - The state data for version v + 1.
    - oldState: State.
        - The state data for version v.
    - oldStateName: string.
    - newStateName: string.
- **Functions:**
    - **updateLeftPane:**
        - Converts the oldState dict into yaml using the newly created Yaml Service and stores the result.
    - **updateRightPane:**
        - Converts the newState dict into yaml using the newly created Yaml Service and stores the result.
    - **fetchPreviousVersionHistory:**
        - Fetches the previous version history for the active state using the newly created backend api service.
        - After fetching the data, it updates the data in the **VersionHistoryService** and increments the _currentPositionInVersionHistoryList by 1.
        - It also checks if new data needs to be fetched by using the **VersionHistoryService.shouldFetchNewData** method. If not, then it just increments the _currentPositionInVersionHistoryList.
    - **ngOnInit:**
        - This function is called once when the modal is opened for the first time.
        - Calls updateLeftPane and updateRightPane functions and renders the diff between the versions.
        - After that, it calls **fetchPreviousVersionHistory** to fetch the previous version history of the state.
    - **onClickMoveBackwardButton:**
        - Gets the backward diff data from VersionHistoryService and updates the properties such as newState, oldState, newStateName, oldStateName etc..
        - After that it will call updateLeftPane and updateRightPane to update the modal with the new diff data.
        - Fetches the previous version history using the **fetchPreviousVersionHistory** function.

- **onClickMoveForwardButton:**
    - It is similar to the previous function.
    - The differences are that this function gets the forward diff data and decrements the _currentPositionInVersionHistoryList after updating the modal with the new data.
    - Also, it does not need to fetch new data as forward diff data will always be present beforehand.

Explanation of all the above three sections (fetching, storing and showing) with an example

The above three sections do not explain much about how they are interconnected. However, they will become more clear after this example.

Suppose that we have an exploration with states a and b. The latest version of the exploration is 6 and the version history of the states a and b look like following:

- a: [4, 3, 2, 1]
- b: [5]

Initially, the values of the properties of version history service will be as follows:

_latestVersionOfExploration	null
_currentPositionInVersionHistoryList	0
_fetchedVersionNumbers	[ ]

(The other properties are not shown for the sake of simplicity. But they will also be updated whenever _fetchedVersionNumbers get updated).

The exploration editor page loads for the first time

Firstly, the value of _latestVersionOfExploration will be set to the latest version of the exploration inside **initExplorationPage.**

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	0
_fetchedVersionNumbers	[ ]

The state editor gets initialized

Here, the **initStateEditor** will be called and the lists will be initialized with the data of the latest version of the exploration.

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	0
_fetchedVersionNumbers	[6]

Again inside the **initStateEditor,** the first version history for the active state (a) will be fetched and the data will be updated.

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	0
_fetchedVersionNumbers	[6, 4]

At this point, the user will be able to see the annotation “Latest commit by XXX at version 4”.

The user clicks on the annotation

The backward diff data will be fetched from version history service and the modal will be shown to the user (diff between versions 4 -> 6).

As soon as the modal shows up, the previous version history will be fetched inside the ngOnInit function and the data will be updated as follows:

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	1
_fetchedVersionNumbers	[6, 4, 3]

At this point, the user will be able to see the annotation “Previous commit by XXX at version 3”.

The user clicks on the “Previous commit by …” button

Similar to the previous step, the diff data will be fetched from the version history service and the modal data will be updated (Now, diff between 3 -> 4 will be shown).
Also, the previous version history will be fetched and after the data is received, the properties will be updated as follows:

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	2
_fetchedVersionNumbers	[6, 4, 3, 2]

At this stage, the button “Next commit by XXX at version 4” will be visible which will help the user to move forward in the version history.

Now, if the user keeps going backward, the same process will be repeated until they reach the end of the version history. At that point, only the “Next commit by …” button will be visible.

The user clicks on the “Next commit by ….” button

In this case, the forward diff data will be fetched from the version history service and the modal data will be updated to show the diff between versions 4 -> 6. Also, the _currentPositionInVersionHistoryList is decremented.

_latestVersionOfExploration	6
_currentPositionInVersionHistoryList	1
_fetchedVersionNumbers	[6, 4, 3, 2]

Now, the “Next commit by ….” button vanishes because the value of _currentPositionInVersionHistoryList has become 1 now. Only the “Previous commit by XXX at version 3” will be visible.

The user closes the modal

Closing the modal will set the value of _currentPositionInVersionHistoryList to 0 and the user can see the annotation “Latest commit by XXX at version 4” again.

However, the already fetched values will not be deleted because the user might open the version history again and then we would not have to fetch them again.

The user changes the active state

On changing the active state, the **initStateEditor** will be called again and the steps explained in [The state editor gets initialized](#) will be repeated.

Performance considerations

- Tested on oppia development server.
- Operating system: Ubuntu 20.04
- Browser: Chrome Version 96.0.4664.110 (Official Build) (64-bit)

Without any precomputation

Attempt	Result (time taken in seconds to fetch the full version history)
1	0.622

2	0.577
3	0.633
4	0.591
5	0.584

- In this case, the average time taken comes out to be 0.602 seconds.
- Hence, average time taken for fetching versions history list of all states = 0.59 seconds.
- Now, we need to consider the average time taken by CompareVersionsService to fetch the diff data and return it.

Attempt	Result (time taken in seconds to fetch diff data)
1	0.079
2	0.095
3	0.103
4	0.161
5	0.11

- Hence, average time taken for fetching diff data between versions using CompareVersionsService = 0.1096 seconds.

With precomputation (Precomputation approach (b))

In this approach, we will be fetching the 'previously edited version number' along with the state dict in the previous version (for the active state) as explained in Fetching of the version history data from the backend.

Attempt	Time taken to fetch the previous version history (in ms)
1	46
2	50
3	55
4	56
5	86

Hence, the average time taken comes out to be 58.6 ms or 0.06 seconds (approx.). These values also depend upon the capability of the development server and the network latency.

However, by looking at the time taken by both methods on the same development server, we can say that precomputation will save us a lot of time.

Launch plan for Subproject (B)

- The new model will be created along with implementation of its lifecycle (creation, updation and deletion).
    - The updation process (explained in [Updation process of the old version history to get a new one during each exploration save]()) will take place if and only if the version history model exists for the previous exploration version (i.e. if the exploration is updated from v -> v + 1, then the updated version history model will only be created for version v + 1 if the version history model for version v exists). For this, an if-condition will be added to check if the model is not None.
    - Hence, at this stage, the feature will only work for explorations which are newly created. For older explorations, the version history models will not be present till now, so the updation process will not take place for them (when they are being saved).
- During the first release, the new model and its lifecycle implementation will be merged and released.
- After the new model is implemented and merged, the beam job will be run on the production server (in maintenance mode) to populate the data for already existing models.
    - This will be a relatively longer process as the job might fail and might take multiple iterations of running it in order for it to fill all the models correctly.
    - While running the beam job, we will face the situation where the version history models will be available for some explorations and not for others.
        - To face that, while running, the job will also fetch the existing version history models for different versions of the exploration.
        - If the version history models for a particular exporation already exist, then the job will keep the models as it is and then push it back. If not, then it will create the models from scratch. It is explained in the section [What if the beam job fails midway?]().
- After the beam job has finished properly, we will be sure that the version history for all the explorations (both newer and older ones) are fully up to date.
    - Now the updation process (while saving some changes to the exploration) will happen for all the explorations as the version history model of the previous version of the exploration will exist for all of them.
    - Hence, after running the beam job, we can be sure that the version histories will remain updated forever (as the lifecycle of the model is merged beforehand).
- During the second release, the frontend will be allowed to show the feature to the users

and make the requests to the backend to fetch the version history data.

Implementation of feature flags

- There will be a feature flag in the frontend called ALLOW_VERSION_HISTORY_NAVIGATION which will prevent the frontend from fetching the version history data from the backend. Its value will be false by default and will be made true once all the data in the backend becomes stable and fully updated (i.e during the second release).

Third-Party Libraries

No.	Third-party library name and version	Link to third-party library	Why it is needed	License[2] (if third-party library)
1	js-yaml (4.1.0)	js-yaml	It is needed to convert metadata dict into yaml from the frontend itself. Currently, it is done by sending a request to the backend and receiving the yaml representation of the dict. However, this approach is asynchronous and takes some time to finish. Also, if the backend api fails sometime, then the version diff visualization would appear broken. Hence, using the frontend library would sweep the cons of using the backend approach.	MIT License
2	deepdiff	deepdiff	It is needed to deeply compare two dictionaries.	MIT License

Impact on Other Oppia Teams

This project will not impact other oppia teams.

Key High-Level and Architectural Decisions

Subproject(a)

For converting metadata dict to yaml string in Subproject(a), the following alternatives were considered for converting the state or metadata dict into yaml string:

1. Backend approach: Currently, the state dict is converted into yaml string by sending a POST request to the backend and doing the conversion in the backend.
2. Frontend approach: This is the newly proposed solution in this document. It will use a party library called ‘yaml’ to do the conversion.

---

[2]Note: Oppia can only use third-party libraries that are compatible with our Apache 2.0 license. If you're unsure about license compatibility, talk to a platform TL.

Among these, I believe that alternative 2 is a better approach because the backend approach is asynchronous and will take more time to complete than the frontend approach. Also, if the backend api fails for some reason, the diff visualization will appear broken. The approaches have been compared below:

	Alternative 1	**Alternative 2**
Performance	Relatively slower than the second method as it is asynchronous and involves sending a request to the server and getting the response.	Relatively faster than the first method as it is synchronous and everything is getting done in the frontend itself.
Probability of failure	If the backend api fails for some reason, then the diff visualization would appear broken.	Since everything is done on the frontend, it is far less probable to fail.
Usage of third party libraries	No new third party library is used.	A new third party library will be introduced called ‘yaml’.

Subproject (b)

For this, the following methods were considered:

- Method 1: Without precomputation
- Method 2: Precomputation approach (a)
- Method 3: Precomputation approach (b) (Efficient and scalable approach)

Among them, I believe that method 3 will be the most desirable and beneficial approach. The comparison among the methods have been shown in Comparison between the three methods explained above.

Risks and mitigations

There will be no security or reliability risks introduced by implementing this solution for both subprojects (a) and (b).

Implementation Approach

Storage Model Layer Changes

ExplroationVersionHistoryModel

It will store the ‘previous version history’ of all the states present in a particular version of the exploration. It is explained in detail in the section [Method 3: Precomputation approach (b) (Efficient and scalable approach)](#).

- **Schema:**
  - **exploration_id:** str. The id of the corresponding exploration.
  - **exploration_version:** int. The version number of the exploration.
  - **version_history**: dict. A mapping of the state names of an exploration in its current version and their version history.
    - **Key:** str. The name of the state.
    - **Value:** dict.
      - The structure of the dict will be as follows:
        - 'previously_edited_on_version': int
        - 'state_name_in_previous_version': str
        - 'committer_id': str
    - **committer_id** is the id of the user who committed the changes from version = 'previously_edited_on_version' to version = 'previously_edited_on_version' + 1.
- **Lifecycle of the model:**
  - **[CREATION]:**
    - Create a new instance of this model whenever a new exploration is created.
    - Initialize the version_history of all the states as (None, None) at the beginning.
  - **[UPDATION]:**
    - The updation process explained in detail in the below section [Updation process of the old version history to get a new one during each exploration save](#).
  - **[DELETION]:**
    - Delete all the instances of this model when the exploration is deleted.

Domain Objects

Frontend

ExplorationMetadata domain object

It is explained in the section Frontend: ExplorationMetadata domain object.

Renaming the existing ExplorationMetadata domain object

- In the frontend, there already exists an ExplorationMetadata domain object. However, it represents only three properties i.e. id, objective and title.
- This domain object is used to represent information for searching explorations.
- Hence, it will be renamed to **ExplorationSearchResult.**

StateVersionHistory domain object

It is explained in Frontend: StateVersionHistory domain object.

Backend

StateVersionHistory domain object

It is explained in the section Backend: StateVersionHistory domain object.

ExplorationMetadata domain object

It is explained in the section Backend: ExplorationMetadata domain object.

User Flows (Controllers and Services)

User stories / tasks

In the below points, "Additional Datastore calls" means the datastore calls introduced by this project.

Subproject (A)

- **User selects two versions in the history tab**
    - **Pseudo algorithm**
        - CompareVersionsService fetches the diff data between the two selected versions.
        - The history tab component now renders the version diff visualization component providing the diff graph data.
        - At this point, the user sees the diff graph with the nodes being the exploration states along with the metadata nodes.
    - **URL Endpoint**

        - /explorehandler/init/<exploration_id>?v=<version>
    - **Handler**
        - ExplorationHandler in reader.py.
    - **Additional Datastore calls**
        - None
- **User clicks on the button "Changes in exploration metadata"**
    - **Pseudo algorithm**
        - This will open the newly created metadata diff modal which will show the metadata changes between the two selected versions.
    - **URL Endpoint**
        - None
    - **Handler**
        - None
    - **Additional Datastore calls**
        - None

Subproject (B)

- **User opens the exploration editor page for the first time or changes the active state from the exploration graph.**
    - **Pseudo algorithm**
        - **initStateEditor** sends a request to fetch the previous version history for the active state.
        - The ExplorationVersionHistoryModel is fetched for the given exploration id and the latest version of the exploration.
        - Along with that, the exploration data for the previously edited version of the state is fetched.
        - The previous version history along with the state dict for the previously edited version is sent to the frontend.
    - **URL Endpoint**
        - /version_history/<exploration_id>/<version>/<state_name>
    - **Handler**
        - ExplorationVersionHistoryHandler.
    - **Additional Datastore calls**
        - [**SYNC**]
            - **get**: 2 (One for fetching the previous version history and other for fetching the exploration data at the previously edited version).

- **User clicks on the link "Latest/Previous commit by XXX at version YYY" (similar to previous case)**
    - **Pseudo algorithm**
        - The diff data between the versions YYY and YYY + 1 will be shown.
        - Also, a request will be sent to fetch the previous version history of the active state.

    - **URL Endpoint**
        - /version_history/<exploration_id>/<version>/<state_name>
    - **Handler**
        - ExplorationVersionHistoryHandler.
    - **Additional Datastore calls**
        - [**SYNC**]
            - **get**: 2 (One for fetching the previous version history and other for fetching the exploration data at the previously edited version).

- **User saves some changes in the exploration.**
    - **Pseudo algorithm**
        - The exploration is saved as per the current system.
        - Additionally, the version history for the exploration states is updated according to the method explained in the section Updation process of the old version history to get a new one during each exploration save.
        - After the save is successful, the **initStateEditor** function runs again and fetches the updated 'previous version history' of the active state.
    - **URL Endpoint**
        - /createhandler/data/<exploration_id>?apply_draft=<apply_draft>
        - /version_history/<exploration_id>/<version>/<state_name>
    - **Handler**
        - ExplorationHandler in editor.py (for saving the exploration)
        - ExplorationVersionHistoryHandler (for fetching the updated version history data)
    - **Additional Datastore calls**
        - [**SYNC**] **get**: 1 (for getting the old version history model)
        - **[SYNC] put**: 1 (for saving the updated version history model)

Modifications in backend handlers

Changes in ExplorationHandler in reader.py

- Its response dict will be updated to include a new property called **exploration_metadata.**
- It is explained in detail in the section Modification of the return value of ExplorationHandler.

Creation of a new ExplorationVersionHistoryHandler for fetching the version history data

- It will be newly created for this project.
- It is explained in detail in the section Fetching of the version history data from the backend.

Web frontend changes

- Creation of a new button saying “Changes in exploration metadata” in the history tab component. When clicked, this button will open a modal showing the changes made in the exploration metadata between the two selected versions. It is explained in the section Changes in History Tab Component.
- Creation of a new Metadata diff modal to show the diff in exploration metadata. It is explained in the section Creation of a new Metadata Diff Modal Component.
- Creation of a new service to store the fetched version history data. It is explained in detail in the section Storage of the version history data in the frontend.
- Creation of a new modal component to view the diff data between the states as explained in Showing the diff data between versions to the user.

Testing Plan

E2e testing plan

#	Test name	Initial setup step	Step	Expectation
1.	Lesson creators can see the changes in exploration metadata between two selected versions in the history tab.	Login and open the exploration editor page.	Create an exploration and make some changes in the exploration properties (title, category etc.). Open the history tab and select the first and the last version.	The creator should see a “Metadata” node along with the other state nodes.
			Click on the “Metadata” node.	The metadata diff modal should popup and show the diff between exploration metadata between the first and the last versions.
2.	Lesson creators can see the annotation “Last edited by XXX at version YYY”.	Login and open the exploration editor page.	Create an exploration and make some changes on the initial state and save those changes. After saving, make some more changes and save them again.	The annotation should show up on the bottom right corner of the page.
			Click on the annotation.	A modal should popup showing the diff between the versions YYY and YYY + 1. Another annotation should be visible on the bottom right corner of the modal.
			Click on the annotation on the bottom right corner of the modal.	The modal data should be updated to show the diff between the previous versions.

Feature testing

Does this feature include non-trivial user-facing changes? **YES**

Implementation Plan

Milestone 1(June 13 - August 14)

Enable creators to see changes in the exploration metadata by clicking a button in the history tab. Make all the necessary backend changes (up to and including the controller layer) for users to be able to navigate through the version history of a state (and the exploration metadata) in an exploration. Also, create the backend api service in the frontend for fetching the version history data.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	Create all the new domain objects: ● ExplorationMetadata (frontend) ● ExplorationMetadata (backend) ● StateVersionHistory (frontend) ● StateVersionHistory (backend)	None	13th June	18th June
2	Create the new ExplorationStatesVersionHistoryModel and implement its lifecycle in the backend (creation, updation etc.).	1	20th June	26th June
3	Create the new beam job and run it on the server.	2	29th June	5th July (Merged and run on the server) 23rd July (Date upto which the job will create all the required data on the server correctly. I am keeping this leeway to handle things properly in case they go wrong).

4	Make the changes in the return value of **ExplorationHandler** to include the new **exploration_metadata property** and also add the new properties (v1Metadata and v2Metadata) in **CompareVersionsService's** return value. Also, create the backend test to ensure that a new metadata property added shows up during comparison.	1	1st July	6th July
5	Install the 'js-yaml' library and create the new **YamlService**. Also, create the new frontend service for caching of the fetched exploration versions.	None	3rd July	9th July
6	Make the other required changes to show the metadata diff to the user: ● Create the new metadata diff modal component. ● Create the new button in the history tab component to open the metadata diff modal.	4, 5	11th July	17th July
7	Write the e2e tests for Subproject (A)	6	19th July	25th July
8	Create the backend handler for fetching the version history data along with the backend api service in the frontend. Also, introduce the feature flag: ALLOW_VERSION_HISTORY_NAVIGATION so that the frontend does not make any request to fetch the version history data. Initial value of this flag will be false.	2	27th July	4th Aug

Milestone 2(August 25 - October 16)

Make all the other required frontend changes so that users can navigate through the version history of a state and the exploration metadata.

No.	**Description of PR / action**	**Prereq PR numbers**	**Target date for PR creation**	**Target date for PR to be merged**
9	Create the frontend service to store the fetched version history data.	2	25th August	31st August
10	Create the new modal to show the diff between different versions of the state.	9	2nd September	8th September

11	Make the other required changes in the exploration editor tab component to complete the subproject(b). Also, make the value of ALLOW_VERSION_HISTORY_NAVIGATION to be true so that the frontend can now make requests to the backend.	10	10th September	17th September
12	Write the e2e tests for subproject(b)	11	22nd September	30th September

Future Work

- I will continue to be a part of oppia in the future.