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 happy to take the default coding period (June 13 - September 12) and will be able to complete the medium sized project within this timeframe.

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 - September 12), I would be able to commit at least 3 - 3.5 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.

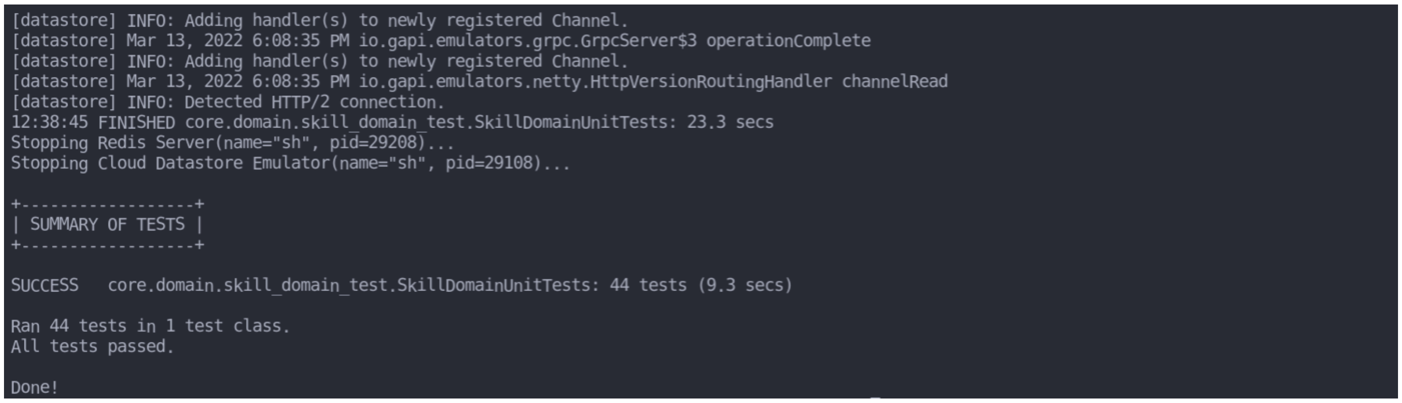
Hence, the following is my time commitment schedule for the coding period:

- **Daily**: 3 - 3.5 hours. Can be increased if required.
- **Weekly**: 21 - 24.5 hours. Can be increased if required.

Essential Prerequisites

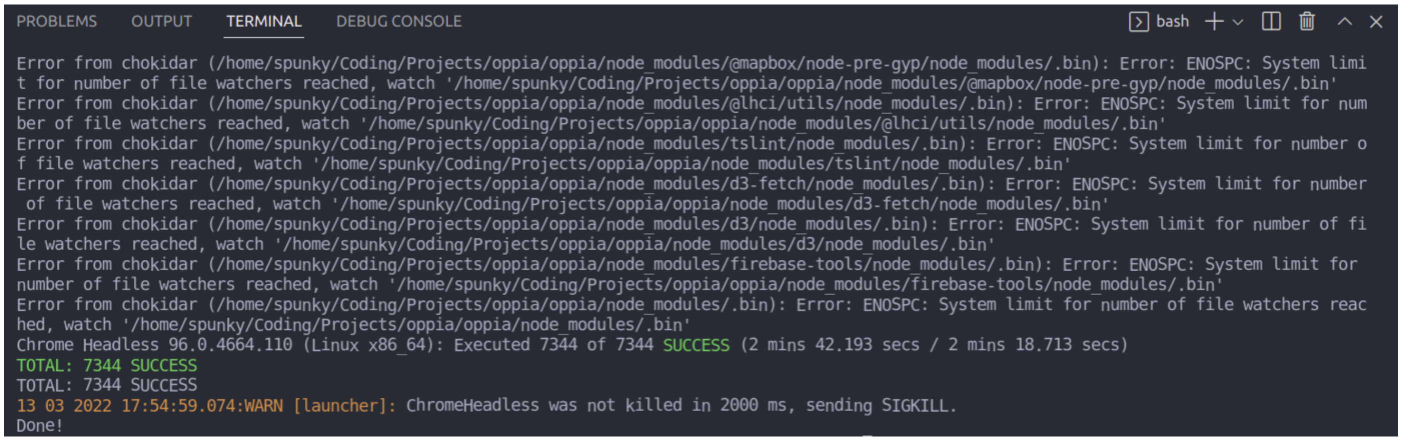
Answer the following questions (for Oppia Web GSoC contributors):

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



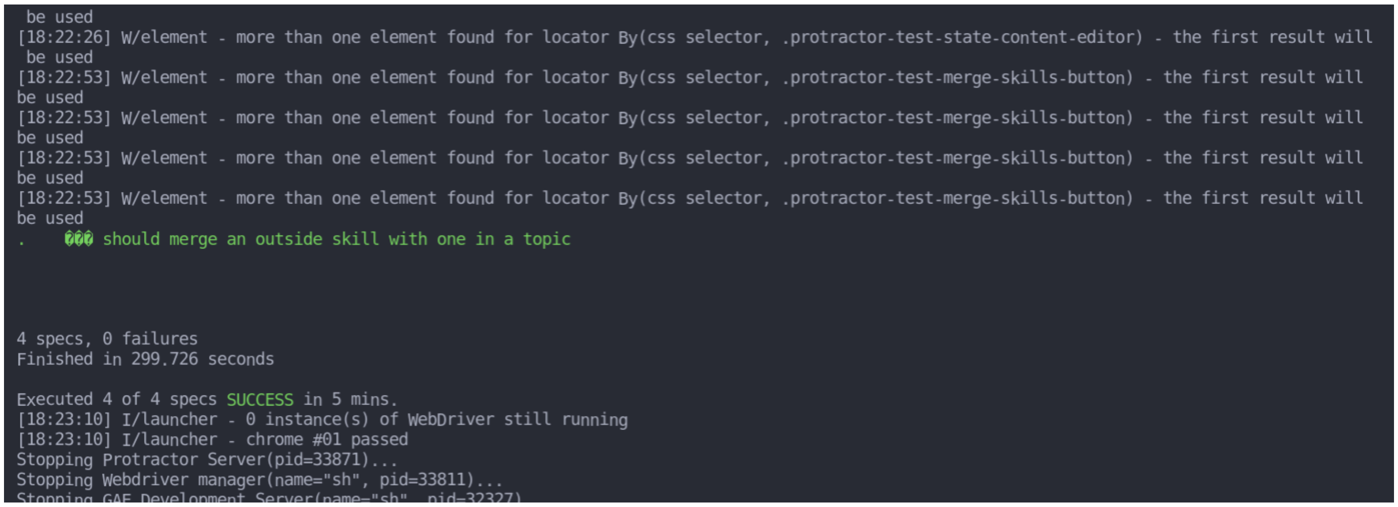
```
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Mar 13, 2022 6:08:35 PM io.gapi.emulators.grpc.GrpcServer$3 operationComplete
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Mar 13, 2022 6:08:35 PM io.gapi.emulators.netty.HttpVersionRoutingHandler 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.)



```
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, it would be a good feature for me to be able to see the changes in exploration metadata (title, goal, tags etc.) along with changes in each state card when comparing between different versions of the exploration. It would help me notice some unwanted changes made by me or any fellow collaborator of that exploration and fix them if needed. - As a lesson creator, it would be a good feature for me to be able to see the list of changes made to a state card and the details regarding who modified it. And if the exploration does not behave as expected in the future, me and my fellow collaborators would be able to spot the exact changes responsible for this and take the necessary action.
What goals do we want the solution to achieve?	- When comparing between two versions of an exploration, the user would see an extra node called "Metadata" along with the other state cards in the comparison graph. Clicking this node will open a modal similar to the state diff modal and would show the changes made in the exploration metadata between the two selected versions. - In each state card, the user would see a link at the bottom right corner of the state editor reading "Last edited 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. Here, YYY + 1 need not be the latest version of the exploration. On the bottom right corner of the modal, another link will appear reading "Previously edited by PPP at version QQQ". This process would continue until we reach the point where the state was created for the first time. At this point, the link will disappear.

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	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.	Part(a) Prototype Part(a) Mock Video
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 "Last edited 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.	Part(b) Prototype Part(b) Mock Video

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

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'** and the value of this property will be the various exploration metadata properties such as title, category, goals etc.. **URL Parameters:** exploration_id: str
2.	/explorehandler/<exploration_id>/version_history/<state_name>	GET	New	This request will be used to fetch the version history for a particular state of an exploration. **URL Parameters:** exploration_id: str state_name: str

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.	/explorehandler/<exploration_id>/version_history/<state_name>	GET	This new request is needed to fetch the version history of a particular state of an exploration.
3.	/userinfohandler	GET	This new call is needed to fetch the user info while initializing the ExplorationStatesVersionHistoryService as this property is required by the service. However, since the user info is cached once it is fetched, this request will not be made every time.

UI Screens/Components

#	ID	Description of new UI component	i18n required?	Mock/spec links	A11y requirements
1.	Additional metadata node on visualization graph.	A new node will be added into the visualization graph. On clicking it, creators can see the metadata changes made to the exploration.	No	Additional metadata node	No
2.	Metadata changes modal.	It is a new modal that shows the difference in the exploration metadata between two selected versions. It will be similar to the existing state diff modal.	No	Metadata changes modal	No
3.	"Last edited by XXX at version YYY" link	It is a new link that is placed at the bottom right corner of the page. When clicked, it will show the changes made in the state card from version YYY to YYY + 1.	No	Last edited by .... Link	No
4.	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

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
3.	The computed version history of all the states of an exploration.	The version history of all the states of an exploration in the format explained in Structure of the version history of a state.	This data would help users to navigate through the version history of a state.	No, because the username of the committer of different versions will be stored.	No	This data can only be deleted if the exploration is deleted. When the user deletes their account, this data will stay in the datastore.	Allow users to export the data.

Other Requirements

The **CompareVersionsService** should be migrated before implementing the solution for part(b). If it is not migrated by that time, then I will raise an extra PR for migrating this service.

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 the active state.

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

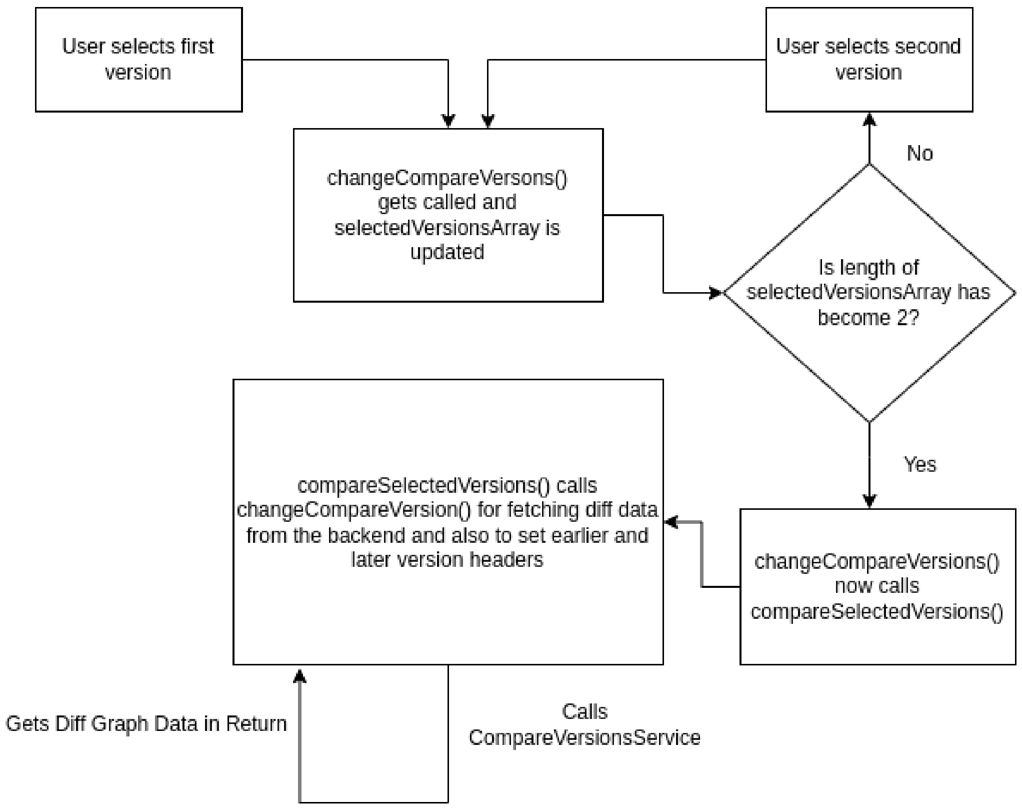
- In order to see the changes in exploration metadata in the history tab, we need to have the following steps:
    - Getting metadata information from the backend during version comparison.
    - Addition of an extra node called “Metadata” in the visualization graph along with the other state nodes.
    - When this node is clicked, show the changes in exploration metadata between the two selected versions.

Getting metadata information from the backend

Current system:

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

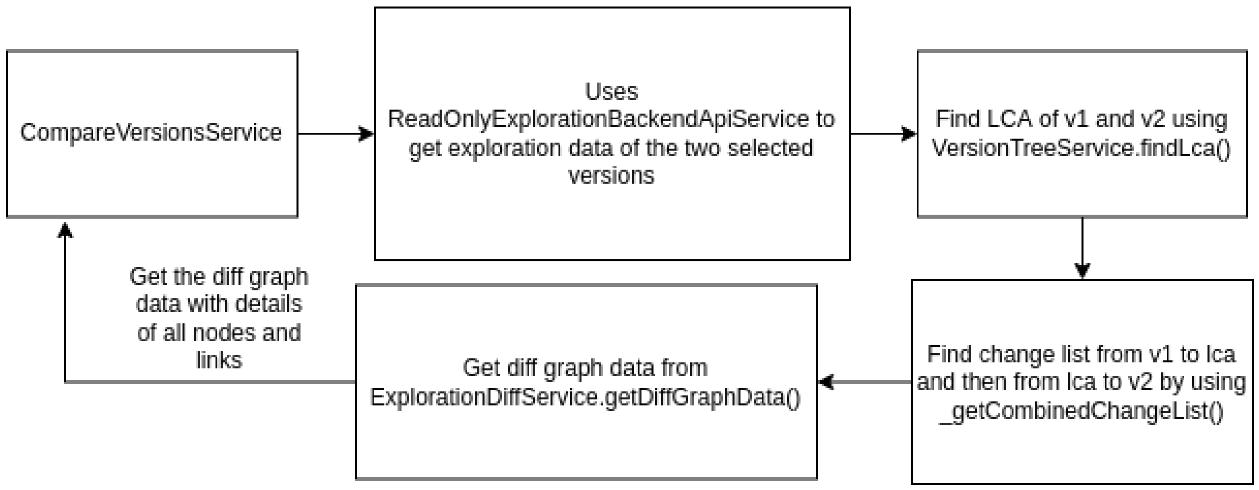
- User selects the first version: **changeSelectedVersions()** of the history tab component gets called. This function modifies the **selectedVersionsArray** to include the first selected version.
- User selects the second version: the same process happens again and **selectedVersionsArray** gets modified.
- 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)).


User selects first version
User selects second version
changeCompareVersons() gets called and selectedVersionsArray is updated
No
Is length of selectedVersionsArray has become 2?
Yes
compareSelectedVersions() calls changeCompareVersion() for fetching diff data from the backend and also to set earlier and later version headers
changeCompareVersions() now calls compareSelectedVersions()
Gets Diff Graph Data in Return
Calls CompareVersionsService

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.


CompareVersionsService
Uses ReadOnlyExplorationBackendApiService to get exploration data of the two selected versions
Find LCA of v1 and v2 using VersionTreeService.findLca()
Find change list from v1 to lca and then from lca to v2 by using _getCombinedChangeList()
Get diff graph data from ExplorationDiffService.getDiffGraphData()
Get the diff graph data with details of all nodes and links

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.

Addition of a new to_metadata_dict function in Exploration domain object

- A new function will be added in the **Exploration domain object** to convert the exploration into the metadata dict. Name of this function will be **to_metadata_dict.** The function will look like the following:

```
def to_metadata_dict(self):
    return {
        'title': self.title,
        'category': self.category,
        'objective': self.objective,
        'language_code': self.language_code,
        'tags': self.tags,
        'blurb': self.blurb,
        'author_notes': self.author_notes,
        'param_specs': self.param_specs,
        'param_changes': self.param_changes,
        'init_state_name': self.init_state_name,
        'auto_tts_enabled': self.auto_tts_enabled,
        'correctness_feedback_enabled': self.correctness_feedback_enabled
    }
```

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 **to_metadata_dict** as mentioned above.

Caching of the fetched exploration versions make the comparison faster

Current system

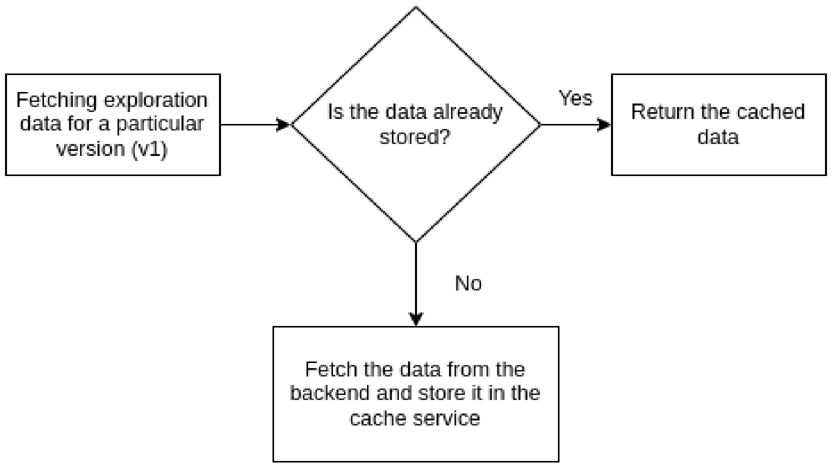
- Currently, in the **read-only-exploration-backend-api-service** there is some sort of caching which stores the latest version of exploration with a particular id.
- However, there is no version specific cache service available to store version specific exploration data. Having a service like this would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


Fetching exploration data for a particular version (v1)
Is the data already stored?
Yes
Return the cached data
No
Fetch the data from the backend and store it in the cache service

- 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

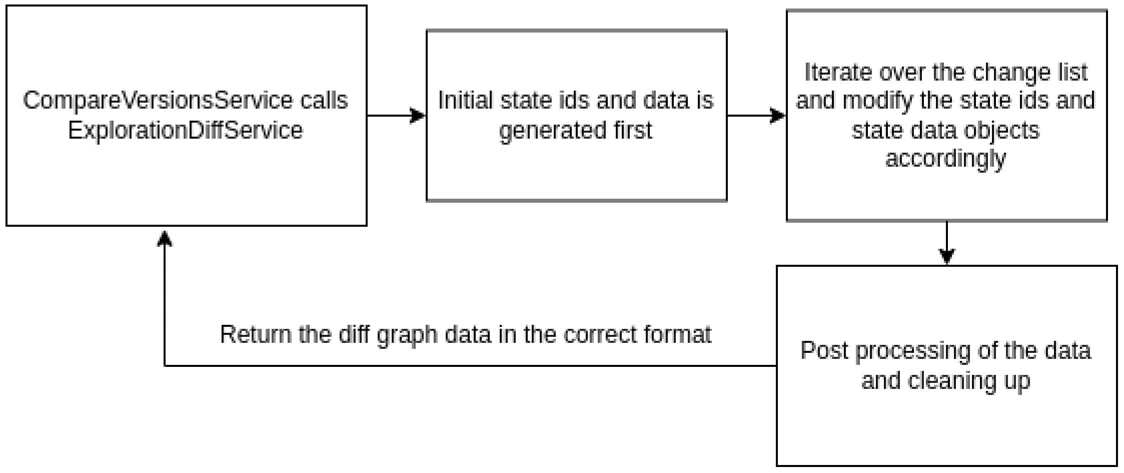
Addition of an extra node called "Metadata" on the diff graph data and recording the changes to exploration metadata

Current system

Steps taking place in ExplorationDiffService.getDiffGraphData

- When **getDiffGraphData** is called, it generates the initial state ids and data by calling **_generateInitialStateIdsAndData**.
- In the next step, it iterates over the change list and modifies the initially generated data accordingly.
- In the last step, we generate the diff graph data from the modified state ids and state data objects resulting after the second step. For this, the function **_postProcessStateIdsAndData** is called.
- In this process, the data is cleaned first by ignoring changes that were canceled by later changes and also by deleting the states which were not present in both v1 and v2. After

all this, the processed data is returned in the correct format. In this way, we get the whole diff graph data which can be used in the version diff visualization component.


CompareVersionsService calls ExplorationDiffService
Initial state ids and data is generated first
Iterate over the change list and modify the state ids and state data objects accordingly
Post processing of the data and cleaning up
Return the diff graph data in the correct format

- As explained, when **ExplorationDiffService.getDiffGraphData()** is called, it first generates the initial state ids and data based on the v1States (states of the earlier version) by calling the **_generateStateIdsAndData()** function. We can use this logic to generate the “Metadata” node at the beginning.

- Also, in the **_getDiffData()** function of **ExplorationDiffService**, we iterate over the change list to update the initially generated state ids and data. While iterating over the change list, we can introduce a new condition to update the state data of the newly added “Metadata” node.

Required changes

Generating metadata node data at the beginning

- The function **_generateInitialStateIdsAndData** will be slightly modified to initialize the metadata node at the beginning.
- Since metadata cannot be considered as an actual state, we need to rename some functions and variables to be more conceptually correct. Some of the variables and properties that will be renamed are:

Old	**New**
newestStateName	newestNodeName
originalStateName	originalNodeName
stateProperty	nodeProperty
stateIds	nodeIds

stateData	nodesData
_generateInitialStateIdsAndData	_generateInitialDiffGraphData

- Since the metadata node will be created at the beginning, **its node id (with respect to diff graph data) will always be 1**. We will make use of this logic (node id of metadata = 1) later.
- The **_generateInitialDiffGraphData()** function will look like the following after modification:

```
_generateInitialDiffGraphData (statesDict) {
  let result = (...Initial result with Metadata details added already...)

  Now, add the remaining states in the result object
  for (let stateName in statesDict) {
    let nodeId = this._generateNewId();
    result.nodesData[nodeId] = {
      newestNodeName: stateName,
      originalNodeName: stateName,
      nodeProperty: this.STATE_PROPERTY_UNCHANGED
    };
    result.nodeIds[nodeName] = nodeId;
  }

  At this stage, we have the metadata node along with other state nodes.
  return result;
}
```

Identifying changes from the changeList that affect the metadata

- Metadata can also be called ‘exploration property’. While iterating over the change list in **_getDiffData()** function, a new if-condition will be added where it will check if (change.cmd === ‘edit_exploration_property’). If this condition returns true, we will change the **nodeProperty** of the “Metadata” node from STATE_PROPERTY_UNCHANGED to STATE_PROPERTY_CHANGED as follows:

```

    --------- Other conditions ---------
    } else if (change.cmd === 'edit_exploration_property') {
      if (nodesData[nodeIds.Metadata].nodeProperty ===
          this.STATE_PROPERTY_UNCHANGED) {
        nodesData[nodeIds.Metadata].nodeProperty = (
          this.STATE_PROPERTY_CHANGED);
          }
        }
      --------- Other conditions ---------
```

The changes can be tabulated as follows:

File name	Function name	List of changes
exploration-diff.service.ts	_generateInitialDiffGraphData	Before adding initial data about other state nodes, add the metadata node with the initial data. Also, rename the function as discussed above.
exploration-diff.service.ts	_getDiffData()	While iterating over the change list, add a new condition to check if the change command is equal to 'edit_exploration_property' and modify the nodeData accordingly.

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.

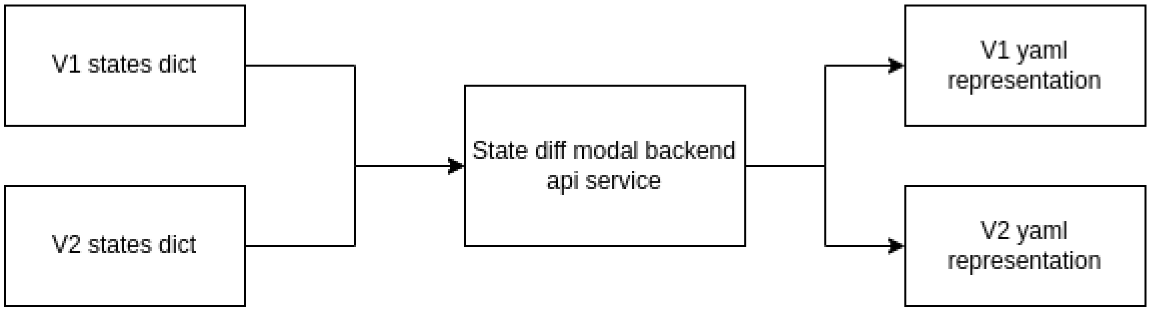
Create a new metadata diff modal component

- A new component will be created called metadata-diff-modal.component.ts for showing the diff data in the exploration metadata.

Conversion of state or metadata dict into yaml string

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.

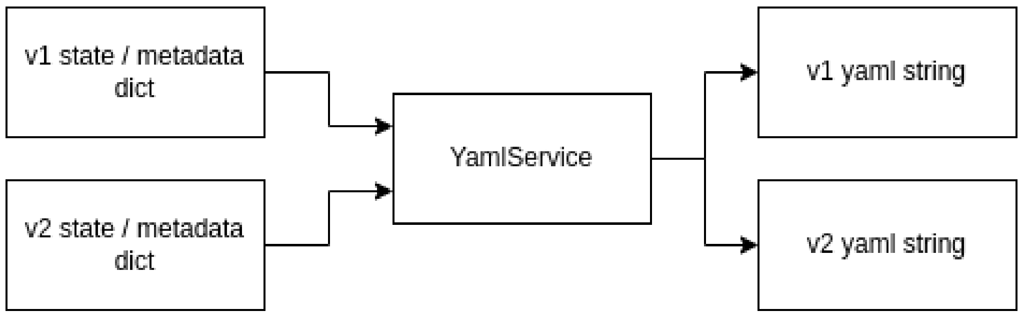


Flaws in the current system

- 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 ‘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.
state-diff-modal.component.ts		Remove the backend approach and use the YamlService to do the conversion.
Metadata-diff-modal.component.ts (newly created by this solution)		Use the YamlService to convert metadata_dict into yaml string.

Changes in version diff visualization component

- Now, we have two modals available to show diff data. One is the newly created metadata diff modal and another is the existing state diff modal.
- Hence, we need to introduce a conditional logic on the click handler of graph nodes to open state diff modal and metadata diff modal.
- As it was explained in the **section** **Addition of an extra node called “Metadata” on the diff graph data and recording the changes to exploration metadata** that the Metadata node will be added at the beginning and hence, its node id with respect to diff graph will always be 1.
- We will make use of this logic to achieve conditional opening of the mentioned modals. If the node id is 1, then open the metadata diff modal. Else, open state diff modal.

The changes can be tabulated as follows:

File name	**Function name**	**List of changes**
version-diff-visualization.component.ts	onClickStateInDiffGraph	Add a conditional logic to check if state id is 1. If true, then open the metadata-diff

		modal. Otherwise, open the state-diff modal.

Extension of this functionality for a newly added metadata field

- For achieving this, we need to make sure that the newly added property gets included in the backend response dict of the **ReadOnlyExplorationBackendApiService.**
- As explained in Addition of a new to_metadata_dict function in Exploration domain object, the metadata properties of an exploration will be received by the newly created method **to_metadata_dict** in the Exploration domain object.
- To make sure that the new property gets included, we need to add this property in the return value of the function. Suppose that the new property is 'prop', then the **to_metadata_dict** function will look like:

```
def to_metadata_dict(self):
    return {
        ------- Other properties --------
        'prop': self.prop
    }
```

- Along with that, we also need to include this property in the New ExplorationMetadata domain object.
- After doing this, the new property will be included in the response dict of the **ReadOnlyExplorationBackendApiService**. As explained in Addition of two new properties in the Compare versions service's return value, we will add two new properties **v1Metadata** and **v2Metadata** which will store metadata information about the two selected versions. Since we have added the new property 'prop' to the return value, **v1Metadata and v2Metadata** will also include this property.
- Now, the new property 'prop' will show up while visualizing the difference between the old (**v1Metadata**) and new (**v2Metadata**).

Subproject (b)

Structure of the version history of a state

Suppose we have an exploration with two states having names "First State" and "Second State" and their version history list looks like the following:

State name	**Version history**
First State	`[{` `version: 1,` `committer: 'abc123'`

	`}, {`
	` version: 3,`
	` committer: 'abc123'`
	`}]`

Second State	`[{`
	` version: 5,`
	` committer: 'abc123'`
	`}]`

Here, version: 1 means that the state has been updated while upgrading the exploration from version 1 to version 2.

ExplorationStateVersionHistoryBackendDict

```
export interface ExplorationStateVersionHistoryBackendDict {
 'version': number;
 'committer_username': string;
}
```

ExplorationStateVersionHistory

```
export interface ExplorationStateVersionHistory {
 version: number;
 committerUsername: string;
}
```

- Steps needed to achieve the solution:
    - When the exploration editor page loads, fetch the version history list of all the states of that particular exploration in the above format. We now have access to the version numbers on which a particular state was modified.
    - In each state card, show the link “Last edited by XXX at version YYY” to the user and when they click on it, show the difference between versions YYY and YYY + 1 for that particular state in the form of a modal. In the modal, there will be a link reading “Previously edited by AAA at version BBB” and clicking on this will show the difference between versions BBB and BBB + 1. This process will continue until the user reaches the point where the state was created i.e. when there is no more “previously edited version” for that state.

Fetching version history list of all states

Method 1: Fetch the versions history list of all states at once (in the above mentioned format) and use CompareVersionsService each time the user clicks on “Last Edited by ….” link to fetch the diff data

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': []
 }
```

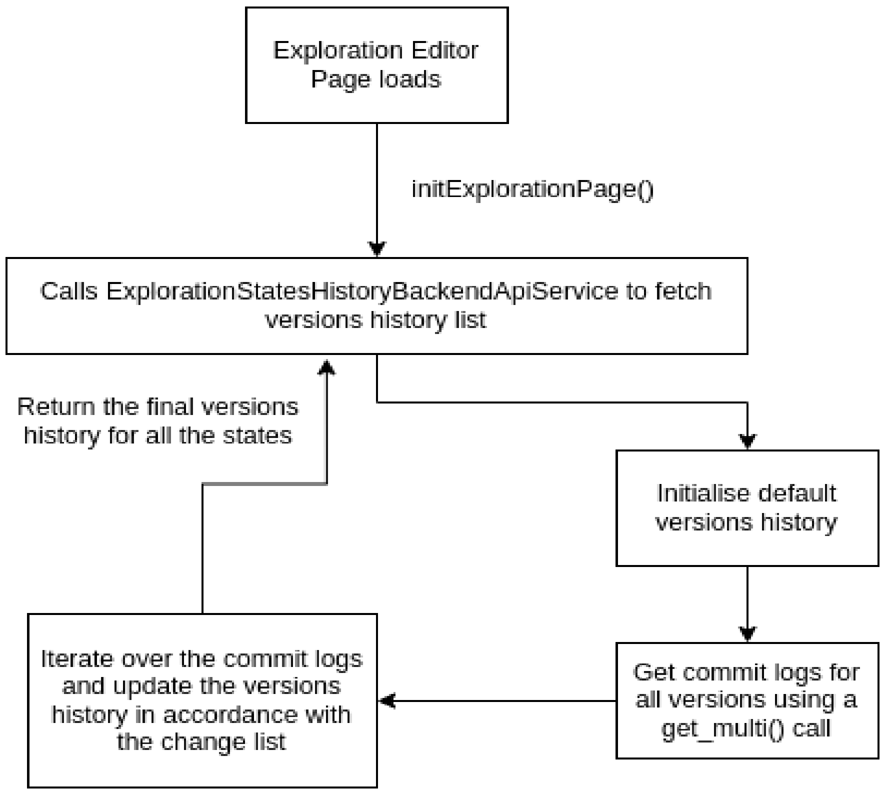
- It is initialized like this because when a default exploration is created, it has only one state and its name is ‘Introduction’.
    - Fetch the commit logs for all the versions of the exploration so that the versions history can be updated accordingly. As discussed above, it is fetched by **ExplorationCommitLogEntryModel**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

The changes can be tabulated as follows:

File name	**Function name**	**List of changes**
main.py		Create the endpoint for fetching versions history list for exploration states.
reader.py	ExplorationStatesVersionHistoryHandler	Create a new handler to handle the request for fetching version history list.
exploration-states-history-backend-api.service.ts		Create the new backend api service.
exploration-editor-page.component.ts	initExplorationPage	Use the newly created backend api service for fetching the version history data.


Exploration Editor Page loads
initExplorationPage()
Calls ExplorationStatesHistoryBackendApiService to fetch versions history list
Return the final versions history for all the states
Initialise default versions history
Iterate over the commit logs and update the versions history in accordance with the change list
Get commit logs for all versions using a get_multi() call

- 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.

Why are we fetching version history for all the states at once?

- The answer to this is efficiency.
- Suppose that instead of fetching the full version history of all the states at once, we are just fetching the “previously edited version” for a particular state every time. For this, we would have to check the change list of every version until we find a version where the state has been modified.
- This would mean that there might be several datastore calls for commit logs of different versions and it would make the process very slow (especially when the state has not been modified for a long time).
- Even if we optimize the process and fetch commit logs for all the versions at once (using **get_multi** call) and find the “previously edited version”, the process will still not be very efficient because the “previously edited version” will be required each time the state editor loads and each time the user clicks on the link “Last edited by ….”.
- Also, on changing the active state, the process will repeat for other states also. Also, users might want to navigate through the versions of a state faster.
- Hence, when we fetch the version history list of all the states at once, we don’t have to query the datastore every time when we need the “previously edited version” for a particular state because we will have this data precomputed for all states. Also, since the

data will be fetched for all the states, changing the active state will also not query the datastore again. This will make the overall user experience reasonable and faster at the same time.

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: Precomputing the version history so that it can be fetched faster during runtime

In this method, since we will be precomputing the version history data, we will be fetching version history data only for the selected or active state.

Current system

- The state editor is loaded by the function **initStateEditor** of the exploration editor tab component.
- This function is called when the exploration editor page loads, when the active state is changed and when some changes are saved in the exploration. We can fetch the version history data here.

Required changes

- For this method, we will be storing the version history of exploration states in a new model called **ExplorationStatesVersionHistoryModel.** A new instance of this model will be created each time a new exploration is created and updated according to the change list each time the exploration is updated.
- However, for this method to work for all explorations, we will require to populate the model for already existing explorations. For this, a new beam job will be created called **ExpStatesVersionHistoryComputationJob.**

- When the **initStateEditor** function is called, we will fetch the version history for the selected or active state and store the response in a service for further use.

```
ExplorationStatesVersionHistoryBackendApiService.fetchVersionHistory(
  ContextService.getExplorationId(), ctrl.stateName
).then((response) => {
  _________ Store the response in a service__________
});
```

Following is the summary of this method:

- **Creation of a new model**
    - A new model called **ExplorationStatesVersionHistoryModel** for storing the version history of all states of a particular exploration.
    - This model will have a single field called **version_history** which will store the version history for all the states of the exploration in the format explained in Structure of the version history of a state.

- **Creation and modification of instances of the model**
    - For new explorations, a new instance of this model will be created each time a new exploration is created.

```
    exp_states_version_history_model =
exp_models.ExpStatesVersionHistoryModel(
        id=exploration.id,
        version_history={
            state_name: [] for state_name in exploration.states
        }
    )
```

    - Also, this model will be updated every time some changes are made into any state of that exploration. Hence, for new explorations, this method will work perfectly fine.

```
        exp_states_version_history_model =
exp_models.ExpStatesVersionHistoryModel.get(
            exploration.id)
        if exp_states_version_history_model is not None:
            exp_states_version_history =
exp_domain.ExplorationStatesVersionHistory.from_dict(
                exp_states_version_history_model.version_history)
            exp_states_version_history.update_history_from_change_list(
                change_list, exploration.version, username)
            exp_states_version_history_model.version_history =
exp_states_version_history.to_dict()
            exp_states_version_history_model.update_timestamps()
            exp_states_version_history_model.put()

```

    - For older explorations, we will need to populate the model for each exploration with the help of a beam job in order for the method to work properly.

- **Creation of the new one-off beam job to populate the model for older explorations**

- The working of the beam job will be similar to how the version history for all the states is generated in Method 1.
- The beam job will work as follows:
    - Fetch all the explorations.
    - For each exploration, create a default version history, get the commit logs for all the versions of that exploration (using the get_multi call) and iterate over the commit logs.
        - In each iteration, get the change list for that particular version and update the version history according to the change_list.
    - After getting the states version history for that exploration, create an instance of the **ExplorationStatesVersionHistoryModel** and save it on the datastore.
- If the beam job fails midway due to some reason, no harm will be caused to the existing data because we are storing this new data on a new model.

Now, similar to method 1, we will use CompareVersionsService each time the user clicks on the link to fetch the diff data.

The changes can be tabulated as follows:

File name	**Function name**	**List of changes**
gae_models.py (exploration)	ExplorationStatesVersionHistoryModel	Create the new model
exp_services.py	_save_exploration	Implement the updation of version_history of the exploration when some changes are saved onto it.
exp_services.py	_create_exploration	Create an instance of ExplorationStatesVersionHistoryModel for the newly created exploration.
exploration-editor-tab.component.ts	initStateEditor	Use a backend api service to fetch the version history data for the selected (or active) state.
exp_states_version_history_computation_jobs.py	ExpStatesVersionHistoryComputationJob	Create the new beam job.

Why are we using a new model to store states' version history rather than storing it in the states of the exploration model?

- This is because storing the data in a new model is much safer than modifying the existing models.
- If we are modifying already existing models and the beam job fails midway, then reverting back to the original configuration would be difficult.
- However, if we are storing the data in a new model, we can just delete the data present in the new model to go back to the original configuration.

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 "Last edited by …" link will be hidden from the user for that particular exploration. The rest of the page will not be affected.

Performance considerations

- Tested on oppia development server.
- Operating system: Ubuntu 20.04
- Browser: Chrome Version 96.0.4664.110 (Official Build) (64-bit)
- In the tests, the timestamps were noted for the beginning and end of the fetching processes.

Method 1

- In this case, the timestamps were recorded in the exploration states history backend api service (this service is explained briefly in next sections).
- The starting timestamp was recorded before making the request.
- The ending timestamp was recorded when the request was complete.

Exploration with 100 commits

Attempt	Result (time taken in seconds to fetch 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.

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.

Method 2

- **On the first load of the exploration editor page**:

Attempt	Result (time taken in seconds to fetch the version history)
1	0.612
2	0.551
3	0.6

- In this case, the average time taken comes out to be 0.587 seconds.

- **On subsequent changing of the active state, the data is fetched again:**

Attempt	Result (time taken in seconds to fetch the version history)
1	0.124
2	0.092
3	0.085

- In this case, the average time taken comes out to be 0.1 seconds.

- **On saving some changes on the exploration, the data is fetched again:**

Attempt	Result (time taken in seconds to fetch the version history)
1	0.22
2	0.178
3	0.149

- In this case, the average time taken comes out to be 0.182 seconds.

Creation of a new backend api service for fetching version history data

- The name of the service will be **ExplorationStatesVersionHistoryBackendApiService**. It will contain a method named **fetchVersionHistory** which will take the exploration id and state name as arguments and send a request to the URL /explorehandler/<exploration_id>/version_history/<state_name> to fetch the version history data.

- The structure of its response dict will be:

```
interface VersionHistoryResponse {
 'version_history': ExplorationStateVersionHistory[];
}
```

- The structure of the function will look like the following:

```
 fetchStateHistory(
     explorationId: string,
     stateName: string ) {
   const start = Date.now();
   return this.http.get(
     this.urlInterpolationService.interpolateUrl(
       '/explorehandler/<exploration_id>/version_history/<state_name>', {
         exploration_id: explorationId,
         state_name: stateName
       })
   ).toPromise()
```

Storage of the version history in a service for further use

For this, a new service will be created named **ExplorationStatesVersionHistoryService**. As the **initStateEditor** receives the version history for the active state, this service will be initialized.

It will contain the following properties:

- **_versionHistory**:
    - The version history of the active state in the state editor.
    - Its structure will be ExplorationStateVersionHistory [ ].
    - It will get reset each time the active state changes or new changes are saved on the exploration.

- **_selectedStateName**:
    - The name of the selected state (or active state).
    - It will get reset each time the active state changes or new changes are saved on the exploration.

- **_index**:
    - This is the index of the version history list of the selected or active state.
    - It gets initialized pointing to the last element of the versions history list of the active state each time the active state changes or new changes are saved on the exploration.
    - This gets decremented each time the user clicks the link "Last/Previously edited by XXX at version YYY" to point to the previously edited version.
    - When the value of the index becomes -1, then there will be no more "previously edited version" of the state and we can stop showing the link to the user.

- **_oldStateName:**
    - It might be possible that the name of the active state has changed in earlier versions of the exploration.
    - Its value will be the name of the active state in the version of the version history list (corresponding to the index: **_index + 1)** in order to handle the cases where the state was renamed. If **_index** is the last index, its value will be the name of the active state in the latest version of the exploration.
    - It is required because the version diff data is indexed by state names and the name of the active state might have some other value at that version (i.e. version corresponding to the index: **_index + 1**).

- **_userInfo:**
    - Info of the logged in user.
    - It is required because we need the committer username whenever some changes are saved in a state and we need to update the **versionHistory**.

The changes can be tabulated as follows:

File name	**Function name**	**List of changes**
exploration-editor-page.comp	initExplorationPage	At the then block of the

onent.ts		function, initialize the newly created service with the received version history from the backend.
exploration-states-version-history.service.ts		Create the new service as mentioned above.

Initialization of the ExplorationStatesVersionHistoryService

This service will be initialized every time the function **initStateEditor** of exploration editor tab component runs. Following changes will be made while the service is initialized:

- Set the **_versionHistory** to the received version history after running ExplorationStatesVersionHistoryBackendApiService.
- Set the **_selectedStateName** to the active state name.
- Set the **_index** to point to the last element of the **_versionHistory.**
- Set the **_oldStateName** to the active state name.
- Fetch the user info and set the value of **_userInfo.**

Decrementing the _index to point to the "previously edited version"

- Each time the user clicks on the link "Last/Previously edited by XXX at version YYY", the index will be decremented to point to the "previously edited version" until its value becomes -1. When it becomes -1, it means that there is no more "previously edited version" for that state and the process ends.

Updating the value of _oldStateName

- Each time the user clicks on the link, the diff data between versions **_versionHistory[_index].version** and **_versionHistory[_index].version + 1** is fetched.
- When the response is received, we can change the value of **_oldStateName** to node.originalStateName as follows:

```
    const stateName = (
      this.explorationStatesVersionHistoryService.getOldStateName());
    let node = null;
    for (let i = 1; i <= Object.keys(response.nodes).length; i++) {
        if (response.nodes[i].newestStateName === stateName) {
          node = response.nodes[i];
          break;
        }
    }
    const newestStateName = node.newestStateName;
    const originalStateName = node.originalStateName;
    const stateProperty = node.stateProperty;
```

```
this.explorationStatesVersionHistoryService.setOldStateName(originalStateName);
```

Creation of a new modal to show the diff

- A new modal will be created to show the difference between versions YYY and YYY + 1 which will contain another link at the bottom right corner reading “Previously edited by ……”.
- It will also use CompareVersionsService to fetch the diff data when the user clicks on the link at the bottom right corner.

Explanation of the solution with an example

- Suppose the exploration has only one state and the version history looks like:

State name	**Versions history**
Introduction	`[{` `version: 1,` `committer: 'abc123'` `}, {` `version: 2,` `committer: 'abc123'` `}, {` `version: 5,` `committer: 'abc123'` `}];`

User opens the exploration editor page

- The page gets loaded and **initStateEditor** fetches the version history data for the state ‘Introduction’ from the backend and stores it in the exploration states version history service. Also, the selected state name and the index are set to ‘Introduction’ and 2 respectively.

Property	**Value**
selectedStateName	Introduction
index	2

User clicks on the link “Last Edited by abc123 at version 5”

- On clicking the link, **CompareVersionsService,** will fetch the comparison data of the exploration between versions 5 and 6.

- After the data is received, a modal will prompt showing the changes made to the selected state from version 5 -> 6.
- The index will be decremented by one to get the “previously edited version”.

Property	**Value**
index	1
selectedStateName	Introduction

User clicks on the link “Previously edited by abc123 at version 2” on the bottom right corner of the modal

- The same process of fetching the comparison data will occur between versions 2 -> 3 and the modal data will be updated. Index will be decremented again.

Hence, at this stage:

Property	**Value**
index	0
selectedStateName	Introduction

- Now, if the user clicks on the link “Previously edited by abc123 at version 1”, the data will be updated again and index will become -1.
- However, since index has now become -1, there will be no more “previously edited version” for the selected state. Hence, the link will no longer be visible to the user.

What If the user dismisses the modal at any point

- In this case, the **index** will be reset to point to the last element of the version history list of the selected state. In our case, it will be 2 again.

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	yaml (1.10.2)	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	ISC License

[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.

			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.	

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.

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 fetching the version history in Subproject(b), the following alternatives were considered for fetching the version history of all states of the exploration:

1. Method 1
2. Method 2

The performance considerations are discussed in detail in the section Performance considerations.

Among these, I believe that Method 2 is a better approach because of the following comparison:

	Method 1	**Method 2**
Performance	Two (1 get + 1 get_multi) datastore calls along with O(N^2) operations for generating the version history from scratch. The get call is to fetch the exploration by id and get_multi call is to fetch the commit logs for all the versions of the exploration.	One (1 get) datastore call to fetch the version history for the active state. Since the data is precomputed, there is no need for further operations. The get call is to fetch the precomputed version history.
Size of data	Here, we are fetching the full version history. Hence, the data could be very large in case of explorations with many states (more than 30) and 100s of commits. Also, most of the data might not even be used by the user. This means that a huge amount of data might be unnecessary in this approach. Lastly, this approach might take a long time to finish for users with slow internet connection if the size of data transferred is huge.	Here, we are just fetching the version history of the active state. Hence, the size of data sent to the frontend is less in this approach. This also reduces the amount of unnecessary data fetched from the backend. This approach would be a bit lighter in the context of slow networks because it only fetches the version history of the active state.

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

A new model called ExplorationStatesVersionHistoryModel will be created. The specs of this model will be as follows:

ExplorationStatesVersionHistoryModel

- It will store the version history of all the states of a particular exploration.
- The key of this model will be the exploration_id.
- **Properties:**
    - version_history: datastore_services.JsonProperty
        - It will store the version history in the format explained in Structure of the version history of a state.
- **Life cycle**
    - **[CREATION]** A new instance of this model will be created each time a new exploration is created
    - **[UPDATION]** The version_history will be updated according to the change_list when new changes are saved onto the exploration.
    - **[DELETION]** The data of the model will be deleted when the exploration gets deleted.
- **Supported query functions**
    - GET BY ID: Get the version history of all the states of an exploration by its id.

Domain Objects

Backend

VersionHistory Domain Object

- **Properties**
    - **version**: int. The version number of the exploration.
    - **committer_username**: str. The username of the user who committed the changes at that version.
- **Methods**
    - **from_dict**
    - **to_dict**

ExplorationStatesVersionHistory Domain Object

- **Properties**
    - **version_history**: dict
        - [KEY] state_name: str.

        - [VALUE] version history of that state: list[VersionHistory].
- **Methods**
    - **__init__(self)**
        - Initializes the default version history object. The state 'Introduction' is written because it is the first state of a default exploration.
```
        self.version_history = {
            'Introduction': []
        }
```
    - **require_state_name_to_exist(self, state_name)**
        - **Args**:
            - state_name: str. The name of the state to check.
        - **Raises**:
            - ValidationError if the key '**state_name**' does not exist in the **self.version_history** dict.
    - **require_version_to_be_unique(self, state_name, version)**
        - **Args**:
            - state_name: str. The name of the state.
            - version: int. The version number to check.
        - **Returns**:
            - bool. Returns true if the **version** does not exist in the version history of the state "**state_name**". Otherwise, it returns false.
    - **update_from_change_list(self, change_list, version, committer_username)**
        - **Args**:
            - change_list: list[ExplorationChange]. The change list for that **version**.
            - version: int. The version number.
            - committer_username: str. The username of the user who committed the changes at that version.
        - **Pseudo Algorithm**:
            - Iterate over the change list.
            - If change.cmd == 'add_state'
                - Add the state in **self.version_history** with its value being [VersionHistory(version, committer_username)].
            - If change.cmd == 'delete_state'
                - Remove the key **state_name** from **self.version_history**.
            - If change.cmd == 'rename_state'
                - Copy the version history of the old state and move it to the new state by creating a new key for the new state in **self.version_history**.
                - Delete the contents of the old state from **self.version_history**.

                - Append the **version** in the version history of the new state if it does not exist already.
            - If change.cmd == ‘edit_state_property’
                - Append the **version** in the version history of the state if it does not exist already.
    - **to_dict**
    - **from_dict**

Frontend

Renaming the already existing ExplorationMetadata domain object

- Since it does not contain all the exploration metadata properties, it will be renamed to **ShortExplorationMetadata.**

New ExplorationMetadata domain object

- After renaming the already existing ExplorationMetadata, a new ExplorationMetadata domain object will be created which will contain all the properties mentioned in Subproject (a).
- It will contain general to_dict and from_dict methods and will be useful in handling metadata information after implementing the solution for subproject(a).

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 “Metadata node”**
    - **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, a request is sent to fetch the version history of the active state.**
    - **Pseudo algorithm**
        - **initStateEditor** sends a request to fetch the version history for the active state.
        - In the backend, the ExplorationStatesVersionHistoryModel is fetched for the given exploration id.
        - The version history for the active state is sent to the frontend (explorationStatesVersionHistoryModel.version_history[state_name]).
        - The ExplorationStatesVersionHistoryService is initialized with the fetched data.
    - **URL Endpoint**
        - /explorehandler/<exploration_id>/version_history/<state_name>
    - **Handler**
        - ExplorationStatesVersionHistoryHandler (newly created).
    - **Additional Datastore calls**
        - [**SYNC**] **get** (by id): 1

- **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 change list as follows:
            - Fetch the ExplorationStatesVersionHistoryModel.
            - Update the model according to the change list.
            - Put the model into the datastore.
        - After that, the **initStateEditor** function runs again and fetches the latest version history for the active state.
        - The ExplorationStatesVersionHistoryService is initialized again with the new data.
    - **URL Endpoint**

        - /createhandler/data/<exploration_id>?apply_draft=<apply_draft>
    - **Handler**
        - ExplorationHandler in editor.py.
    - **Additional Datastore calls**
        - [**SYNC**] **get** (by id): 1
        - **[SYNC] put**: 1

- **User changes the active state.**
    - **Pseudo algorithm**
        - The function **initStateEditor** runs again and fetches the version history data for the new active state.
        - The ExplorationStatesVersionHistoryService is initialized again with the new data.
    - **URL Endpoint**
        - /explorehandler/<exploration_id>/version_history/<state_name>
    - **Handler**
        - ExplorationStatesVersionHistoryHandler (newly created).
    - **Additional Datastore calls**
        - [**SYNC**] **get** (by id): 1

- **User clicks on the link “Last/Previously edited by XXX at version YYY”**
    - **Pseudo algorithm**
        - CompareVersionsService will use ReadOnlyExplorationBackendApiService to fetch the data of exploration versions YYY and YYY + 1.
        - Fetch the ExplorationModel, ExplorationRightsModel and UserSettingsModel.
        - Process the fetched data and return the read only exploration data.
        - After fetching the data from the frontend, a new modal will popup which will show the difference between the versions YYY and YYY + 1.
        - The modal will contain another link at the bottom right corner and on clicking this, the same process will repeat again until the user reaches the version when the state was created.
    - **URL Endpoint**
        - /explorehandler/init/<exploration_id>?v=<version>
    - **Handler**
        - ExplorationHandler in reader.py
    - **Additional Datastore calls**
        - None

Modifications in backend handlers

ExplorationHandler (in reader.py)

- The response dict of this handler will be slightly modified to include a new property called **exploration_metadata**. The url endpoint for this handler is /explorehandler/init/<exploration_id>?v=<version>.

ExplorationStatesVersionHistoryHandler (in reader.py)

- This handler will be newly created to handle the request for fetching the version history of the active state.
- **URL Parameters**
    - exploration_id: str. The id of the exploration.
    - state_name: str. The name of the active state.
- **Pseudo algorithm**
    - Fetch the ExplorationStatesVersionHistoryModel for the given exploration id.
    - Retrieve the version history of the active state.
    - Send it as the response.

Beam Jobs

ExpStatesVersionHistoryComputationJob

- This will be a one-off job to populate the ExplorationStatesVersionHistoryModel for already existing explorations.
- **Pseudo algorithm:**
    - For each ExplorationModel:
        - Generate default ExplorationStatesVersionHistory object.
        - Get the commit logs for all the versions of the exploration.
        - Iterate over the commit logs and in each iteration:
            - Get the change list.
            - Apply ExplorationStatesVersionHistory.update_from_change_list by supplying the change list to update the version_history dict.
        - Now, we have the version history of all the states of that exploration.
        - Create ExpStatesVersionHistoryModel.
        - Return the ExpStatesVersionHistoryModel.
    - For each ExpStatesVersionHistoryModel:
        - Apply ndb_io.PutModels().

Web frontend changes

Create a new metadata diff modal component

- This component will be used for visualizing metadata diff similar to state diff modal.

- **Properties:**
  - oldMetadata: Metadata of first selected version.
  - newMetadata: Metadata of second selected version.
  - yamlStrs: yaml strings of the left and the right sides.
- **Methods:**
  - ngOnInit:
    - Set the right and the left side metadata yaml strings.

```
this.yamlStrs.leftPane = this.yamlService.stringify(this.oldMetaData);
this.yamlStrs.rightPane = this.yamlService.stringify(this.newMetaData);
```

Create a yaml service to use the newly installed ‘yaml’ library

- **Methods:**
  - stringify(object: Object):
    - Convert the given object into yaml string.
  - parse(yamlStr: string):
    - Parse the yaml string and return the corresponding object.

Create ExplorationVersionCacheService

- This service will store version specific exploration data to make the history tab faster.
- Structure of the exploration version cache object:

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

- **Properties:**
  - static _cache: ExplorationVersionCache = {}
- **Methods:**
  - get(explorationId: string, version: number):
    - Get the cached exploration data of the given version if it exists.
  - set(explorationId: string, version: number, explorationData: FetchExplorationBackendResponse):
    - Set the explorationData for the given exploration id and given version.
  - reset()
    - Reset the value of _cache to {} (empty object).

Create ExplorationStatesVersionHistoryService

- This service will store the version history of the active state of an exploration. It is explained in Storage of the version history in a service for further use.
- **Properties:**

    - _versionHistory: ExplorationStatesVersionHistory
    - _userInfo: UserInfo
    - _selectedStateName: string
    - _index: number
    - _oldStateName: string
    - These properties are explained in detail in Storage of the version history in a service for further use.
- **Methods:**
    - getPreviouslyEditedVersion()
        - Returns the version from _versionHistory corresponding to the current _index.
    - getPreviouslyEditedCommitterUsername()
        - Returns the committerUsername from _versionHistory corresponding to the current _index.
    - decrementIndex():
        - Decrements the index by one.
    - resetIndex():
        - Resets the index to point to the last element of the _versionsHistory array.
    - getOldStateName():
        - Returns the value of _oldStateName.
    - setOldStateName(stateName: string):
        - Sets the value of _oldStateName to the given name.
    - setSelectedStateName(stateName: string):
        - Sets the value of _selectedStateName to the given name.

Create a new modal component for navigating through the version history of the states

- It will be used to navigate over the version history of a state.
- It will contain a link at the bottom right corner reading "Previously edited by ……" which on clicking, would update the modal data to the "previously edited version" of the state. The modal is explained through mocks and videos in the section Key User Stories and Tasks.
- **Properties:**
    - The properties will be similar to the state diff modal.

```
newState: State | null;
oldState: State | null;
newStateName: string;
oldStateName: string;
headers: headersAndYamlStrs;
yamlStrs: headersAndYamlStrs = {
  leftPane: '',
  rightPane: '',
};
```

- **Methods:**
  - **fetchDiffData**():
    - It will fetch the diff data between versions YYY and YYY + 1 using the CompareVersionsService when the user clicks on the link “Previously edited by XXX at version YYY”.
  - **getPreviouslyEditedVersion**()
    - Get the previously edited version from ExplorationStatesVersionHistoryService.getPreviouslyEditedVersion().
  - **getPreviouslyEditedCommitterUsername**()
    - Get the previously edited committer username from ExplorationStatesVersionHistoryService
      .getPreviouslyEditedCommitterUsername().

Changes in exploration editor tab component

- **initStateEditor**
  - Fetch the version history for the active state each time the function runs and initialize the ExplorationStatesVersionHistoryService.
- **fetchDiffData**
  - Create this function to fetch the diff data between versions YYY and YYY + 1 using the CompareVersionsService when the user clicks on the link “Last edited by XXX at version YYY”.
- **getPreviouslyEditedVersion**
  - Get the previously edited version from ExplorationStatesVersionHistoryService.getPreviouslyEditedVersion().
- **getPreviouslyEditedCommitterUsername**
  - Get the previously edited committer username from ExplorationStatesVersionHistoryService
    .getPreviouslyEditedCommitterUsername().
- Add a new link on the bottom right corner of the page reading “Last edited by XXX at version YYY”.

Launching plan for Subproject (B)

Run the beam job **ExpStatesVersionHistoryComputationJob** in maintainence mode to populate the model for already existing explorations.

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 - July 24)

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	• Create the VersionHistory and ExplorationStatesVersionHistory domain objects in the backend. • Also, create the new ExplorationMetadata domain object in the frontend and rename the existing one to ShortExplorationMetadata.	None	13th June	17th June
2	• Make the changes for getting metadata information from the backend (Getting metadata information from the backend) • Create the frontend cache service for caching exploration versions (Caching of the fetched exploration versions make the comparison faster) • Modifications in frontend and backend tests.	1	17th June	21st June
3	• Install the new 'yaml' library • Create the YamlService. • Use the new YamlService in the StateDiffModal to eliminate usage of the backend request for conversion of state_dict to yaml. • Modifications in frontend and backend tests.	None	22nd June	26th June
4	• Create a new UI component to show the comparison in exploration metadata between the two selected versions. • Write frontend tests for it.	2, 3	27th June	30th June
5	• Make the changes required in exploration diff service (Addition of an extra node called "Metadata" on the diff graph data and recording the changes to exploration metadata) • Modifications in the frontend tests.	2	1st July	5th July
6	• Make the changes in CompareVersionsService to add the two new properties in its return value (Addition	4, 5	6th July	10th July

	of two new properties in the Compare versions service's return value) • Make the changes in version diff visualization component (Changes in version diff visualization component). • Modifications in the frontend tests.			
7	• Write the e2e tests for Subproject(a)	6	13th July	18th July
8	• Create the ExplorationStatesVersionHistoryModel and implement its life cycle (creation, deletion and updation). • Create a function in exp_fetchers.py to fetch the version history for an exploration by its id. • Modifications in the backend tests.	None	18th July	22nd July
9	• Fixing bugs (if any).	__	22nd July	24th July

Milestone 2(July 25 - September 4)

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
10	• Create the ExpStatesVersionHistoryComputationJob along with its backend tests. • Run it on the server to populate the model for already existing explorations.	8	25th July	29th July (Merged) 2nd August (Run on the production server)
11	• Create the new URL endpoint and the new handler for fetching the version history for the active state. • Create the new backend api service for fetching the version history data. • Create the new service to store the version history data. • Write tests for the above services and handlers.	8, 10	2nd August	8th August
12	• Create the new modal component to navigate through the version history of the active state and implement all the required functionalities explained in Create a new modal component for navigating through the version history of the states. • Write frontend tests for the modal component.	11	8th August	13th August

13	• Make the required changes in the exploration editor tab as explained in Changes in exploration editor tab component component. • Modifications in the frontend tests.	12	13th August	19th August
14	• Write the e2e tests for Subproject(b)	13	23rd August	31st August
15	• Fixing bugs (if any)	—	1st September	4th September

Future Work

- I will try to extend the feature as mentioned in the project idea so that users can move forward and back over the version history of a state.
- Moreover, I will continue to be a part of oppia in the future.