Section 1: About You

Goals and Objectives

<table>
<tr><td>Why do you want to do a GSoC project with Oppia?</td><td colspan="2">Hi! I'm Vir Kothari, someone who has been extremely passionate about coding ever since my childhood. I began my journey independently around 8th grade with Java, proceeding to dabble heavily in game development with Unity and later learning about web development.

I'm very enthusiastic about web development and I've been contributing to Oppia for the past couple of months. It has taught me a lot so far, with the marvelous goal of providing free education to kids being highly inspirational.

Currently being a freshman who has a lot more to learn, doing a GSoC project with Oppia means a lot to me, it would be a tremendously exciting opportunity and I cannot stress enough how much it would help me learn more, developing my skills further. I genuinely couldn't ask for a better opportunity. Also being someone who has always deeply valued knowledge, I believe that Oppia's goals greatly align with my values - making it the perfect opportunity.</td></tr>
<tr><td>What do you hope to learn/achieve during GSoC?</td><td colspan="2">Deepen my understanding of underlying architectures of the website - especially the backend, learn to communicate more effectively and learn to plan projects better. Make the most out of this brilliant opportunity in every way possible, always striving to learn more in every domain.</td></tr>
<tr><td rowspan="3">Which Oppia teams have you collaborated with, and what have you done on those teams?</td><td>LaCE</td><td>- #19910: Help triage an issue with decimal inputs on iOS.
- #19527: Help a new contributor fix their first issue, making their first PR.</td></tr>
<tr><td>Developer Workflow</td><td>1. Investigate why the pre-push hook is slow.
2. Raise concerns on why the frontend tests are slow on M1 Macs.
3. Raise concerns on why Chrome disconnects while running tests in Docker.</td></tr>
<tr><td>Release Team</td><td>- Assist with testing Oppia Release 3.3.7</td></tr>
<tr><td>Contact information</td><td colspan="2">virkothari8@gmail.com</td></tr>
<tr><td>Preferred method of communication</td><td colspan="2">E-mail, Google Chat</td></tr>
<tr><td>Which timezone will you primarily be in during the summer?</td><td colspan="2">GMT +5:30, IST</td></tr>
<tr><td>(If you are a student)</td><td colspan="2">1 June, 2024 - 1 August, 2024.</td></tr>
</table>

When are your school holidays?	
What other obligations might you need to work around during the summer?	Exams from 21 May, 2024 - 31 May, 2024.
Planned time commitment	1 hour a day until 31 May as my exams end. 6 hours a day, 6 days/week from 1st June until 1st August. 4-5 hours a day, 5 days/week as my semester starts in August.

Section 2: About Your Project

Project Details

Project title	Improvements to the exploration editor page
Project size	Large (~350 hours)
Why did you choose this project?	I chose this project because the exploration editor page being one of the core integral parts of Oppia, I strongly believe that it is extremely important to maintain the quality of these functionalities. It is of utmost importance to ensure that features such as addressing misconceptions are made available so that an enhanced learning experience is provided for learners across Oppia.

Required Skills

WEB	
I can write Python code with unit tests.[1]	#19471: Blog Admin page migration to Angular #19524: Add Feature Flag for Lesson Player redesign
I can write TS + Angular code with unit tests.[2]	#19471: Blog Admin page migration to Angular #19441: Make and integrate primary-button component #19676: Rework Primary Button component
I have good UI judgment and attention to detail.[3]	#19676: Rework Primary Button component #19808: Create Sidebar component for New Lesson Player

[1] To develop this skill: *Some LaCE and Contributor Dashboard issues have a backend component, and many Dev Workflow issues do too. Focus on issues that aren't marked as "backlog".*
[2] To develop this skill: *Most LaCE and Contributor Dashboard issues have a frontend component. Focus on issues that aren't marked as "backlog".*
[3] To develop this skill: *Tackle a non-backlog responsiveness issue from the LaCE team.*

I can debug and fix CI failures / flakes.[4]	Fix e2e test flake introduced unexpectedly as a result of migration - #19508: Learner Dashboard & Curriculum Admin page migration to Angular
I can write or modify e2e or acceptance tests.[5]	#19914: Curriculum Admin - Topic Management acceptance tests
I can communicate effectively using debugging docs.[6]	*N/A*
I can write or modify Beam jobs.[7]	*N/A*
I have participated in QA testing.[8]	Participated in testing Release 3.3.7: Issue #19955: Progress Page UI of Learner Dashboard broken in RTL languages
I can make changes to production code and/or tests that involve platform parameters.[9]	#19524: Add Feature Flag for Lesson Player redesign #19635: Make lesson player redesign accessible at /lesson route

Project Timeframe

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

Coding period	• I will adhere to the above deadlines.

Communication Channels

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

---

[4] To develop this skill: *See issues labelled "CI breakage" on GitHub, or look at CI failures in develop (or cross-signs here) and figure out how to fix them.*
[5] To develop this skill: *Solve part of issue #17712 by covering the CUJs for one or more sets of users.*
[6] To develop this skill: *Tackle an issue that requires creating a debugging doc that you share with the broader team. Fixing CI failures counts.*
[7] To develop this skill: *See this doc for some examples of issues to work on.*
[8] To develop this skill: *Join the release testers mailing list at oppia-release-testers@googlegroups.com, and look out for calls to help with release testing.*
[9] To develop this skill: *Any CLaM issues that affect features gated behind feature flags (i.e. unreleased features) would qualify. If you're unsure of a particular issue would include a feature flag, ask in our general GSoC Q & A Discussion board. If you're unable to demonstrate this skill, we'd also accept an explanation for how feature flags work and how they're used in both production and testing code.*

I can commit to sending daily updates to my mentor by email, each day I work during the GSoC period.	• Yes
In addition to the above: how often, and through which channel(s), do you plan on communicating with your mentor?	I plan on communicating daily with my mentor using Google Chat, and having daily or weekly calls on Google Meet, whenever possible and appropriate.

Section 3: Proposal Details

Note: *This section is largely adapted from* *Oppia TDD Template (Jun 2023) -- PLEASE MAKE A COPY* . ***We <u>strongly recommend</u> reading that doc**, since it has detailed explanations of how to fill out the various sections.*

Problem Statement

Target Audience	Lesson Creators / Curriculum Admins
Core User Need	1. As a lesson creator, I need to be able to update translations for trivial changes more efficiently, being able to view and edit them at once without the need for translators to repeat the process for each language. 2. As a lesson creator, I need to be able to see a list of misconceptions associated with the card's skills, in order to take further action. 3. As a lesson creator, I need to be able to tag answer groups with misconceptions in order to address them appropriately, drastically improving the learner experience.
What goals do we want the solution to achieve?	- Add ability to update translations for existing languages as well, when changes are made in any language in the state editor. - Display a list of non-optional misconceptions associated with a card in a curated exploration, highlighting the ones that haven't been addressed. - Add functionality to tag answer groups with misconceptions in order to address them, for curated explorations.

Section 3.1: WHAT

Key User Stories and Tasks that will be implemented

#	Title	User Story Description (role, goal, motivation) *"As a ..., I need ..., so that ...."*	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	Trivial Translations Management System	As a **lesson creator**, I **need to** update trivial translations more effectively, **so that** the process is streamlined and translations don't have to be updated for each language one-by-one.	Open exploration editor page for an exploration with translations.	Translation Modal Mocks
			Edit a state card or any user-facing exploration content.	
			See a modal asking the user to confirm action to be taken on current translations.	
			Click 'Modify existing translations', indicating current translations need to be marked as needing an update or can be updated on the spot.	
			See a modal with the available languages, along with the current translations for these languages.	
			Click on a translation to edit it appropriately, and press 'Save'.	
			Check the languages for which translations have been made.	
			Publish the exploration.	
2	Misconception Implementation in Exploration Editor	As a **lesson creator**, I **need to** be able to address common misconceptions effectively **so that** learners get appropriate feedback and a better learning experience.	Open a curated exploration's editor page.	Misconception Mocks
			Open the state editor for a card demanding response from the learner.	
			Add a relevant skill associated with misconceptions.	
			See a list of unaddressed misconceptions below response groups.	
			Clearly see mandatory misconceptions being distinguished from non-mandatory	

			misconceptions.	
			Have the option to mark a non-mandatory misconception as inapplicable.	
			Click on a response group and see a button to tag the response group with a misconception.	
			Click on the button to tag the response with a misconception.	
			See a pop-up modal for misconception selection and select a misconception, press 'Done'.	
			See response group being tagged with misconception, and not in the unaddressed misconceptions list.	
			Publish the exploration.	

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.	/entity_translations_bulk_handler	GET	New	For a particular entity, fetch all available translations from the backend. This new endpoint is necessary because we already have the functionality to fetch available translations but no usable endpoint. The only endpoint using said functionality (/explorehandler) faces 2 problems - 1. It filters and fetches only the displayable languages making it not possible to update translations in progress.

				2. It fetches a huge amount of other data alongside that is unnecessary for the feature.

UI Screens/Components

#	ID	Description of new UI component	i18n required?	Mock/spec links	A11y requirements
1.	Translation Changes Modal	Ask the lesson creator if the translations are trivial. On confirmation, displays a list of existing languages and the current translations of the content.	No	Translation Modal Mocks	Ensure that the modal is completely functional with just a keyboard. Make sure that the checkboxes follow web accessibility guidelines, and proper aria tags are maintained.
2.	Misconception Editor Component	This component is currently responsible for the 'Tag with misconception' button that leads to the misconception tagging modal, as well as changing already tagged misconceptions in the question editor. Currently, it's designed to be visible only in the question editor by checking if the editor is in question mode, in the answer-group-editor and the add-answer-group modal. The key task to make it generalized is to initialize the stateEditorService with misconceptions as mentioned here, since this service is what's used to make this component functional. The input parameters for this component's API are unchanged. They are:	No	Misconception Tag Button Mock & Misconception Tagging Modal Mock	Make sure that the elements are fully accessible with appropriate aria labels.

		1. outcome - The feedback of the response group. 2. isEditable - Boolean defining if the answer group is editable. 3. rules - The rules of the response group. 4. taggedSkillMisconceptionId - The ID of the tagged skill misconception.			

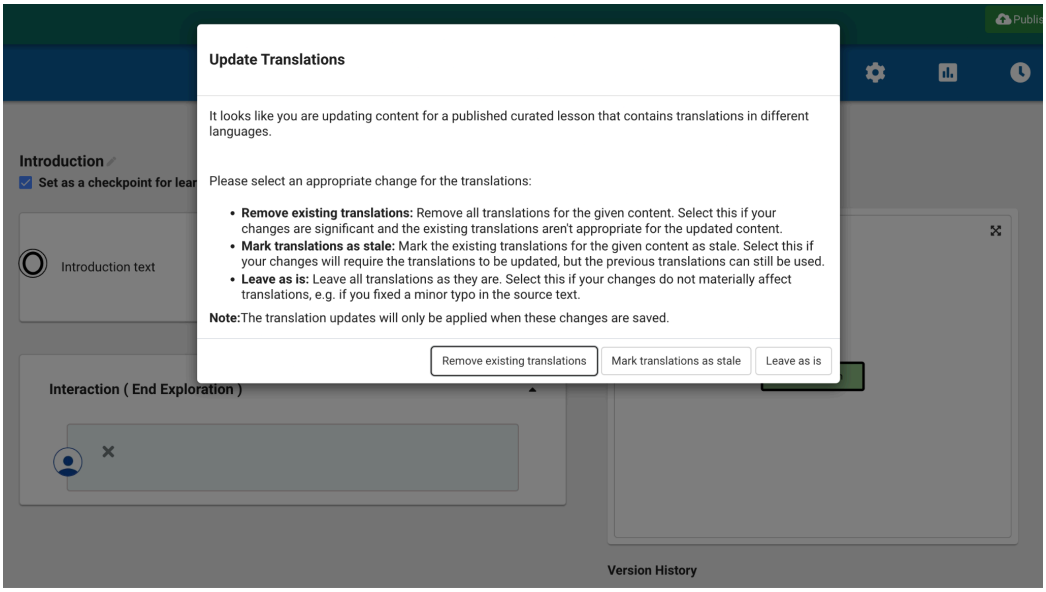
Other Requirements

N/A

Section 3.2: HOW

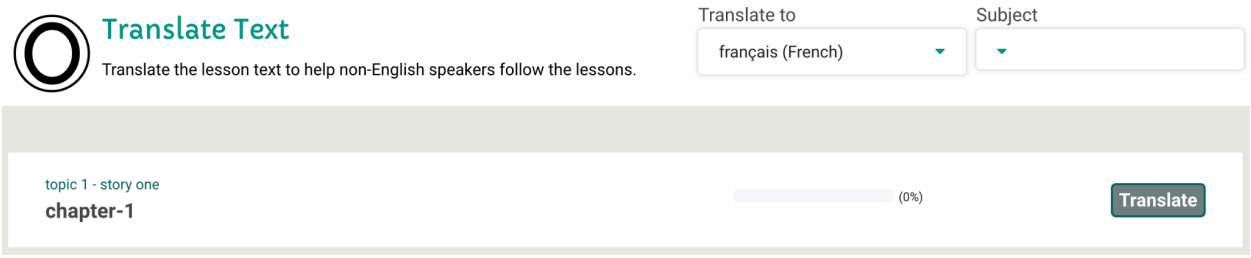
Existing Status Quo

1. Translations Management -
   Currently when a change is made to a part of a card, this results in a pop-up modal asking if translations should be updated.



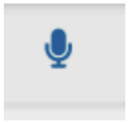
Marking the translations as ‘stale’ or needing an update will result in those translations being invalidated, and the whole content will need to be retranslated using the contributor dashboard. Trivial changes such as updating images are common for all translations, which the creator can apply themselves.

Although trivial, these changes will have to be made for every single language, one by one, either through the contributor dashboard -



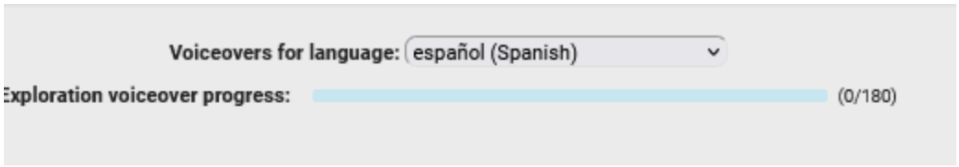
Or through the translations tab, as described in the steps 22-26 of this CUJ, performing repetitive steps unnecessarily for every single relevant language.

22. To upload images for other language translations, click on the translation icon



23. At the upper right hand of the page, click on 'translation mode', the one by the left, denoted by 'A-Z'

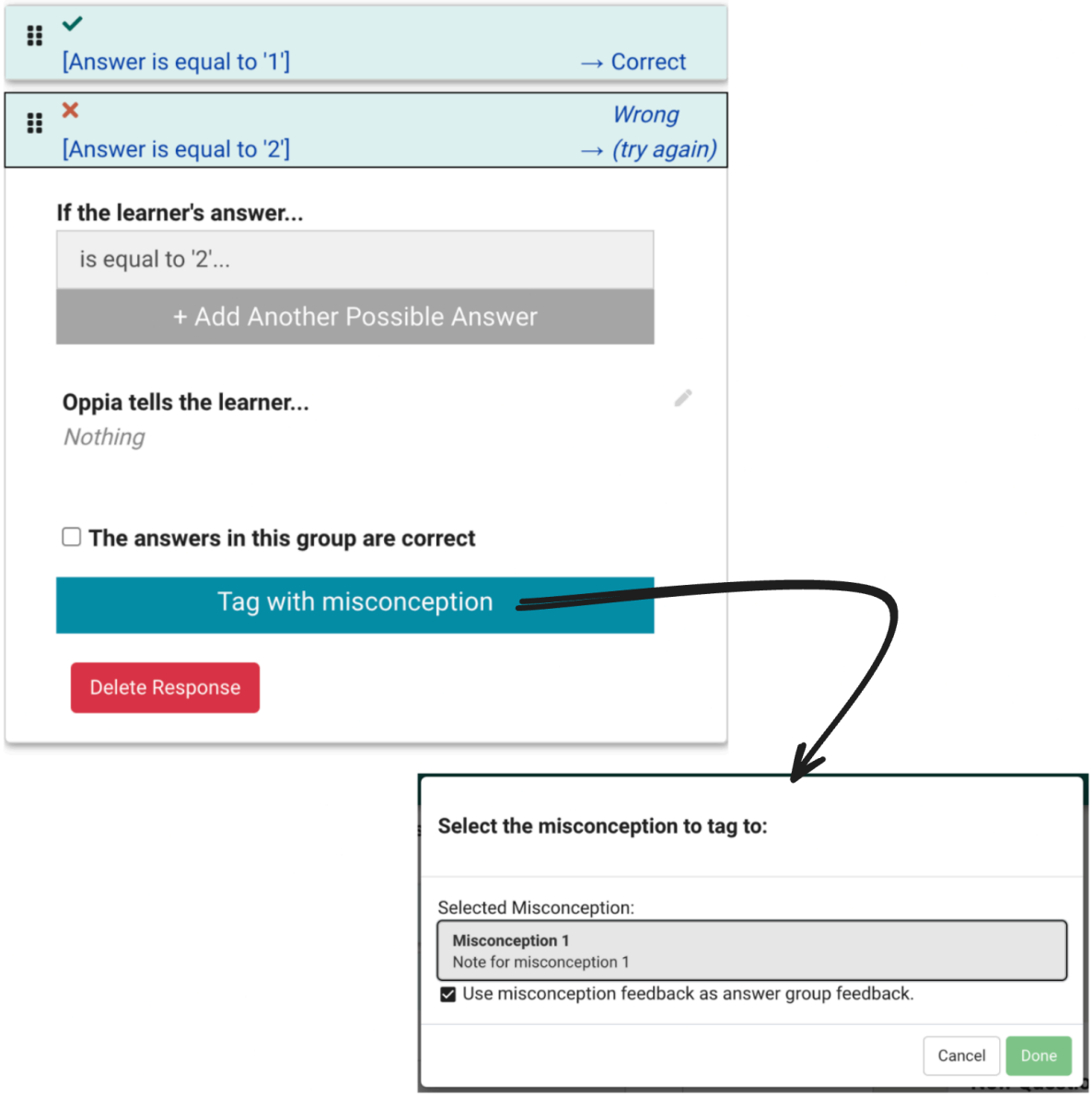
24. Go to the card you have selected, as in step 10 above.
25. Select the specific language you are working on. E.g. Arabic (**We currently have Arabic, Hindi, Pidgin, Portuguese and Spanish** available.)



As a result, this becomes really redundant and the creators or translators have to spend time unnecessarily on common translation changes - leading to decreased efficiency.

2. Exploration Misconceptions -
Currently, explorations at Oppia can often have missing feedback for learners, even in cases where the cards have skills containing compulsory misconceptions associated with them and it’s well known that the lesson can lead to specific misconceptions of learners being caught.

We already handle this in the skill question editor page by allowing questions to be tagged with misconceptions as follows -



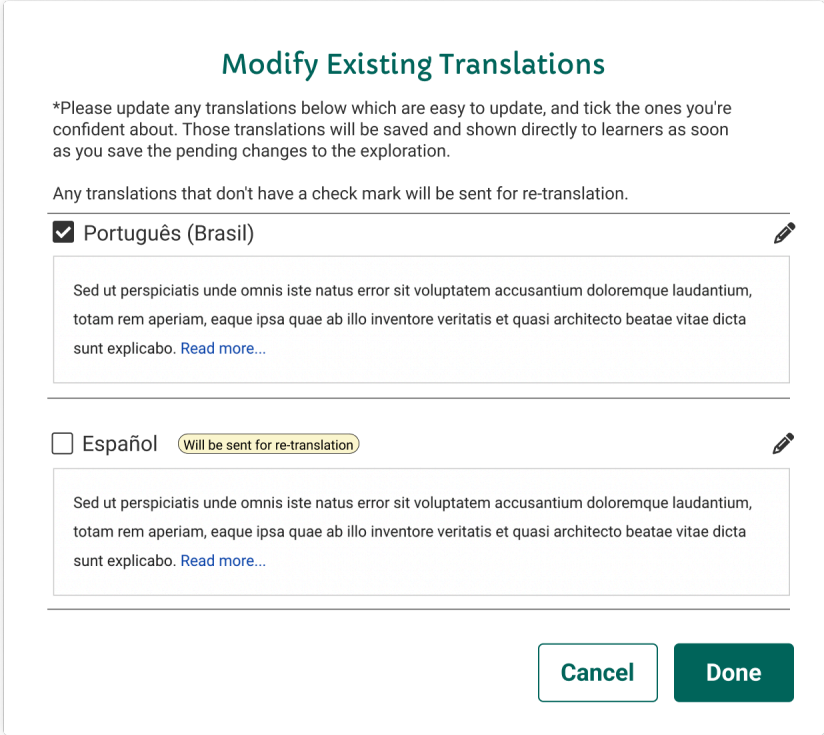
However, no such functionality to tag questions with misconceptions currently exists in the exploration editor page.

Also, one can see the list of unaddressed compulsory misconceptions in the skill question editor as well. This functionality also currently does not exist for the questions inside the exploration editor page.

All of this combined, makes it extremely difficult for the lesson creators to address the misconceptions of learners appropriately. The misconceptions of learners often aren't cleared in a convenient fashion, leading to a decreased quality of lessons.

Solution Overview

1. Translations Management -
   To address this issue of trivial changes being extremely tedious to deal with, when the user makes changes to a card in a curated exploration - and since the changes are trivial, decides to mark translations as stale - we implement a modal providing an interface to allow the user to update the translations themselves.



Users can choose to update only specific languages with the checkboxes - in cases where they can make the edits to some languages but not all. Translations that are unchecked will be treated as stale and sent back to CD.

The modal fetches the existing translations for the card, allowing the user to effectively update them, for trivial changes such as numbers or images that can very easily be done.

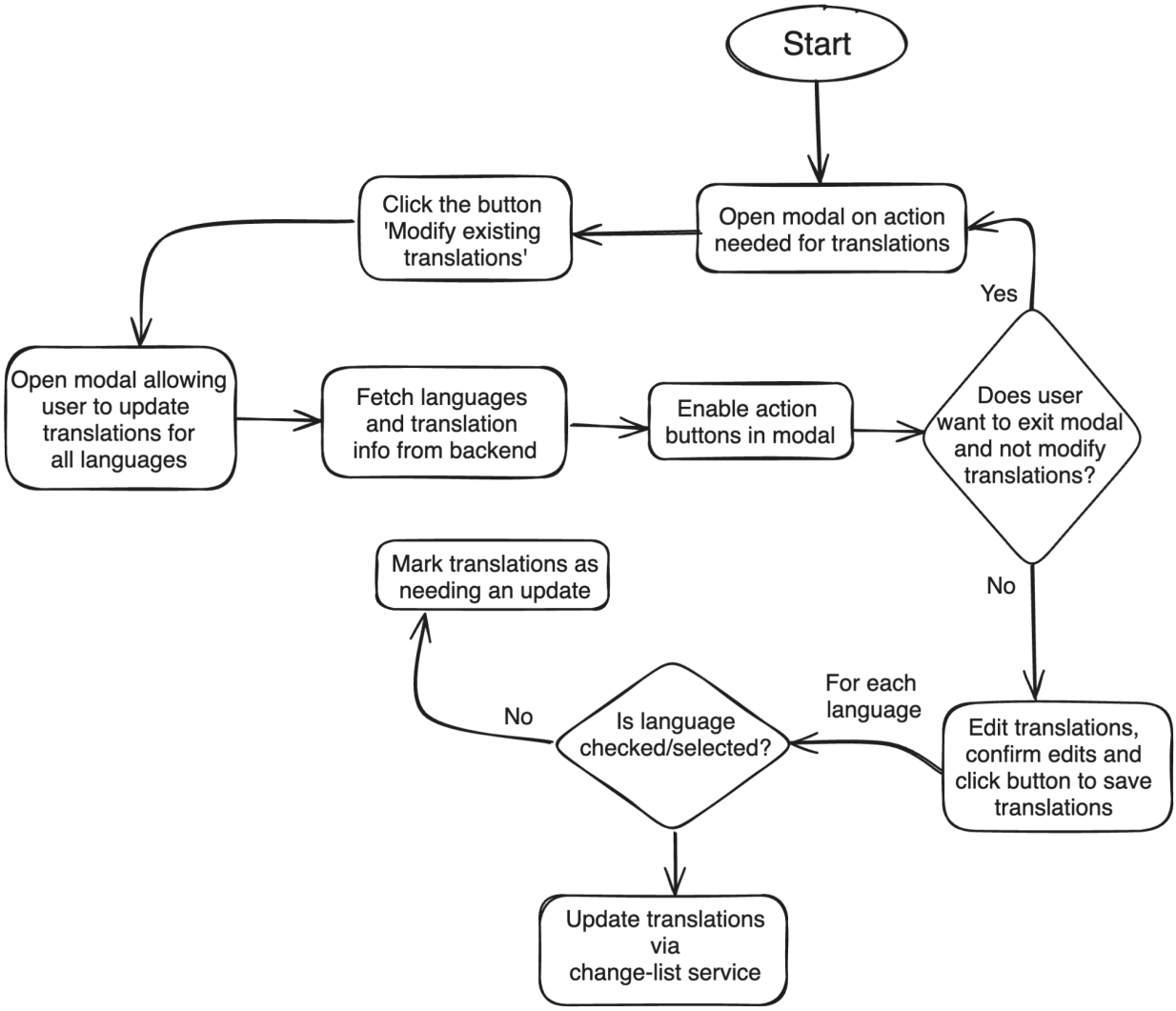
Clicking on a language to edit it will open the oppia-translation-modal used in the CD, which is beneficial especially for large translations and a large amount of languages since attempting to handle it all in a single simplified modal can make it very cluttered and difficult for the lesson creators to handle.

This approach drastically saves the time of translators in updating cards for trivial changes, improving overall productivity and efficiency.

This functionality can conveniently reuse existing interfaces for translation management such as the EntityTranslationBackendDict, ensuring consistency across the codebase.

```
28  export interface EntityTranslationBackendDict {
29    entity_id: string;
30    entity_type: string;
31    entity_version: number;
32    language_code: string;
33    translations: TranslationMappingDict;
34  }
```

Proposed workflow:

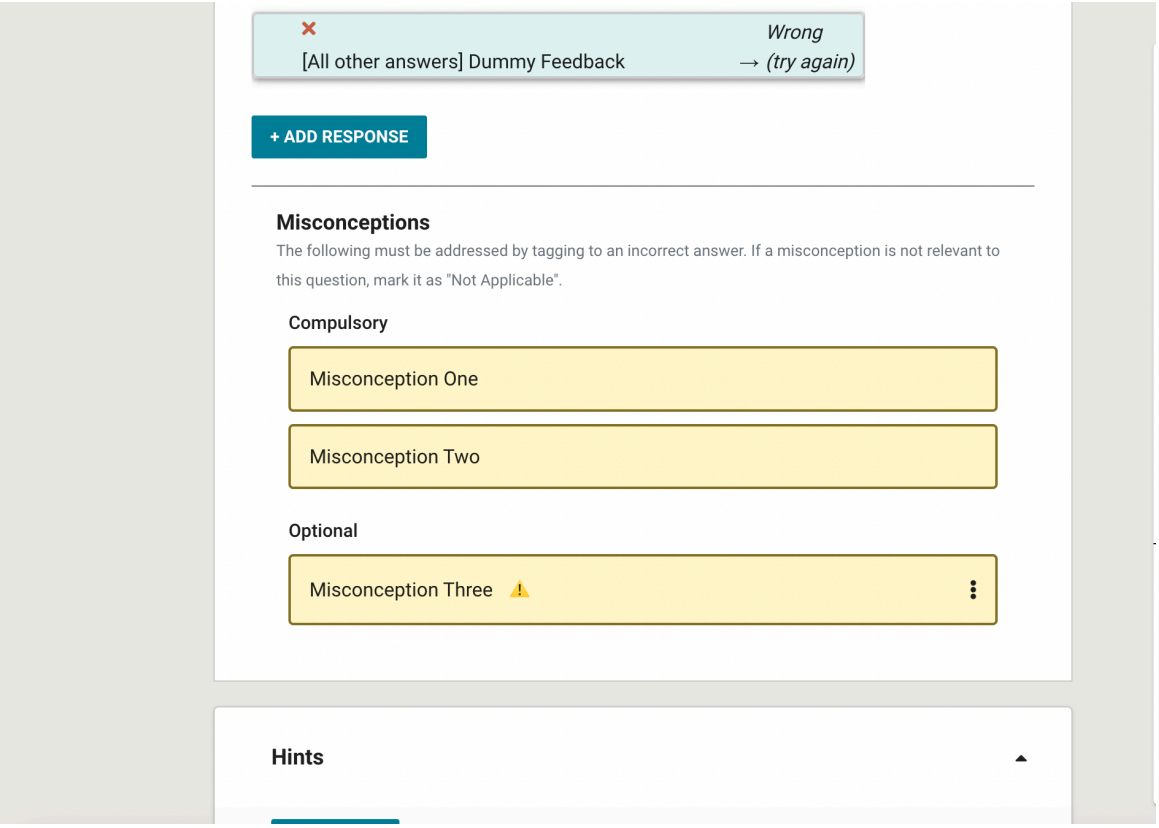


When the user decides that the current translations are stale as per the existing modal, they will then be asked to update the translations themselves if the changes are trivial, with action buttons disabled until translations are fetched, to avoid mishaps. These changes may include images and any type of content within the scope of the card, including unicode and rules.

The functionality of updating translations is flexible, allowing the user to update translations only for select languages since often trivial translations such as integers may use non-Arabic (e.g. Chinese) numerals that the creator may not be able to update easily.

2. Exploration Misconceptions -
To address this issue of misconceptions of learners being uncaught due to lack of feedback, we implement a convenient system to display a list of misconceptions for the card and provide functionality to tag responses with misconceptions in the exploration editor. These misconceptions are those associated with skills added to the cards.

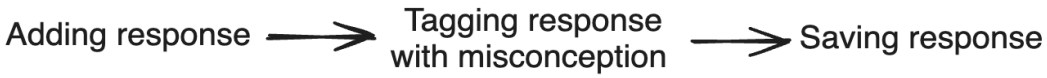
The user will see a list of misconceptions in the exploration editor, similar to the question editor -



List of misconceptions

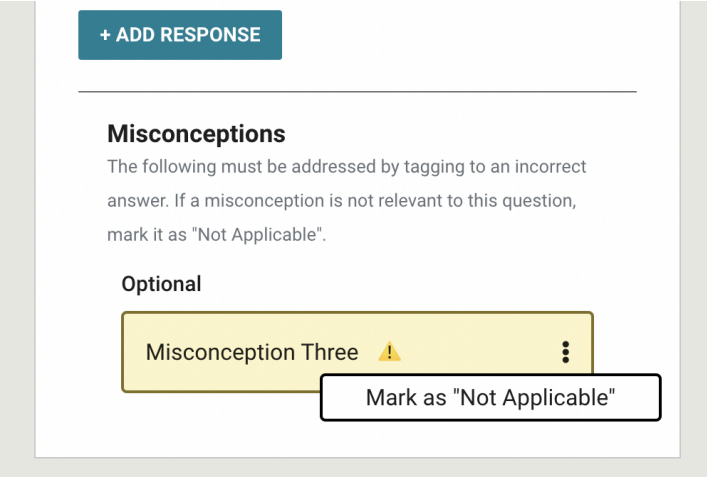
Following which, the user can appropriately tag response groups with misconceptions and ensure that all compulsory misconceptions have been addressed correctly. This will effectively avoid scenarios where misconceptions are not caught (and feedback is absent) even though it is known that the skill will likely result in learners making known errors.

Overall, tagging exploration responses will follow a very straightforward workflow similar to the current skill question editor -



All while being able to see a list of unaddressed misconceptions as mentioned above.

Also, the ability to mark non-mandatory/optional misconceptions as “not applicable” in cases where they don’t need to be addressed is very crucial for this functionality. This is to ensure that if an optional misconception is not addressed by a creator, it is an intentional choice and the misconception hasn’t been skipped by the user.



Currently, for the skill question editor page, this is being handled through managing the inapplicable_skill_misconception_ids data on the question domain object.

```python
class QuestionDict(TypedDict):
    """Dictionary representing the Question domain object."""

    id: str
    question_state_data: state_domain.StateDict
    question_state_data_schema_version: int
    language_code: str
    version: int
    linked_skill_ids: List[str]
    inapplicable_skill_misconception_ids: List[str]
    next_content_id_index: int
```

Since in the exploration editor, each page is a “state” to which the skills are linked, in order to implement this functionality in the exploration editor, we need to store this data on the state domain object as shown below.

```python
class StateDict(TypedDict):
    """Dictionary representing the State object."""

    content: SubtitledHtmlDict
    param_changes: List[param_domain.ParamChangeDict]
    interaction: InteractionInstanceDict
    recorded_voiceovers: RecordedVoiceoversDict
    solicit_answer_details: bool
    card_is_checkpoint: bool
    linked_skill_id: Optional[str]
    classifier_model_id: Optional[str]
    inapplicable_skill_misconception_ids: List[str]
```

For correct implementation, we will need to write a state schema migration job following the steps here. More information on the implementation is described below.

Third-Party Libraries

No.	Third-party library name and version	Link to third-party library	Why it is needed	License[10] (if third-party library)	[Android only] Min / target / max SDK version that the library supports
1	N/A				

"Service" Dependencies

No.	Dependency name	Why it is needed	What our plan is, if the dependency fails under us
1	N/A		

Impact on Other Oppia Teams

1. Translation Team - Translators will no longer have to spend time unnecessarily on trivial changes marked as stale, leading to a better workflow and improved efficiency.

Key High-Level and Architectural Decisions

Decision 1: For displaying the existing translations, we need to first fetch the existing language and translation data for a state from the backend.

We have considered the following alternatives:

[10]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.

1. Retrieving language data of complete translations

   We can use the '/explorehandler/init/{id}' API call as it is, to very easily fetch the language codes for which translations currently exist for the exploration.

   This would be very convenient, however, would only work for translations that are "complete" or only for displayable language codes.

   Also, we don't modify the /explorehandler endpoint to fetch all translations including incomplete translations optionally because -

   a. It's used for initializing explorations, where we do not want to display incomplete translations ever.
   b. It would still fetch a huge amount of exploration data we do not need, if we fetch strictly the translations with a condition then it wouldn't make sense for it to be in the explorehandler.

   Accordingly, we can fetch the current translations for the respective card for each completed language with these fetched codes.

2. Creating an API endpoint to retrieve and fetch all translations using existing functionality

   Another approach for dynamically fetching data to take incomplete translations into consideration can be iterating through all language codes and checking if the translation exists for that particular state, and then returning the list of languages with their corresponding translations together.

   To make this efficient and avoid a large number of API calls each time, we incorporate a new API call that does what we want - fetch the translations in bulk.

   There already exists a 'get_all_entity_translations_for_entity' functionality for doing exactly this - which makes the process pretty efficient, we just provide an endpoint to it. In terms of L and N where L is the number of languages and N is the number of content IDs, the query complexity is just L since N isn't a deciding factor in the query, and the response payload size is L*N, which is as minimal as it can be taking into consideration the current translation models.

   (We cannot use projection queries here to minimize the payload since `translations` is a JSON property, and creating a new optimized model seems like it might be too much work at this time when we aren't sure about whether payload size would be an issue in practice.)

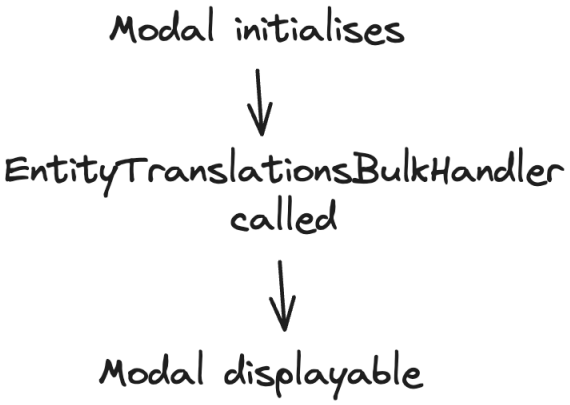
```python
def get_all_entity_translations_for_entity(
    entity_type: feconf.TranslatableEntityType,
    entity_id: str,
    entity_version: int
) -> List[translation_domain.EntityTranslation]:
    """Returns a list of entity translation domain objects.

    Args:
        entity_type: TranslatableEntityType. The type of the entity whose
            translations are to be fetched.
        entity_id: str. The ID of the entity whose translations are to be
            fetched.
        entity_version: int. The version of the entity whose translations
            are to be fetched.

    Returns:
        list(EnitityTranslation). A list of EntityTranslation domain objects.
    """
    entity_translation_models = (
        translation_models.EntityTranslationsModel.get_all_for_entity(
            entity_type, entity_id, entity_version)
    )
    entity_translation_objects = []
    for model in entity_translation_models:
        domain_object = _get_entity_translation_from_model(model)
        entity_translation_objects.append(domain_object)

    return entity_translation_objects
```

Let's call our API 'EntityTranslationsBulkHandler', it serves the sole purpose of calling the aforementioned function and returning serialized data accordingly.



As depicted above, the flow of data is extremely straightforward with only one API call being needed, all while **no** new functionality has to be added.

The response of the API call can look something like the below image, having the content identifiers as the keys, making it a simple process to later on update the translations from the modal.

Among these, we believe that creating an API endpoint to retrieve and fetch all translations using existing functionality, because:

- It's very straightforward and reuses existing functionality, while also addressing all translations instead of just the displayable translations, and without needing storage changes.

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

	Alternative 1: Dynamically retrieving language data through the existing /explorehandler/init API	**Alternative 2: Creating an API call to retrieve and fetch all translations using existing functionality**
Implementation difficulty	Less complex, doesn't require changes to current storage structures in the backend.	Straightforward, provides a simple endpoint to the backend.
Migration	No migration necessary, utilizes current exploration objects - reducing unnecessary complexity.	No migration of data, reuses existing functionality to fetch relevant data.
Practicality	Updates can only be made to current translations which have been completed.	Updates can be made for all translations, including those which aren't displayable
Number of interactions	**1** interaction at the outset and then **N** interactions on demand where N is the number of languages.	**1** on demand every time the modal is displayed.
Query and time complexity	Only once at the start, **8+** GET requests, in **O(N*M)** time, where N is the number of languages and M is the	Single GET request, in **O(N)** time, where **N** is the number of languages.

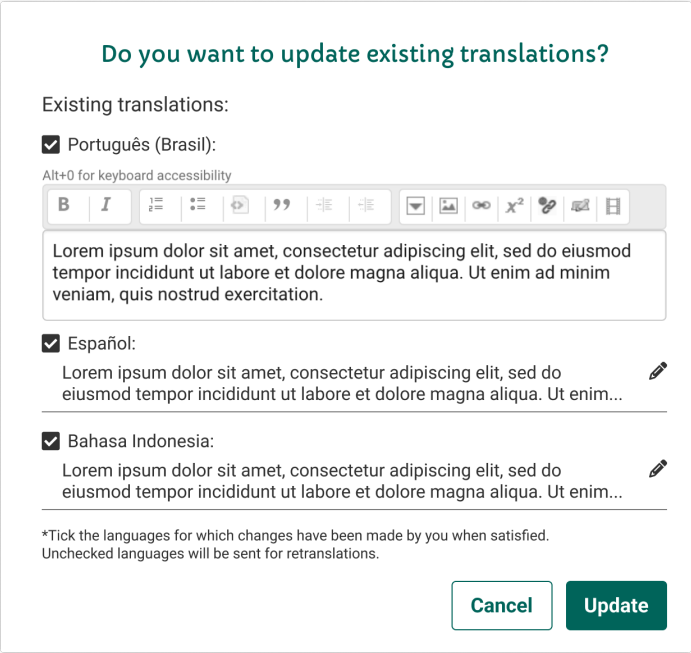
	number of content items of a single translation. Then for every single language, **N** GET requests to the datastore where N is the number of languages, in O(1) time.	
Size of payload	Unnecessarily extremely large, having all the data of the exploration.	As minimal as possible, having strictly the translation data of the exploration, but still contains unnecessary data of irrelevant content IDs.

Decision 2: Frontend modals for editing translations

We have considered the following alternatives:

1. Using a single modal to handle all translations at once

   This approach includes handling translations for all languages available through a single modal, at once, in a way similar to the following depiction -



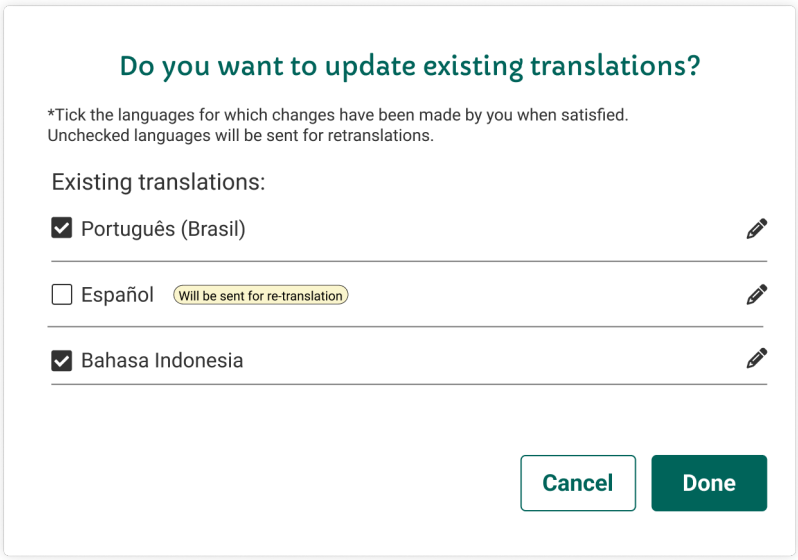
   This approach looks very convenient for trivial translations when the content is short. However, there are fundamental problems with it as it can become really messy in cases of content having a large amount of characters.

We can create scrollbars and limit heights to mitigate this, as depicted in the mocks but it no longer remains convenient for the end user when the content is large and there are a lot of languages simultaneously.

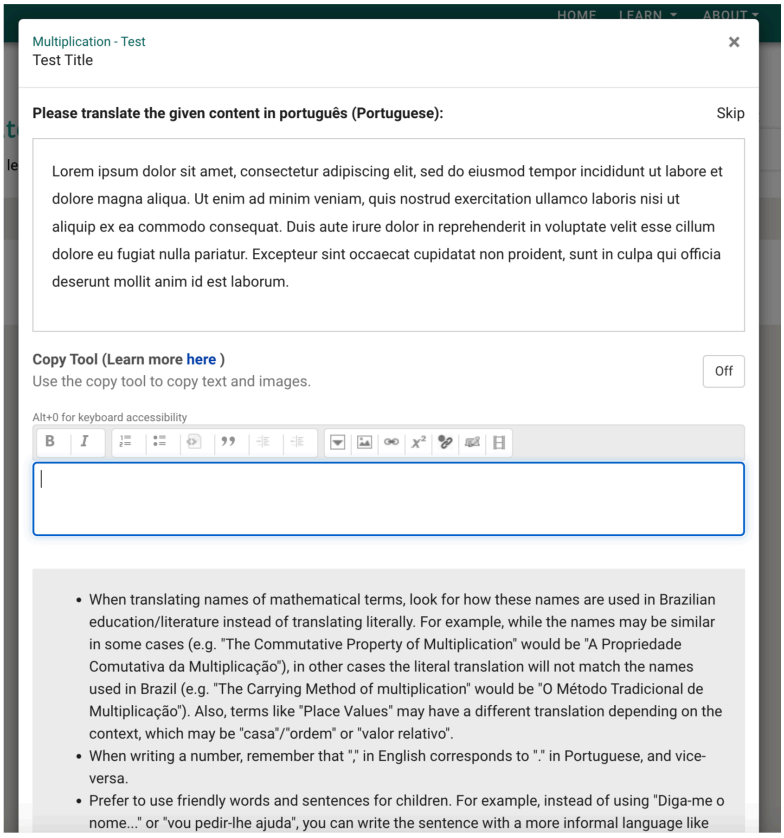
Another problem this approach which seems convenient at first glance faces is that the English content is not visible, which might be crucial for large content even if the change is trivial.

2. Reusing the CD translation editor component separately for each language

   We can turn our modal into strictly a list of languages and a gateway or a panel for the end user to edit corresponding translations in a clean and systematic way.



   Following which, we reuse the contributor dashboard translation editor component called oppia-translation-modal, to allow users to make edits -



This approach makes it very convenient for the creator especially in cases where the content is too large or when there are big images and such. Also, since it involves reusing the current CD modal component, it ensures a consistent way to update translations.

Another benefit of this approach is that the user can see the English content on top if needed, for cases where it can be confusing to make edits even if the change is trivial.

Among these, we believe that reusing the CD translation editor component separately for each language is the best approach, because:

- It is cleaner and more systematic comparatively.
- It is more convenient for a large number of languages.
- It is more convenient for a large amount of content.

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

	Alternative 1: Using a single modal to handle all translations at once	**Alternative 2: Reusing the CD translation editor component separately for each language**
Large number of languages	It can become easily cluttered and confusing, if there's a large amount of languages.	It remains clean and systematic, even with a large number of languages.

Large amount of content	A large amount of content can become confusing even with only a handful of languages, since this approach handles all translations in a single modal.	A large amount of content isn't a concern since it uses the already existing translation modal, which was designed to cater to large content sizes.
Small amount of content	It's very convenient for a small amount of languages and content, since the user can directly make the edits and update them without any hassle.	It can become a hassle if the amount of content is small, such as a single integer for an answer group, since the user will have to open the modal for every language one-by-one.
Convenience	It is less convenient for complex content, since the end user cannot see the original English content which might be necessary.	It is more convenient for the end users since they can see the original English content as part of the CD translation component.

Risks and mitigations

Potential Risk	**Mitigation**
Users might accidentally click on the button to update translations before they have loaded and have been changed/verified.	Disable the action buttons of the modal until translation data for each language has been fetched.
Translation modal might overflow the screen in case of an abundance of languages.	Set a max-height for the modal such that an internal scrollbar is created in case of content overflow.

Exploration Misconceptions -

Decision 1: Misconception component functionality in exploration editor

We have considered the following approach -

1. Reusing current skill question editor components

   We can generalize the current skill question editor components and services, reusing them in the exploration editor for faster development.

   These components include:
   a. "Tag With Misconception" button
   b. List of unaddressed misconceptions
   c. "Select misconception to tag to" modal

Reusing these components would enforce consistent UI across different parts of the site, simpler maintenance in case of future updates or bug fixes, as well as an easier development process building on top of existing schemas.

We believe that reusing current skill question editor components is the best and most suitable approach, because:

- This would prevent duplication of identical components, ensuring consistency across the site and reusability of services.

Risks and mitigations

Potential Risk	**Mitigation**
Existing explorations would become unsavable if tagging misconceptions is compulsory.	Don't incorporate validation checks currently as explorations will be updated on an incremental basis.
Some non-mandatory misconceptions might not be relevant to the card.	Allow users to appropriately mark non-mandatory misconceptions as not applicable for a particular question in the exploration editor itself, just like the question editor.

Implementation Approach

Translations Management

User Flows (Controllers and Services)

General user flow as mentioned above

For implementation of the required user flow, changes would mainly have to be made to the MarkTranslationsAsNeedingUpdate modal component, which is instantiated by the ExplorationStates service of the exploration editor page.

1. We would create a new 'update-translations' modal component in the exploration-editor-page/modal-templates directory.

```
@Component({
  selector: 'oppia-exploration-update-translations-modal',
  templateUrl: './exploration-update-translations-modal.component.html',
})
export class UpdateTranslationsModalComponent extends ConfirmOrCancelModal {

  constructor(private ngbActiveModal: NgbActiveModal) {
    super(ngbActiveModal);
  }
```

2. In the markNeedsUpdate() function, we will make changes to have it close the old modal and open this new modal component, instead of directly marking translations as stale.

3. When the user does choose to update existing translations, the modal has to display all existing translations. As mentioned above, we can fetch both the language codes and the corresponding translations very conveniently in a single API call by simply providing an endpoint to an existing function.

   This function is defined in a translation fetchers domain already, existing in the domain as get_all_translations_for_entity.

   Whether a translation is stale or not can be determined by the “needs_update” property of the translation, and in such cases, we do not display such languages or allow any ability to update them.

   We can add an endpoint to it by simply defining an EntityTranslationsBulkHandler controller in editor.py, as follows -

```python
class EntityTranslationsBulkHandler(
    base.BaseHandler[Dict[str, str], Dict[str, str]]
):
    """The handler to fetch all available translations for a
    given entity.
    """

    GET_HANDLER_ERROR_RETURN_TYPE = feconf.HANDLER_TYPE_JSON
    URL_PATH_ARGS_SCHEMAS = {…
    HANDLER_ARGS_SCHEMAS: Dict[str, Dict[str, str]] = {
        'GET': {}
    }

    @acl_decorators.open_access
    def get(
        self,
        entity_type: str,
        entity_id: str,
        entity_version: int,
    ) -> None:
        translations = {}
        entity_translations = translation_fetchers.get_all_entity_translations_for_entity(
            feconf.TranslatableEntityType(entity_type), entity_id,
            entity_version)

        for translation in entity_translations:
            translations[translation.language_code] = translation.to_dict()

        self.render_json(translations)
```

It would be a simple GET request, returning the relevant language codes along with the respective translations in the response, as depicted above.

Changes in main.py for linking the functionality with the route -

```python
get_redirect_route(
    r'/entity_translations_bulk_handler/<exploration_id>/<version>',
    editor.EntityTranslationsBulkHandler),
```

4. Now, since the above request has returned translations associated with content items of the entire exploration, we need to specifically target the translations we want.

   Considering the API response that follows the format of the below image, we iterate through each available language code and check if our unique content ID such as “hint_32” for example exists in the language or not.

   We create a dict having the language and the translation as the key-value pairs, and use it in our translations modal to populate the languages and the content. We don’t store this dict, and the translation changes are stored via an already existing entity translations service.

   {

   “de”: “ Content in German ”,

   “fr”: “ Content in French ”

}

5. Now that we have fetched the language codes as well as the corresponding translations conveniently with a single call, this data can be rendered in the modal and displayed to the user on the frontend - allowing the user to make changes.

   This enables the user to make changes to the translations for all existing languages via a single modal - without any limitations, and without having to go to the translations tab and make edits for every language one by one.

6. When the user opens the translations tab for the content, they will see the newly changed translation instead of the old one even if changes haven't been published yet - and vice versa, the modal will display new translations even if they haven't been published yet.

   This is currently already facilitated in the translations tab, by utilizing the entity translations service responsible for fetching and storing translations for a given entity in a given language, and it is managed as changes related to translations are appended to the change list.

   We also make adjustments to have our modal check if edits to translations already exist, either from the translations tab or any previous modals, and if they do, then we show the new unpublished translations instead of the currently published translations.

```
this.changeListService.editTranslation(
  this.contentId,
  this.languageCode,
  this.activeWrittenTranslation
);
this.entityTranslationsService.languageCodeToEntityTranslations[
  this.languageCode
].updateTranslation(this.contentId, this.activeWrittenTranslation);
```

7. These changes won't be pushed immediately to the backend, and they would rather follow the flow of the exploration editor page and utilize functionalities of the change-list service - specifically the editTranslation function. After changes have been finalized, all that's left is to publish the exploration and push these changes to the backend.

```
/**
 * Saves a change dict that represents editing translations.
 */
editTranslation(
  contentId: string,
  languageCode: string,
  translatedContent: TranslatedContent
): void {
  this.addChange({
    cmd: 'edit_translation',
    language_code: languageCode,
    content_id: contentId,
    translation: translatedContent.toBackendDict(),
  });
}
```

Below is a depiction of change list items for edited and stale translations respectively -

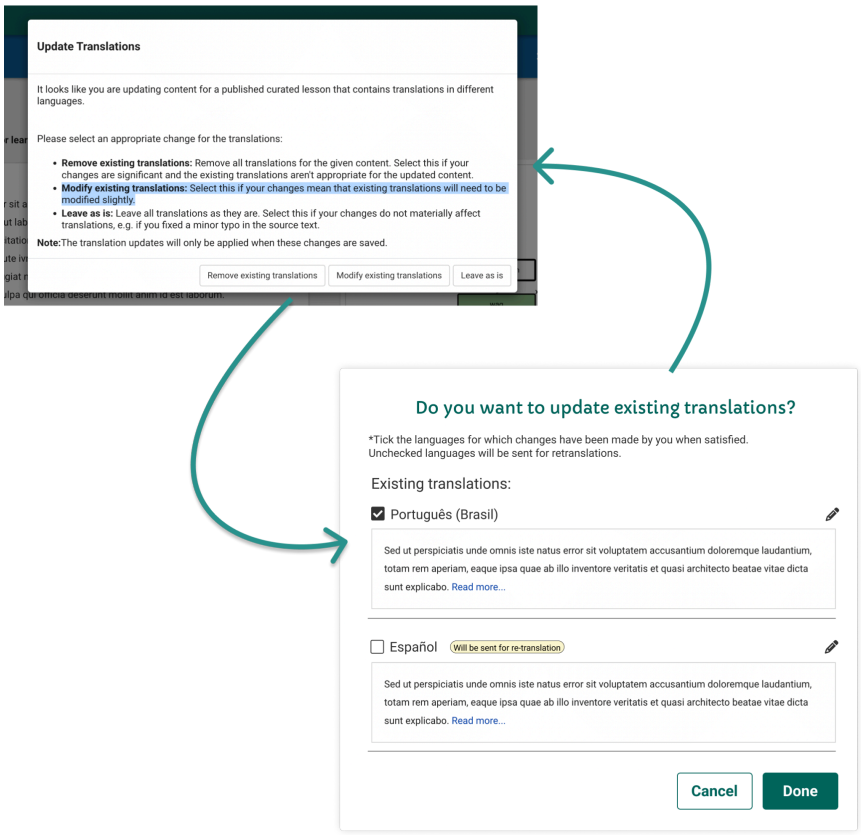
```
{
  "cmd": "edit_translation",
  "language_code": "pt",
  "content_id": "content_0",
  "translation": {
    "content_value": "<p>Edited portuguese translation</p>",
    "content_format": "html",
    "needs_update": false
  }
},
{
  "cmd": "edit_translation",
  "language_code": "fr",
  "content_id": "content_0",
  "translation": {
    "content_value": "<p>Stale french translation</p>",
    "content_format": "html",
    "needs_update": true
  }
}
```

8. Above change list items are only stored when the user finishes the complete flow and clicks the 'Done' button at the end. The unchecked languages will be added to the list as needing update and the checked languages will be added with the new translations and don't need an update.

   If a user makes some changes and then doesn't check the language, it will only be marked as needing an update and the edits won't be considered.

9. When the user decides to cancel the modal and hits the 'Cancel' button in case they made the wrong choice, it will take them back to the initial modal that took them here. Also, note that the

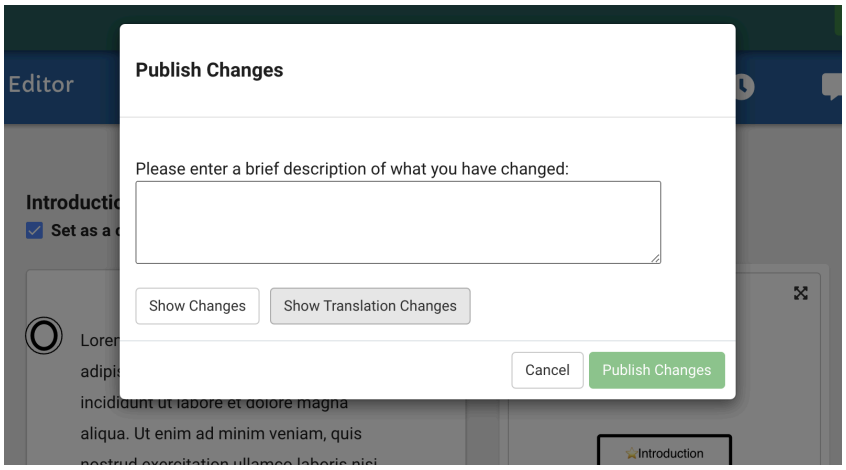
initial modal cannot be cancelled, and the user has to decide what shall be done with current translations.



We do this by reusing the functionality in [exploration-states] service that opened the modal in the first place, and the English content in the editor is populated via this service as well. We do not need to specifically reset any changes in case a user decides to cancel and then go back to the modal, since the modal starts anew when it's opened.

```
const modalRef = this.ngbModal.open(
  MarkTranslationsAsNeedingUpdateModalComponent,
  {
    size: 'lg',
    backdrop: 'static',
    // TODO(#12768): Remove the backdropClass & windowClass once the
    // rte-component-modal is migrated to Angular. Currently, the custom
    // class is used for correctly stacking AngularJS modal on top of
    // Angular modal.
    backdropClass: 'forced-modal-stack',
    windowClass: 'forced-modal-stack',
  }
);
modalRef.componentInstance.contentId = contentId;
modalRef.componentInstance.markNeedsUpdateHandler =
  this.markTranslationAndVoiceoverNeedsUpdate.bind(this);
modalRef.componentInstance.removeHandler =
  this.removeTranslationAndVoiceover.bind(this);
this.initalContentsMapping[contentId] = content;
}
```

10. And finally, to ensure the most optimum user experience, when the user decides to publish their changes, we give them the ability to review the translation changes they've made in addition to the already existing functionality for reviewing changes.



We already store the new content, however, the old content isn't stored anywhere. So in order to facilitate this functionality, we maintain a dict of old content for only translations that have been modified. This dict will be stored in the already existing entity translations service responsible for fetching and storing translation data.

We replicate the dict used for storing latest translations to ensure consistency as follows,

```
// Dictionary storing translations that exist in the last published version
// of the exploration.
public languageCodeToLastPublishedEntityTranslations: LanguageCodeToEntityTranslations =
  {};

// Dictionary storing new translations that were modified in the editor and haven't
// been published yet.
public languageCodeToUpdatedEntityTranslations: LanguageCodeToEntityTranslations =
  {};
```

where the interface used is as shown below -

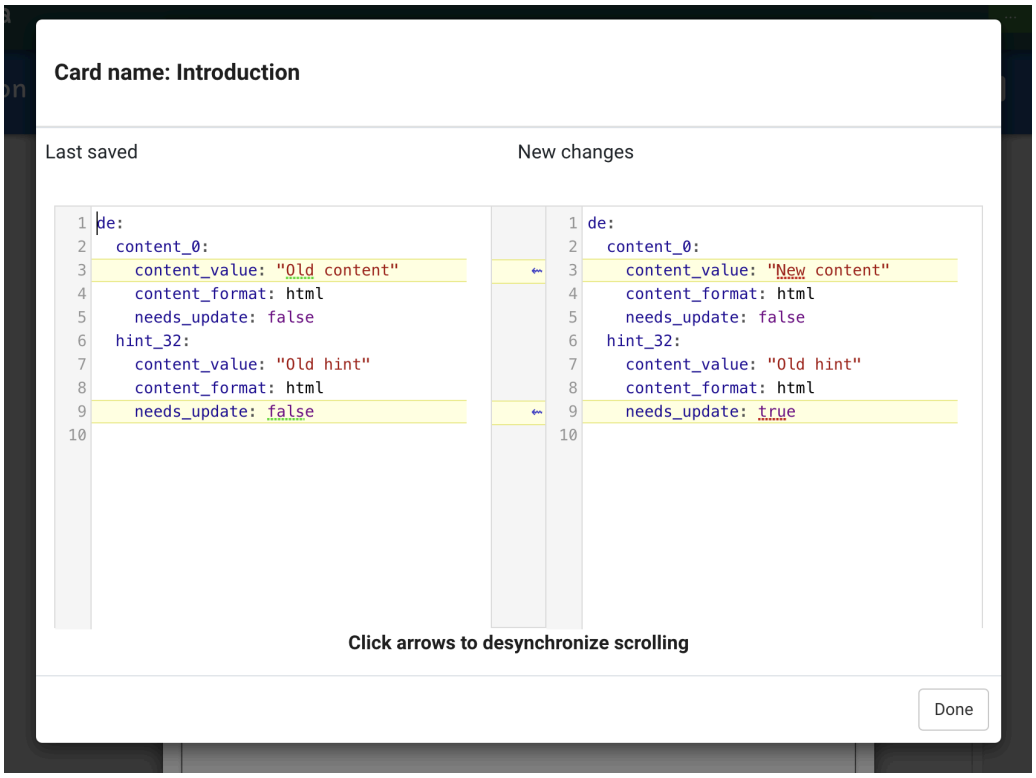
```
27  export interface LanguageCodeToEntityTranslations {
28    [languageCode: string]: EntityTranslation;
29  }
```

Above dict storing original, last published translations is updated whenever a user fetches existing translations from the backend - and is also utilized as a cache system in case the data is again needed in the future, instead of making additional backend API calls. This includes the translation modals as well as the translation tab.

The dict storing updated translations is updated every time changes are added to the change list to ensure it has the latest changes to be published.

By this approach, we can conveniently maintain the dict using the EntityTranslation object's functionality to update translations and such.

It becomes all the more simpler since both - the old and the new dicts are of the same type and follow a similar structure. When the user wishes to view translation changes, the old translations dict is contrasted side-by-side with the new changes similar to the below image.



[Web only] Web frontend changes

The implementation of this system is mainly based around modals and does not interfere with the current frontend UI elements.

Exploration Misconceptions

User Flows (Controllers and Services)

The implementation of this system is pretty straightforward, with services and components mostly being reused - as it mainly focuses on extending the functionality of the skill question editor page to the curated exploration editor page. We plan to specifically target curated explorations by utilizing the `'explorationIsLinkedToStory'` check already existing in the state-editor component.

We don't need to store associated misconceptions since those depend on skills, which are already a part of the state. We also don't need additional implementation to store the linked misconception IDs, since those are already a part of the answer response group objects.

To fetch misconceptions and keep them up-to-date, we load the skill objects in the exploration editor in a convenient way with a simple call to the skill-backend-api-service, similar to how it's done for the question editor. This call happens when the exploration editor is initially loaded, as well as when a skill is removed or added to the state, to maintain consistency.

We determine the skill ID used to fetch misconceptions via the state-editor service's `'linkedSkillId'` variable that is initialized with the state. We store the misconception data in the same state-editor service, similar to how it's done for the question editor, using the already existing `'misconceptionsBySkill'` variable.

For cases of a skill being changed or removed after a misconception has been tagged to a response group, as we fetch the misconception IDs, we verify that if misconceptions previously tagged to any response groups still exist in the state, through the response group objects that have been fetched as a part of the state. This also happens on initial load in case a misconception was removed from a skill through the skill editor, after it was tagged to a response in the exploration editor.

Besides this, we do need to write a state schema migration job in order to store the inapplicable misconceptions associated with the state in the backend. The process is very clear and following the detailed instructions here, below is a very rough depiction of the migration.

```python
class AnswerGroup(translation_domain.BaseTranslatableObject):
    """Value object for an answer group. Answer groups represent a set of rules
    dictating whether a shared feedback should be shared with the user. These
    rules are ORed together. Answer groups may also support a classifier
    that involve soft matching of answers to a set of training data and/or
    example answers dictated by the creator.
    """

    def __init__(
        self,
        outcome: Outcome,
        rule_specs: List[RuleSpec],
        training_data: List[str],
        tagged_skill_misconception_id: List[str]
    ) -> None:
        """Initializes a AnswerGroup domain object.
```

We make changes to feconf.py and exp_domain.py to increment the schema versions, and write the method to convert the states in exp_domain.py as follows, where 61 is the new exploration schema version and 56 is the new state schema version.

```python
    @classmethod
    def _convert_states_v55_dict_to_v56_dict(
        cls, states_dict: Dict[str, state_domain.StateDict]
    ) -> Tuple[Dict[str, state_domain.StateDict], int]:
        """Converts from v55 to v56. Version 56 adds an
        inapplicable_skill_misconception_ids list to the State.
        """
        for _, state_dict in states_dict.items():
            state_dict['inapplicable_skill_misconception_ids'] = []
        states_dict = state_domain.State

        return states_dict
```

```python
    @classmethod
    def _convert_v60_dict_to_v61_dict(
        cls, exploration_dict: VersionedExplorationDict
    ) -> VersionedExplorationDict:
        """Converts a v60 exploration dict into a v61 exploration dict.
        Introduces the inapplicable_skill_misconception_ids list into
        the state properties.

        Args:
            exploration_dict: dict. The dict representation of an exploration
                with schema version v60.

        Returns:
            dict. The dict representation of the Exploration domain object,
            following schema version v61.
        """
        exploration_dict['schema_version'] = 61

        exploration_dict['states'] = (
            cls._convert_states_v55_dict_to_v56_dict(
                exploration_dict['states'])
        )
        exploration_dict['states_schema_version'] = 56

        return exploration_dict
```

And finally, we need to write the conversion function in draft_upgrade_services.py to upgrade the state version from 55 to 56.

```python
    @classmethod
    def _convert_states_v55_dict_to_v56_dict(
        cls, draft_change_list: List[exp_domain.ExplorationChange]
    ) -> List[exp_domain.ExplorationChange]:
        """Converts draft change list from state version 55 to 56. Version 56
        adds an inapplicable_skill_misconception_ids list property to the
        state.

        Args:
            draft_change_list: list(ExplorationChange). The list of
                ExplorationChange domain objects to upgrade.

        Returns:
            list(ExplorationChange). The converted draft_change_list.

        Raises:
            InvalidDraftConversionException. The conversion cannot be
                completed.
        """
        for exp_change in draft_change_list:
            if exp_change.cmd == exp_domain.CMD_EDIT_STATE_PROPERTY:
                raise InvalidDraftConversionException(
                    'Conversion cannot be completed.')
        return draft_change_list
```

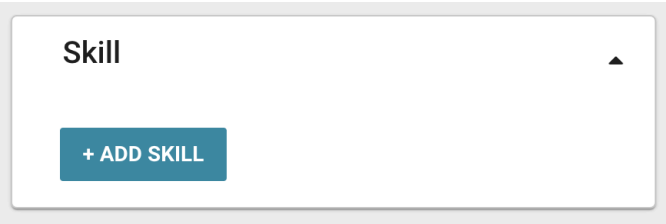
Now, for the frontend, we can reuse existing interfaces from MisconceptionObjectFactory for implementation -

```typescript
export interface MisconceptionBackendDict {
  feedback: string;
  id: number;
  must_be_addressed: boolean;
  name: string;
  notes: string;
}
export interface MisconceptionSkillMap {
  [skillName: string]: Misconception[];
}

export interface TaggedMisconception {
  skillId: string;
  misconceptionId: number;
}
```

And additionally, we reuse the current state-editor service to fetch relevant skill-misconception data, similar to how it's currently done for the question editor page, keeping track of addressed and unaddressed misconceptions for the card.

1. Skills are added to a card of a curated exploration, with the help of the state-skill-editor component, inside the exploration editor page.



2. We can then fetch the linked skill IDs for the particular card from the current exploration-backend services and interfaces, without having to implement any new functionality

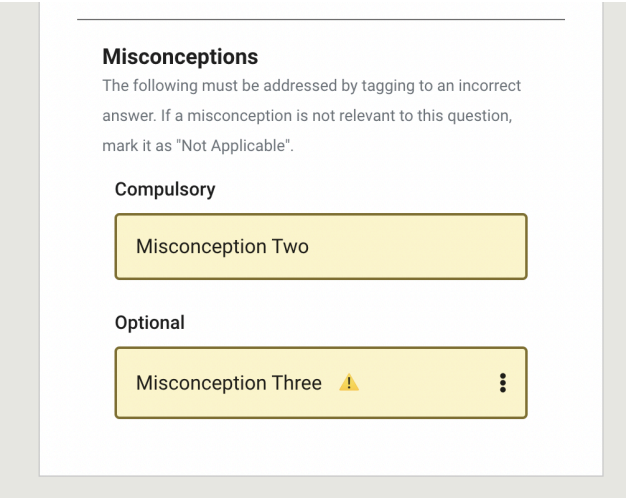
   This dictionary returned by the exploration backend service ultimately contains data of the state objects from the backend, including the linked skill IDs.

```
export interface StateBackendDict {
  // The classifier model ID associated with a state, if applicable,
  // null otherwise.
  classifier_model_id: string | null;
  content: SubtitledHtmlBackendDict;
  interaction: InteractionBackendDict;
  param_changes: readonly ParamChangeBackendDict[];
  recorded_voiceovers: RecordedVoiceOverBackendDict;
  solicit_answer_details: boolean;
  card_is_checkpoint: boolean;
  // This property is null if no skill is linked to the State.
  linked_skill_id: string | null;
}
```

3. Now that we have the IDs of the linked skill, the misconception data can again be fetched from existing services - specifically the skill-backend-api service in this case.

```
export interface SkillBackendDict {
  all_questions_merged: boolean;
  description: string;
  id: string;
  language_code: string;
  misconceptions: MisconceptionBackendDict[];
  next_misconception_id: number;
  prerequisite_skill_ids: string[];
  rubrics: RubricBackendDict[];
  skill_contents: ConceptCardBackendDict;
  superseding_skill_id: string;
  version: number;
}
```

4. We do not need a new component to show the list of misconceptions in the editor, since it's already a part of the state-responses-component but hidden under a check to determine if the editor is in question mode.



5. The main thing to be done here for the implementation of this system is transforming the current 'Tag with misconception' button and the misconception tagging modal to be common and fully functional between both, the question editor page and the exploration editor page.

   This button again along with the functionality to edit already tagged misconceptions, is a part of the question-misconception-editor component that is hidden from the answer-group-editor and the add-answer-group modal via a check to determine if the editor is in question mode.

   This misconception editor component is the core element we need to generalize, and to facilitate this common ground, the key thing we need to do is initialize the misconceptions in the state editor service, which is currently done only for the question editor and not for the exploration editor.

   Here, the misconceptionsBySkill are the misconceptions fetched from the skill-backend-api service mentioned previously.

```
this.stateEditorService.setMisconceptionsBySkill(
  this.misconceptionsBySkill
);
```

[Web only] Web frontend changes

This project changes the frontend of the exploration-editor page in a few ways, with the addition of a list of misconceptions displayed to the user, similar to the current question editor page - and consequently, buttons to tag response groups with misconceptions for the respective skill.

There are no changes to the current frontend elements besides the two mentioned - the button and the misconceptions list, as the system depends upon reusing current modals to handle tagging misconceptions.

Testing Plan

Acceptance tests for core user flows

#	Test name	Initial setup steps	Steps	Expectations
1.	As a lesson creator, the user can update translations for the following in an exploration - 1. Main content 2. Interactions 3. Hints 4. Responses 5. Solutions And finally, 6. Cancel the modal flows appropriately	- Login as curriculum_admin@example.com (curriculum admin account) - Go to the creator dashboard, create and publish a test exploration with the following states - 1. Minimal content 2. Multiple choice - - Add a multiple choice interaction to the exploration, with appropriate groups. - Add a hint to the card. - Appropriately redirect a response to a card 3. 3. Text input - - Add a text input interaction to the exploration, with appropriate response groups. - Add a solution for the state card.	1.1 Open the exploration editor page for the 1st state, open the translations tab and switch to translation mode. Switch the language to German and add a translation for the main card. Publish the changes. Now, make an edit to the original card and on the popup, select 'Modify existing translations'.	1.1 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.
			1.2 Click on the edit button for the German language.	1.2 The CD editor should be seen having the entire German translation populated.
			1.3 Edit the translation to a different sentence and press 'Save'.	1.3 The CD editor modal should close and the previous modal with the list of translations should be focused.
			1.4 Tick the German language and click on 'Save Changes'.	1.4 The modal should close and the user should be able to publish changes.
			1.5 Publish the changes, reload the page and go to the translations tab. Select German and switch to translation mode. Select the relevant main card.	1.5 The newly updated translation from the modal should be visible.
			1.6 Now, open the translations tab and edit the same text there. Open the state editor and make a	1.6 The German translation here should be the recently updated one here from the translations tab.

		- Appropriately redirect a response to a new card to end the exploration, and publish it. - Go to the topic and skills dashboard and create a new topic. - Add a subtopic and a skill with 3 questions, and publish the skill. - Link the skill to the subtopic and add it as a diagnostic test skill. - Publish the topic. - Create a story with a chapter using the ID of the exploration created previously, and publish changes. - Go to the classroom admin page, create a classroom. - Add the topic created above to the classroom. - Login as exploration_editor@example.com (exploration editor account). This setup has created a curated exploration with the curriculum admin logged in.	change to the same card so that the modal pops up. Select 'Modify existing translations'.	
			2.1 Open the exploration editor page for the 2nd state, open the translations tab and switch to translation mode. Switch the language to German, select 'Interaction', click on a choice and add a translation to it. Publish the changes. Now, make an edit to the original choice and on the popup, select 'Modify existing translations'.	2.1 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.
			2.2 Click on the edit button for the German language.	2.2. The CD editor should be seen having the entire German translation populated.
			2.3 Edit the translation to a different sentence and press 'Save'.	2.3 The CD editor modal should close and the previous modal with the list of translations should be focused.
			2.4 Tick the German language and click on 'Save Changes'.	2.4 The modal should close and the user should be able to publish changes.
			2.5 Publish the changes, reload the page and go to the translations tab. Select German and switch to translation mode. Select interactions and choose the relevant multiple choice option.	2.5 The newly updated translation from the modal should be visible.
			3.1 Open the exploration editor page for the 2nd state, open the translations tab and switch to translation mode. Switch the language to German, select 'Hints', click on a hint and add a translation to it. Publish the changes. Now, make an edit to the original hint and on the popup, select 'Modify existing translations'.	3.1 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.

			3.2 Click on the edit button for the German language.	3.2 The CD editor should be seen having the entire German translation populated.
			3.3 Edit the translation to a different sentence and press ‘Save’.	3.3 The CD editor modal should close and the previous modal with the list of translations should be focused.
			3.4 Tick the German language and click on ‘Save Changes’.	3.4 The modal should close and the user should be able to publish changes.
			3.5 Publish the changes, reload the page and go to the translations tab. Select German and switch to translation mode. Select ‘Hints’ and choose the relevant hint.	3.5 The newly updated translation from the modal should be visible.
			4.1 Open the exploration editor page for the 3rd state, open the translations tab and switch to translation mode. Switch the language to German, select ‘Rules’, click on the relevant text input response group and add a translation to it. Publish the changes. Now, make an edit to the original response and on the popup, select ‘Modify existing translations’.	4.1 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.
			4.2 Click on the edit button for the German language.	4.2 The CD editor should be seen having the entire German translation populated.
			4.3 Edit the translation to a different sentence and press ‘Save’.	4.3 The CD editor modal should close and the previous modal with the list of translations should be focused.
			4.4 Tick the German language and click on ‘Save Changes’.	4.4 The modal should close and the user should be able to publish changes.
			4.5 Publish the changes, reload the page and go to the translations tab. Select German and switch to translation mode. Select ‘Rules’ and choose the relevant response rule.	4.5 The newly updated translation from the modal should be visible for the relevant part inside square brackets of the rule.

			5.1 Open the exploration editor page for the 3rd state, open the translations tab and switch to translation mode. Switch the language to German, select 'Solution', click on the relevant solution and add a translation to it. Publish the changes. Now, make an edit to the original solution and on the popup, select 'Modify existing translations'.	5.1 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.
			5.2 Click on the edit button for the German language.	5.2 The CD editor should be seen having the entire German translation populated.
			5.3 Edit the translation to a different sentence and press 'Save'.	5.3 The CD editor modal should close and the previous modal with the list of translations should be focused.
			5.4 Tick the German language and click on 'Save Changes'.	5.4 The modal should close and the user should be able to publish changes.
			5.5 Publish the changes, reload the page and go to the translations tab. Select German and switch to translation mode. Select the 'Solution' and click on the relevant solution.	5.5 The newly updated translation from the modal should be visible.
			6.1 Open the exploration editor page for the 1st state, open the translations tab and switch to translation mode. Switch the language to German and add a translation for the main card. Publish the changes. Now, make an edit to the original card.	6.1 A modal should be seen with a button to modify existing translations.
			6.2 Click on 'Modify existing translations'.	6.2 A modal allowing the user to update existing translations should be seen, with a list of languages - German in this case, and their corresponding translations visible.
			6.3 Click on 'Cancel' to close this new modal.	6.3 The original modal should be seen with a button to modify existing translations.

			6.4 Click on 'Leave as is' when prompted what action shall be taken for translations..	6.4 The modal should close and allow the user to make further edits to the exploration.
2.	As a lesson creator, the user cannot see misconception functionality for non-curated explorations.	- Login as curriculum_admin@example.com (curriculum admin account) - Go to the creator dashboard and create an exploration. - Add a multiple choice interaction to the exploration, with appropriate answer groups. - Add a hint to the card. - Appropriately redirect a response to a new card to end the exploration, and publish it. - Login as exploration_editor@example.com (exploration editor account). This setup has created a curated exploration with the curriculum admin logged in.	1. Open the exploration editor page for the created exploration.	1. A button to add a skill shouldn't be visible, and a list of misconceptions should not be visible.
			2. Click on an answer group.	2. A button to tag the answer group with a misconception should not be visible.
3.	As a curriculum admin, the user can - 1. Tag response groups with misconceptions for a curated exploration 2. Change the tagged misconception on an answer group 3. Mark optional misconceptions as inapplicable 4. Verify that misconceptions no longer exist are removed.	- Login as curriculumadmin@example.com (curriculum admin account) - Go to the creator dashboard and create an exploration. - Add a multiple choice interaction to the exploration, with appropriate answer groups. - Add a hint to the card. - Appropriately redirect a response to a new card to end the exploration, and publish it. - Go to the topic and skills dashboard and create a new topic. - Add a subtopic and a skill with 3 questions, add 2 compulsory misconceptions and 1 optional misconception to the skill and publish it. - Link the skill to the	1. Open the exploration editor page and add the skill created to the multiple choice card.	1. Buttons to tag response groups with misconceptions should be seen. List of unaddressed misconceptions should be seen with the misconceptions added to the skill.
			2. Open a response group and click on the tag misconception button.	2. Modal showing list of misconceptions that the response can be tagged to should be seen.
			3. Select the first misconception in the modal and press 'Done'.	3. The misconception should be visible in the 'tagged misconception' part of the response group.
			4. Close the response group and view the exploration editor page.	4. The first misconception should no longer be visible in the unaddressed misconceptions list.
			5. Click on the response group that was tagged to the first misconception	5. The first misconception should be seen tagged to it alongside an edit button.

		subtopic and add it as a diagnostic test skill. - Publish the topic. - Create a story with a chapter using the ID of the exploration created previously, and publish changes. - Go to the classroom admin page, create a classroom. - Add the topic created above to the classroom. This setup has created a curated exploration with the curriculum admin logged in.	6. Click on the edit button to edit the misconception tagged.	6. A list of misconceptions should be seen that the response group can be tagged to.
			7. Click on the second compulsory misconception and click on 'Save Misconception'.	7. The second compulsory misconception should be seen as the one that is tagged to the response group.
			8. Go to the misconceptions list below the answer groups.	8. The optional misconception should be seen in a yellow background with an alert depicting it needs to be addressed.
			9. Click on the 3 dots and select 'Mark as not applicable'.	9. The misconception should now be seen in a white background without any alert.
			10. Remove the linked skill from the state.	10. The response group should no longer have the aforementioned misconception tagged to it, and the misconception should not be visible in any list.

Implementation Plan

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

Milestone 1

Key objective for this milestone:

- Allow creators to see a list of existing translations for a particular part of a lesson when editing it.
- Allow creators to edit translations of particular parts of an exploration directly if the edits are easy to make.

Note: All functionality (as well as acceptance tests) will work in both desktop and mobile viewports.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	Add 'exploration_editor_can_modify_translations' feature flag for allowing translations to be updated when editing. Add 'exploration_editor_can_tag_misconceptions'	-	May 28	May 30

	feature flag for allowing response groups to be tagged with misconceptions in the exploration editor.			
2	Create an API endpoint and frontend service functionality for fetching all entity translations	-	June 2	June 4
3	Create the modify translations modal and have it display the existing translations of the relevant card. (At this point, the only action available to the user would just be to save the changes, i.e. everything would be "marked as needs retranslation" by default.) Write acceptance tests for CUJs 1.X.1 where X denotes the index of the content types, and 1.6	1, 2	June 8	June 11
4	Enable user to edit translations from the modal and publish them Write acceptance tests for CUJs 1.X.2 - 1.X.5, and 1.1.6, where X denotes the index of the content types.	2, 3	June 14	June 17
5.	Enable users to view translation changes when publishing the exploration.	-	June 21	June 24
	Product Demo		June 25	
6.	Bug fixes if required		-	< Jul 10

Milestone 2

Key objective for this milestone:

- Enable lesson creators to tag response groups with misconceptions.
- Enable lesson creators to see any non-optional misconceptions that haven't been addressed.

Note: All functionality (as well as acceptance tests) will work in both desktop and mobile viewports.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	Write state migration job (release testing documentation) to add the "inapplicable_skill_misconception_ids" list variable to the state object.	-	July 16	July 19

2	Allow user to see the misconception list in the curated exploration editor page Write acceptance tests for CUJs 3.1 and 3.10	M1.1	July 21	July 23
3	Allow user to mark optional misconceptions as inapplicable Write acceptance tests for CUJs 3.8 - 3.9	1	July 25	July 28
4	Allow user to tag response groups with misconceptions and change tagged misconceptions Write acceptance tests for CUJs 2.1 - 2.2, and 3.2 - 3.7	2	Aug 4	Aug 7
	Product Demo		Aug 11	
5.	Bug fixes if required		-	< Aug 24

Future Work

***Note**: This section is mainly for reference (since items listed here won't be part of the GSoC project). Proposals will primarily be evaluated based on the implementation plan above.*

- Auto-generate translations for simple changes such as purely images, integer solutions or integer response groups.
- Add validation checks for unaddressed misconceptions after all curated explorations have been modified accordingly.
- For the most optimum functionality, the user might need to return to the modal and edit or discard the changes they've made as necessary, before proceeding to publish them.