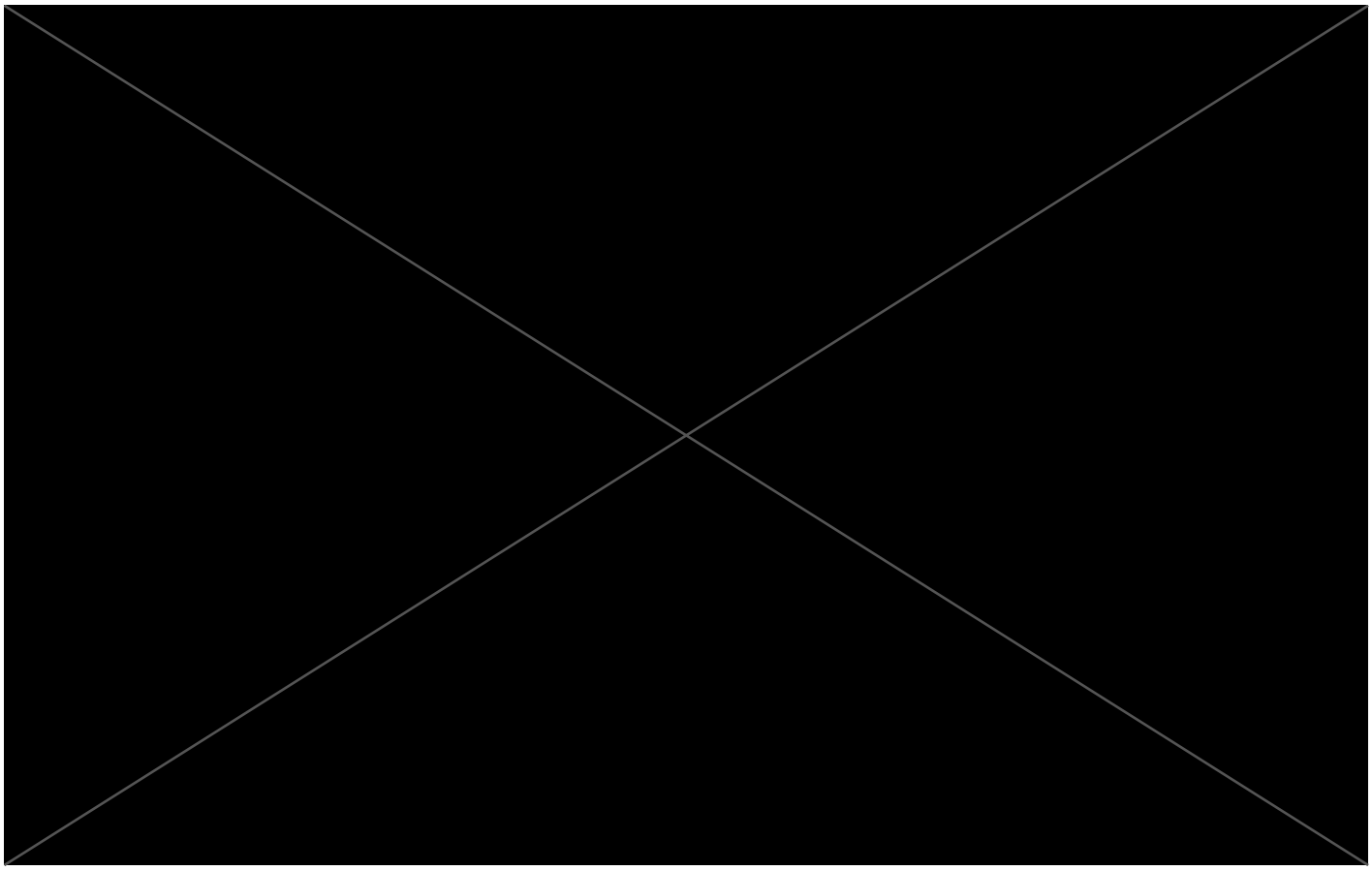


Section 2: About Your Project

Project Details

Project title	Serial Chapter Launch
Project size	Medium (~175 hours)
Why did you choose this project?	This project is an optimal combination of easy and challenging one for my web development skills. I also have experience working on issues in this section, and have explored the codebase of this section relatively thoroughly. Also, the duration of this project fits well with my other commitments during the timeframe.

Project Timeframe

Note*: Oppia will only be offering a single GSoC coding period timeframe this year, starting on **May 29**. All work for Milestone 1 must be completed and submitted by **July 14**, and all work for Milestone 2 must be completed and submitted by **Sept 15**. We will not be able to extend these deadlines.*

Coding period	● I will adhere to the above deadlines.
Planned time	From May 29 to July 17, I will be able to put 30 to 35 hours per week.

commitment	For the rest period, I will be able to put around 20 hours per week, due to college classes and examinations. Still, I will definitely try to make more time, if the project needs, during the program, and strictly prioritize the deadlines laid.
What other obligations might you need to work around during the summer?	I plan to devote much of my available time to the project during May 29 to July 17, due to my college vacations. After July 17, I will also have my college classes and mid-semester exams around a month later. I will also be having my internship exams and interviews around the same time roughly.

Communication Channels

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

I can commit to sending daily updates to my mentor by email, each day I work during the GSoC period.	• Yes
In addition to the above: how often, and through which channel(s), do you plan on communicating with your mentor?	I would prefer to communicate via email and Google Chat. Also I can discuss the progress, issues and workarounds with my mentors, via Google Meet, around two or three times per week.

Section 3: Proposal Details

Problem Statement

Link to PRD (or N/A if there isn't one)	Serial Chapter Launch PRD
Target Audience	1. Curriculum Admins, 2. Learners who have completed available chapters of a topic and looking forward to more launches, 3. Learners exploring newer content in Oppia

Core User Need	1. As a **Curriculum Admin,** I would like to publish chapters in a topic incrementally via a queue, so as to provide continuous learning, rather than waiting for preparing a complete story and publishing it. Also, I would like to inform the learners about the upcoming chapter releases in a topic. 2. As a **Learner,** I would like to know the upcoming chapters of a story that I have completed. Also, I can subscribe to these topics, to be notified of their new releases. I must also be able to identify newly published chapters, to explore them or continue learning the topics if I have attempted their earlier chapters before.
What goals do we want the solution to achieve?	1. Foster interest of learners in Oppia, by engaging the learners more, informing them about upcoming launches and speeding up the process of new content release. 2. To increase **Topic Breadth**, with different releases in different topics, allowing the learner to pick up new chapters of another topic. 3. To increase **Active Days with chapter completes,** if new chapters are available, a learner can learn those chapters, rather than waiting for the complete story release.

Section 3.1: WHAT

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

Key User Stories and Tasks

Note: All UI Mocks have been taken as given in PRD

#	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	Chapter Pipeline Management & Selective Publication	As a **Curriculum Admin**, **I need to** publish topics in installments **so that** Oppia can create more marketable moments related to topic launches	Must have	- A Chapter Control Panel in the **Story Editor Page**, that shows a queue of chapters of the story. - Click 'Add Chapter' to add a new chapter to the Chapter Control queue. - Create a Draft Chapter. - Click on a chapter in the Chapter Control or Edit option in Kebab Menu to add mandatory fields to make the chapter Ready-To-Publish. - Select a consecutive sequence of Ready-To-Publish	Chapter Control Panel Chapter Editor Page Unpublish warning modal

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

				chapters in the Chapter Control queue continuing from the last published chapter and click Publish	
				- Change the queue position of a chapter and edit it by selecting the option in Kebab Menu.	
2	Tracking Publication Progress	As a **Curriculum Admin**, **I need to** track the chapter publication schedule **so that** the Curriculum team can be ready for marketing launch announcements	Must have	- View story-wise chapter publication status inside "Canonical Stories" section of **Topic Editor Page**. - To view more chapter details, click on a story to land on **Story Editor Page** and view the publication details in Chapter Control Panel, like planned/published date, current status, etc.	Canonical Stories list in topic editor page
			Must have	- View overall chapter publication count and notifications for each topic row in **Topic and Skills Dashboard.** - Check more chapter publication details, by going to **Topic Editor Pag**e for a topic, and further clicking on a story to see details in Chapter Control Panel of **Story Editor Page**.	Topic and Skill Dashboard Stories Info
			Should Have	- Sorting the **Topic and Skills Dashboard** based on options like 'Most upcoming launches', 'Most behind schedule' or filtering based on 'Partially Published Topics'.	Topic and Skill Dashboard sorting and filtering
			Must Have	- Curriculum admin receives automated weekly email, notifying about the upcoming and pending launches.	Curriculum Admin email

				- Click the link in the email to visit the **Story Editor Page** and view the details in the Chapter Control Panel. - Visit the **Story Editor Page** through the application and view the details in the Chapter Control Panel.	
3.	Representation of Chapter Availability	**As a Learner**, **I need to** identify available chapters **so that** I'm aware how far I can progress in learning this topic currently and how much more content can I anticipate in the future.	Must Have	- View list of currently available chapters and upcoming chapters (grayed out) for each story in the **lessons tab of the topic**. - Attempt all available chapters. - Can subscribe to email notifications for future chapter launches to **this topic.**	Chapters List Display inside topic lessons tab Subscription confirmation modal Subscription requires user to be logged in message
			Must Have	- **End card of the last published chapter of a story** must show the message about completion of available content in the story, as well as, number of upcoming launches. - Can subscribe to email notifications for future chapter launches.	End Card of last available chapter
4	Representation of Recently Published Chapters	**As a Learner**, **I need to** continue learning the topic once new chapters are available **so that** I can fully learn the topic	Must Have	- In the 'Continue where you left off' section of **Learner Dashboard Page,** the stories with recent launches are visible with 'new chapters available' label. - Click on the card and start learning from the new published chapter.	Continue where you left off section
			Must Have	- Visit the **Classroom**	Classroom Page with newly

				Page and identify the topics with new launches via the 'New Chapters Available' label. - Click on any topic card to visit **Topic Page** and identify the new chapters with a 'new' label inside 'Available Chapters'. - Click on the chapter and start learning.	released chapters
			Must Have	- Inside **Topic Page,** identify newly launched chapters by a 'new' label inside the 'Available Chapters' section. - Click on the chapter and start learning.	Identify newly launched chapters in the topic lessons tab of the topic
			Must Have	- Receive email notifications if the learner has subscribed. View the newly published chapters for each topic. - Click on 'Click to view Topic' to land on T**opic Page**. - Click on a chapter to start learning.	Learner email
5	Keeping the Learner engaged	**As a Learner**, **I need to** continue learning even after exhausting a partially available topic **so that** I can continue having a productive learning session	Must Have	- At the end card of the last available chapter of a story, view suggestions for further activities. - Click on buttons to navigate to Classroom Page or Revision Cards.	End Card of last available chapter

Technical Requirements

Additions/Changes to Web Server Endpoint Contracts

#	Endpoint URL	Request	New / Existing	Description of the request/response contract (and, if applicable, how it's different from the previous one)

		type (GET, POST, etc.)		
1.	/story_editor_handler/data/<story-id> (frontend) (backend handler)	PUT	Existing	Request still contains 3 parameters • version • commit_message(optional) • change_dicts[] Just the scope of usage of this endpoint is further extended to **also saving the chapter's publication status (Draft/Ready To Publish/Published)**.
2.	/topic_subscription_handler/	POST	New	Request contains topic_id and user_id of the learner who subscribes to a topic. Response returns status true if successful, else error.
3.	/topic_editor_story_handler/<topic_id> (frontend) (backend handler) (UI served)	GET	Existing	Request contains topic_id. Response now contains story summaries with additional parameters, • **published_chapters_count** (integer) • **total_chapters_count** (integer) • **upcoming_chapters_count** (integer) • **overdue_chapters_count** (integer)
4.	/topics_and_skills_dashboard/data (frontend) (backend handler) (UI Served)	GET	Existing	Response contains data of Topics and Skills Dashboard, now including following parameters for each topic • **upcoming_chapters_count** • **overdue_chapters_count** • **published_stories_chapter_counts** (array of stories objects, each containing total_chapters_count and published_chapters_count for that story)
5.	/classroom_data_handler/<classroom_url_fragment> (frontend) (backend handler) (UI Served)	GET	Existing	Response must contain an extra boolean parameter "**newly_published_chapters_exist**", to mark a topic in Classroom Page with "New Chapters Available" label.
6.	/topic_data_handler/<classroom_url_fragment>/<topic_url_fragment> (frontend) (backend handler) (UI Served)	GET	Existing	Request contains classroom url fragment, topic url fragment and topic name. Response contains • topic object, with its "canonical_story_dicts" containing "**Published" and "Ready To Publish" story nodes** and **visited_node_titles** • **user_is_subscribed** (boolean) • **topic_has_upcoming_chapters** (boolean) • **unpublishing_notification_message**(string/None)
7.	/story_expectations_d	GET	New	Request contains **topic_url_fragment** and

	ata_handler/ (UI served)			**story_url_fragment** of current topic and story, when playing an exploration in story mode. Response contains 3 parameters ● **user_is_subscribed** (boolean) ● **upcoming_chapters_count** (integer) ● **topic_has_upcoming_chapters** (boolean)
8.	/cron/mail/curriculum_admins/chapter_publication_notifications/	GET	New	Request and response both do not contain any data.

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.	/story_editor_handler/data/<story-id> (frontend) (backend handler) (UI Served)	PUT	It will be used to save ANY changes in a story and its chapters. ● Presently, saving different fields(title, description, etc) changes in a StoryNode model (chapter) occurs using this endpoint (on clicking **Save Draft**). ● Now new fields will be added to the model for **status** (string)**, last modified, published date(**date-time type**), planned publication date(**date-time type) and **unpublishing reason** (string) So, we have added a new **status** field in the StoryNode model. This field change can thus be simply made using the same endpoint. ● On clicking **Save as Ready To Publish**, the change_dicts[] parameter in the call will now also contain the change of status to "Ready To Publish". ● On clicking **Publish,** the change_dicts[] parameter will contain all the selected chapters with their change of status to "Published". ● On clicking **Unpublish**, the change_dicts[] parameter will contain the change of status to "Draft", and the unpublishing_reason to the reason selected by the user as per the given table.
2.	/topic_subscription_handler/	POST	Called when the learner clicks on "Subscribe" to a topic. It adds that topic id to a new list called "topic_ids" in the UserSubscriptionsModel of that learner, that will contain IDs of topics the learner subscribes to.
3.	/topic_editor_story_handler/<topic_id> (frontend) (backend handler) (UI served)	GET	Called to collect story summaries to display in the **Story Control Panel** in Topic Editor Page. This endpoint now also adds these parameters for each story summary, namely, ● **published_chapters_count**: number of chapters published in that story ● **total_chapters_count**: number of chapters in the story

			• **upcoming_chapters_count:** number of chapters with planned publication date within 14 days. • **overdue_chapters_count:** number of chapters with planned publication date behind current date and status not Published
4.	/topics_and_skills_dashboard/data (frontend) (backend handler) (UI Served)	GET	Called to collect Topics and Skills Dashboard data. Now following will also be included in the response for each topic: • **upcoming_chapters_count** (to get the number of chapters planned to be released in the next 14 days) • **overdue_chapters_count** (to get the number of chapters running behind scheduled publication date) • **published_stories_chapter_counts** (so that the number of partially published and fully published stories of a topic, with the number of published and total chapters for each story, can be displayed)
5.	/classroom-data-handler/<classroom_url_fragment> (frontend) (backend handler) (UI Served)	GET	The boolean parameter, **newly_published_chapters_exist,** must also be added for each topic in "topic_summary_dicts" in response, to mark the topics with "New Chapters Available" in Classroom Page. Its value is set to true based on whether it was launched within the past 28 days and the learner has not yet visited the chapter.
6.	/topic_data_handler/<classroom_url_fragment>/<topic_url_fragment> (frontend) (backend handler) (UI Served)	GET	• We only need to display those chapters in the story tab of the topic viewer page that have status "Published" (under Available) or "Ready To Publish" (under Coming Soon). So each story in the response must contain only such story nodes. • Each story must also contain **visited_node_titles**, that can be used to check if a chapter is un-visited - for displaying "New" label on them. • **user_is_subscribed** is used to display Subscribe/Unsubscribe message on the button. • **topic_has_upcoming_chapters** is used to decide whether to display the Subscribe/Unsubscribe button or not. • **unpublishing_notification_message** is the message to be shown if there are unpublished chapters, which are not yet notified to the user. It is based on the unpublishing reason, as per the table given in PRD. If this parameter is None, then no notification will be shown.
7.	/story_expectations_data_handler/ (UI served)	GET	This endpoint is called, when the chapter exploration is loaded, to get the information if the chapter is the last chapter of the story, so that the end card can be rendered, with the recommended links, upcoming chapters (**upcoming_chapters_count)** and option to subscribe/unsubscribe (based on **user_is_subscribed is false/true**). If **topic_has_upcoming_chapters** is false, then no Subscribe/Unsubscribe button is shown.
8.	/cron/mail/curriculum_admins/chapter_publication_notifications/	GET	This endpoint is called weekly by the Google Cloud Scheduler, and it collects all the topic Ids and its chapters which are upcoming within 14 days or behind schedule, and sends their links in email to curriculum admins.

UI Screens/Components *(taken from PRD)*

#	ID	Description of new UI component	i18n required?	Mock/spec links	A11y requirements
1.	Chapter Control Panel	A queue of published and unpublished chapters, listed along with their status, publication date and last modified/published date.	Yes	Chapter Control Panel	Yes (see footnote below the table)
2.	Warning Message before unpublishing chapters	It gives a warning message to the curriculum admin, who is attempting to unpublish a chapter, based on the reason selected, and seeks confirmation for unpublishing.	Yes	Warning Message Modal	Yes (see footnote below the table)
3.	Modified Story Control Panel in Topic Editor page	Apart from story name and index, It also displays the story publication status, chapters published and publication notifications for each story.	Yes	Story Control Panel in Topic Editor Page	Yes (see footnote below the table)
4.	Modified Lessons tab of learner Topic page	Now, available chapters as well as upcoming unpublished chapters are also listed for each published story. Inside the available chapters section, recently launched chapters are labeled as “New”. Option to subscribe to the topic is present.	Yes	Topic Page lessons tab for learners Subscription Confirmation Modal	Yes (see footnote below the table)
5.	End Card	Option to subscribe to the topic is present. Also navigation links to other topics in the same classroom, and revision tab are present.	Yes	End Card	Yes (see footnote below the table)
6.	Continue Where you left off section	“Continue where you left off” Section is additionally populated with newly released chapter cards of the previously attempted topics.	Yes	Continue where you left off section	Yes (see footnote below the table)
7.	Topics and Skills Dashboard	Columns displaying number of added stories,Published Stories count and chapter publishing notifications are added.	Yes	Topic and Skills Dashboard extra columns	Yes (see footnote below the table)

In the a11y requirements, I will adhere to the suggested practices mentioned in the Accessibility wiki, that is maintaining proper aria tags in the components like buttons, icons, inputs etc, maintaining the alt attribute in the images, putting headings in correct format and adding proper text to the important icons.

Data Handling and Privacy

#	Type of data	Description	Why do we need to store this data?	Anonymized?	Can the user opt out?	Wipeout policy	Takeout policy
1.	Topic ids a learner subscribes to	The learner can subscribe to multiple topics for notifications upon future releases, so we need to store these topic ids.	This needs to be stored to keep track of which users to send email notification upon new chapter release in a topic.	No	The user can choose not to subscribe to a topic, then they will not receive any email notifications about the topic's future releases.	The topic_ids is a field of the UserSubscriptionsModel that follows the 'DELETE' deletion policy, so upon user account deletion the data will be removed, following the procedure described in wiki.	Like all other fields in the UserSubscriptionsModel, topic_ids will also follow a takeout policy 'EXPORTED'. The model has 'ONE_INSTANCE_PER_USER' association with the user.

Other Requirements

N/A

Section 3.2: HOW

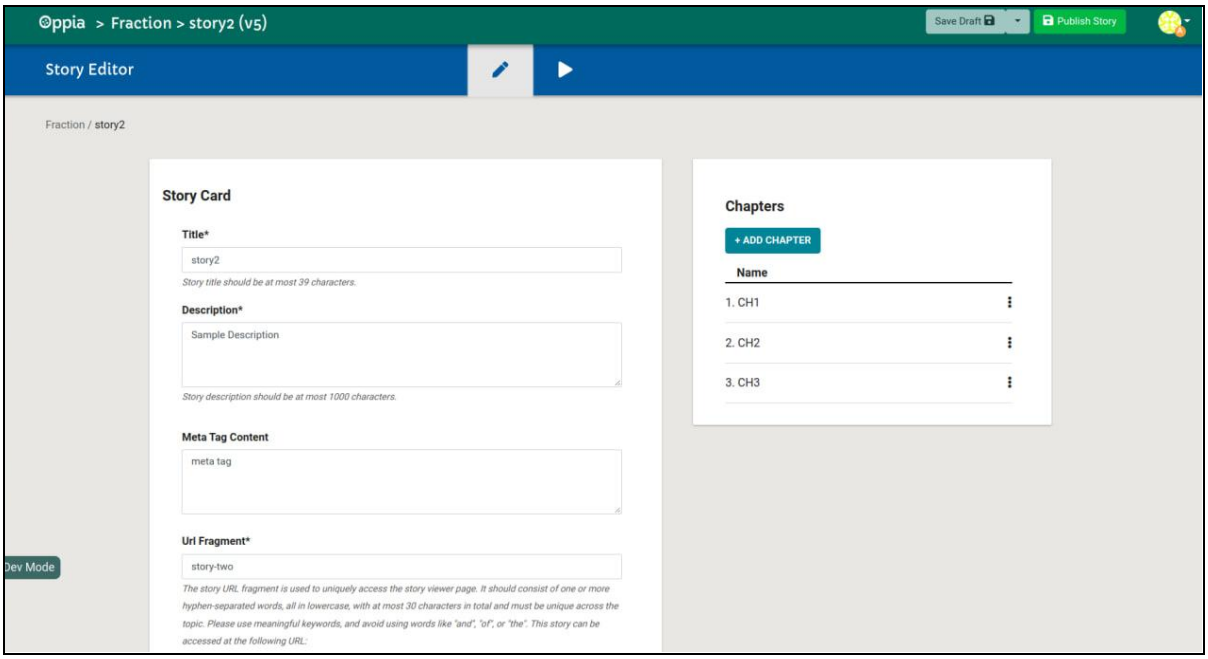
Existing Status Quo

Presently, there can be multiple stories inside a topic, and on publishing a story, all chapters inside it get published at once. Hence, all the chapters need to be in a finalized state so that a story can be published.
Now the major issues with this mechanism of creating stories are:

Team Facing

- Until and unless all the chapters are available for publishing, the story cannot be published. There is no provision of maintaining a queue of unpublished chapters, which can be published sequentially and incrementally with time, so that learners do not have to wait for the entire story to be available at the end.

Like in this story editor page below, on clicking “Publish Story”, all the 3 chapters will get published at once. The curriculum admin cannot shortlist and release only a few selected chapters, although, after publishing the story, new chapters can be further added to this list.



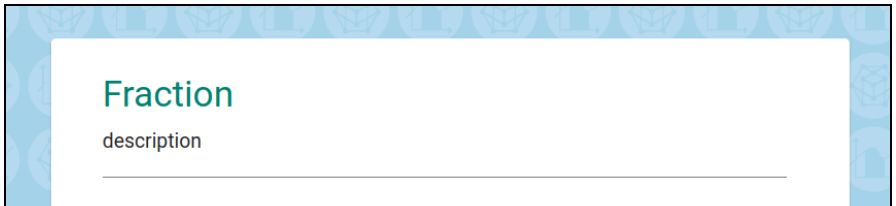
Now, this definitely does help in maintaining continuity for the learners while learning a topic, without dropping the learner’s interest. But, this also makes the learner wait for too long after finishing the available resources (almost 1 quarter for a new topic to be published) and hence they might lose interest.

User Facing

- The learners cannot be notified about the upcoming launches, so that they can anticipate the further content of the topic.
  Like in the topic card below, the learner can view the different published stories, and the chapters in them. But they cannot know what might be released in future, and therefore, as it takes a long time to make new releases, there runs a chance of learners losing interest in the topics that are being developed at the time.

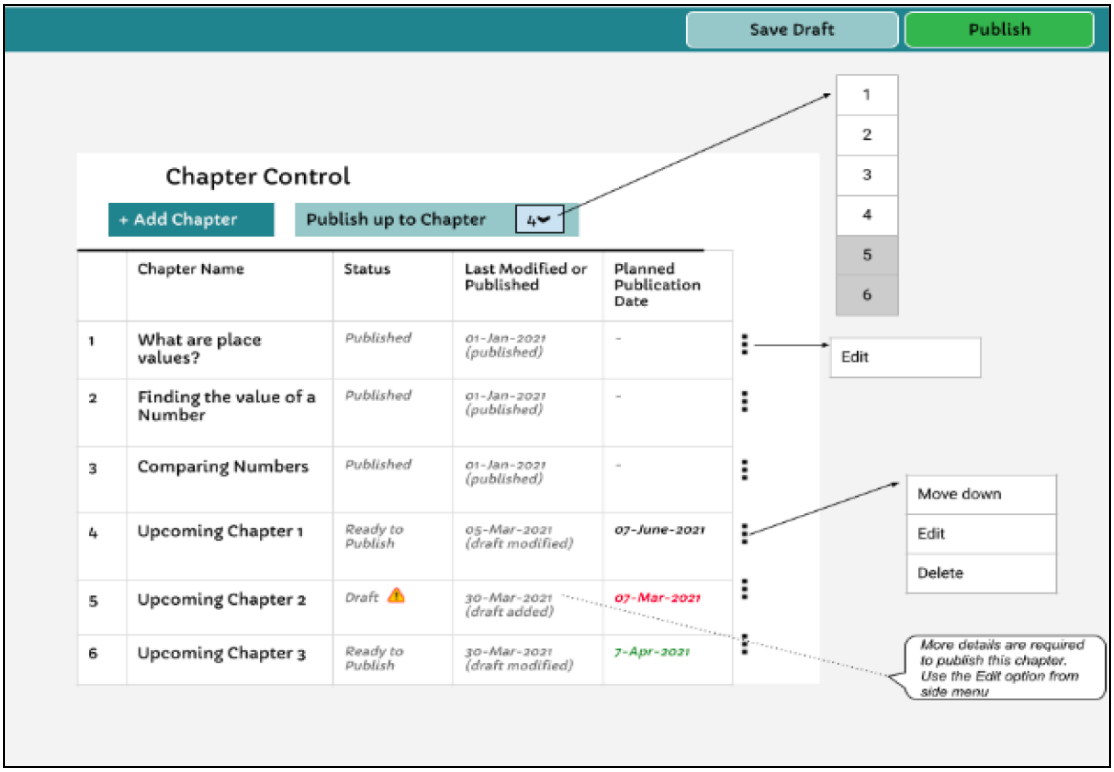
- Moreover, there is also no label to identify the newly released chapters in the topic, so the learner might overlook the recent releases.

- Also since there is no notification or subscription mechanism, they might never get to know about the new launches.



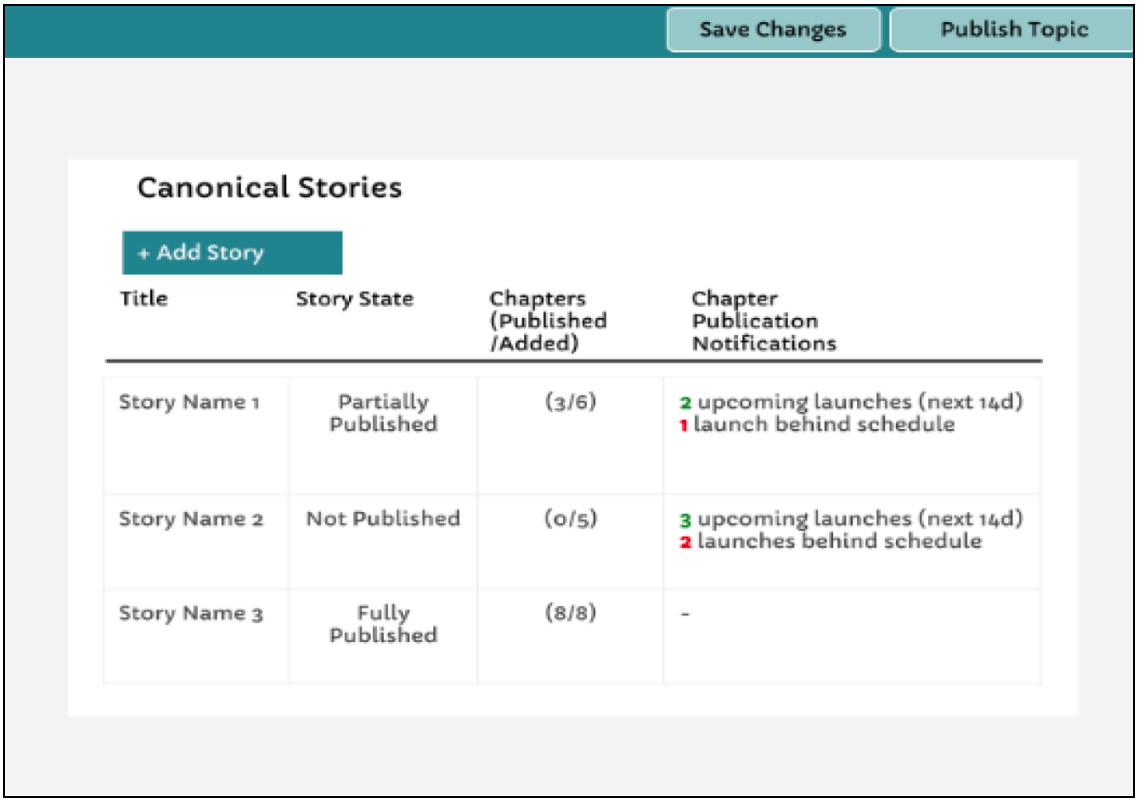
Solution Overview

- To address the main problem of publishing stories incrementally and sequentially, a queue of chapters will now be maintained, in which chapters will be categorized as **Published** or **Ready To Publish** or **Draft,** as shown in the following Chapter Control Panel



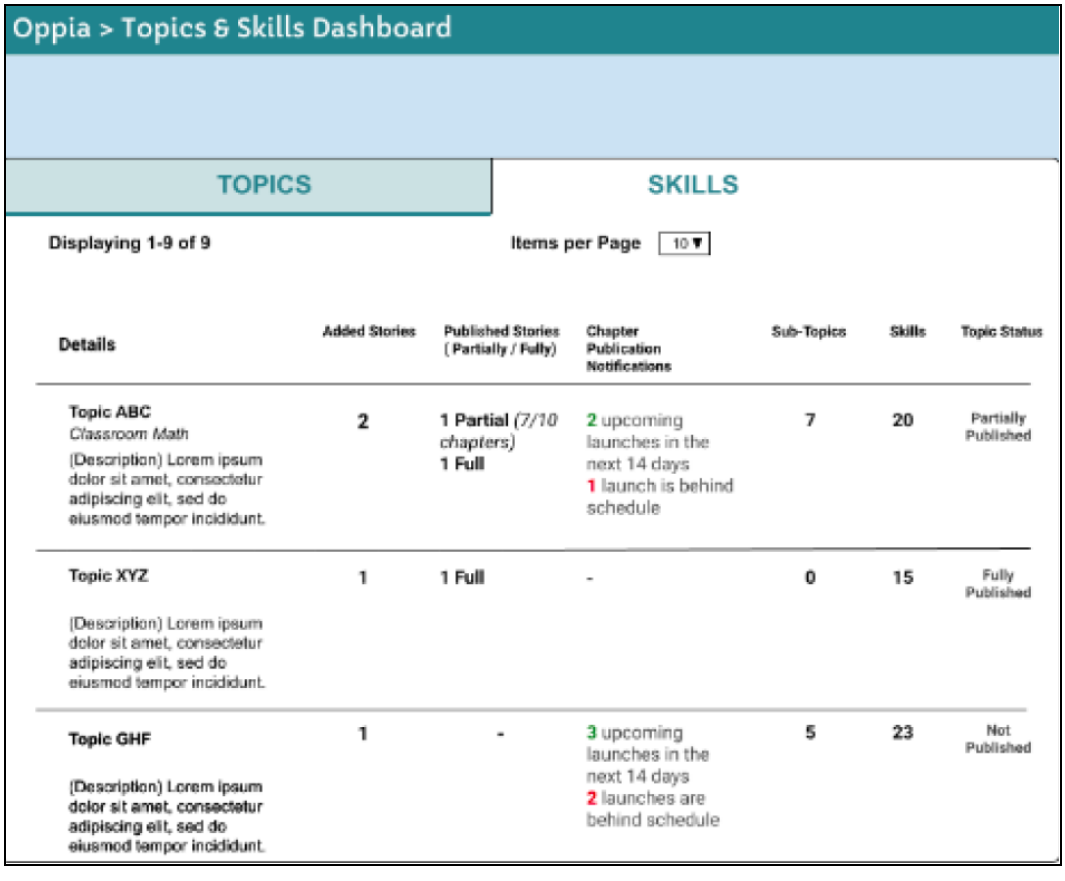
Now only the consecutive initial chapters with **Ready To Publish** status, after the last published chapter, can be selected for publishing. This solves the problem, as now chapters can be published incrementally and in sequence.

However, this introduces the need for effective chapter management for the curriculum admins, because now they will need to track the status of stories, not simply by published or non published, but instead by the **number of chapters published**, **ready to publish**, and also by their **upcoming release dates** and **behind schedule launches**.
For this, they must be able to see a brief story overview of its chapters, on opening the corresponding Topic Editor Page, as well as the Topics and Skills Dashboard.



Story Control Panel in Topic Editor Page

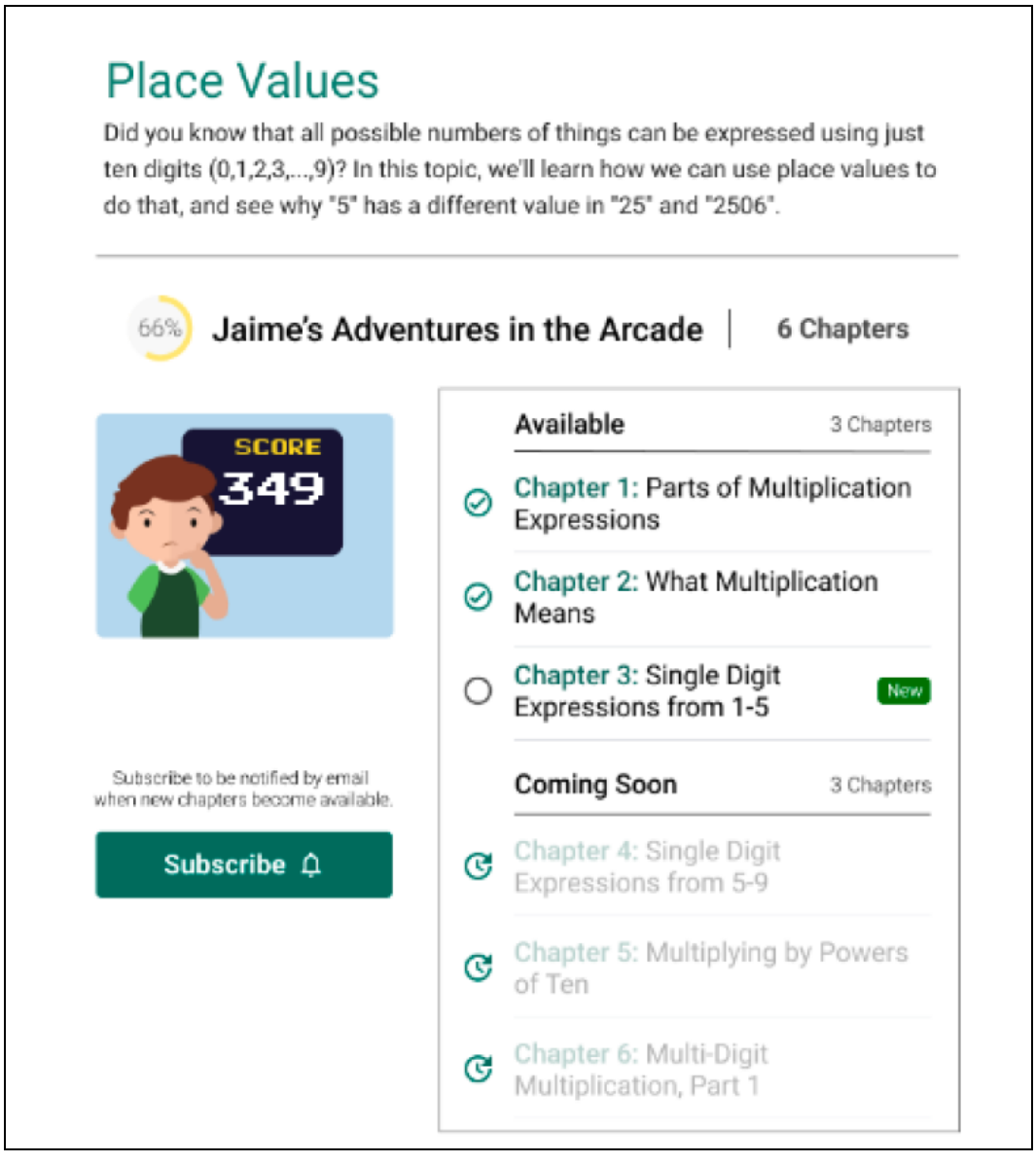
Topics and Skills Dashboard Page



They must also be notified via regular emails, reminding them about the upcoming and behind-schedule launches.

- For the learners, the topic viewer page, apart from containing the already published chapters, will now also contain the upcoming launches, but in disabled grayed-out state, that will consist of chapters categorized as **Ready to Publish.** Hence the learner can thus view the upcoming launches, and will not lose interest in the topic.

- As we can see in the UI below, the new chapters, which are unvisited by the learner, will have a “New” label attached, which helps them identify the new launches, and continue from where they last finished off, without overlooking the new chapters. Similar labels will also be visible on stories in the “Continue where you left off” section in Learner Dashboard Page, and on topic icons in Classroom Page. (See UI links here)



- A subscription feature is also added, that will notify the learner via email about new chapter releases in the topic.

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)
	N/A			

“Service” Dependencies

No.	Dependency name	Why it is needed	What our plan is, if the dependency fails under us
1	Mailgun	To send email notifications to curriculum admin and learners It already exists in the codebase.	If the mailgun service fails transiently, then the status code of the unsent mail is different from 200. Checking this, we can queue the unsent mail into the Google Cloud task queue for a later re-attempt and do this repetitively till the email is successfully sent (status code 200). This is similar to the UnsentFeedbackEmailHandler in tasks.py for re-sending unsent feedback related emails. Here is the link to the issue filed for it. Until the issue is resolved, if Mailgun fails to send an email (which would be rare), the unsent email would not get

[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.

			re-sent again, since there is no such error handling mechanism presently.

Impact on Other Oppia Teams

- The Curriculum Team will benefit by publishing the stories in parts, automating and simplifying the process of chapter status tracking and thus helping maintain the learner engagement on the platform.
- The Marketing team will make efforts to quarterly announce the recently published chapters and popularize this feature.

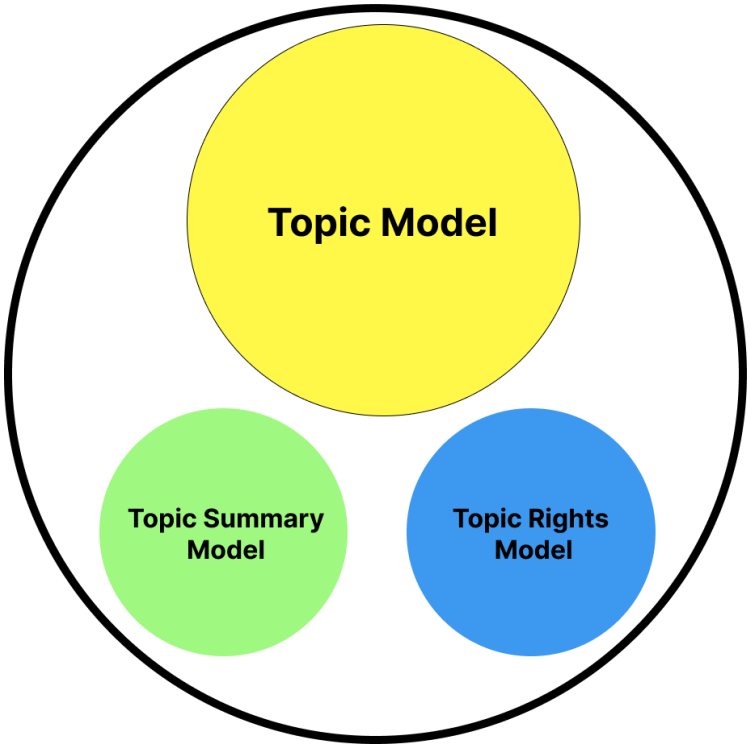
Key High-Level and Architectural Decisions

The core issue that we have to address in the project is to serialize the structure of chapters in a story so that they can be launched **in a sequence** and **in parts.**

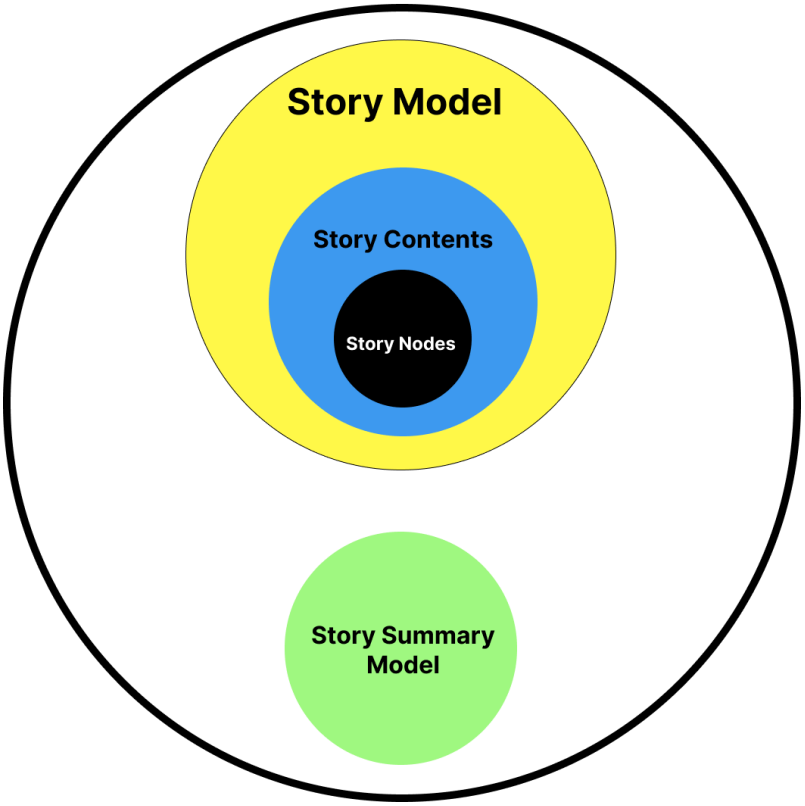
Before coming to the solution, I would like to highlight a key feature in the codebase organization.
If we analyze the application involving the Topic, Story and Chapter features, we can find that Topic is treated as one separate entity and Story is treated as another entity. What I mean by this can be shown through a Venn diagram below.

Throughout the codebase, we find that all the topic-related models, functions, services, components are grouped together in the same file/folder. Similarly all the Story and Story-related models, services and components are grouped together, wherever they are present in the codebase.

Topic and Topic-related model overview



Story and Story-related model overview



Obviously, these two entities are linked, following the hierarchy, (leftmost is the bottom-most in the hierarchy)

Chapter —> Story —> Topic

But a clear distinction between Story and Topic concepts is maintained throughout the codebase.
Contrastingly, as we see from the same diagram above, no such separation exists between Chapter (Story Nodes) and Story, as Chapter is treated as a part of Story only, not as a completely distinct entity. This can also be seen in the storage model structure, where Topic and Story have different storage models, but there is no separate storage model for Chapter (StoryNode), it is stored as a JSON property inside the field story_contents of StoryModel (see code).

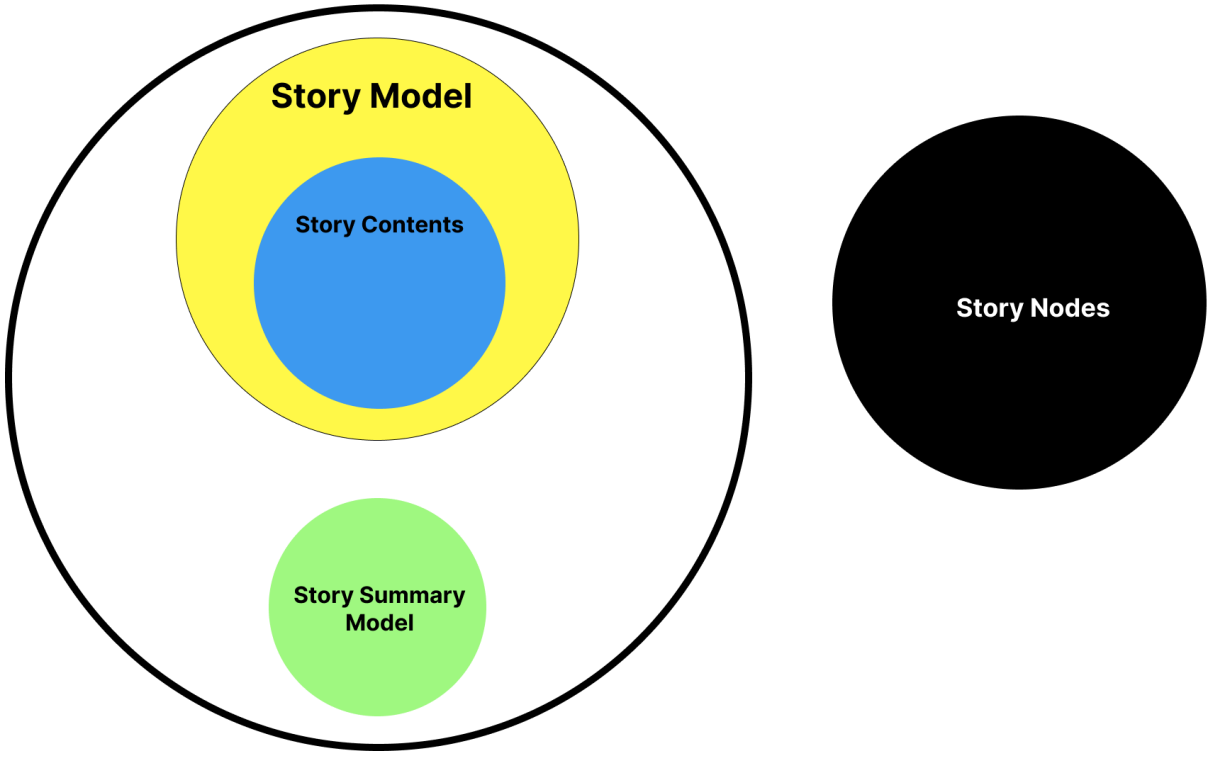
Now coming to the high level solution discussion, till now we have only published all the chapters in a story at once, that is we have maintained the same publication state of all the chapters (either all are published or all are unpublished). But the core need described in the project involves maintaining different chapters in different publication states. We are removing the binding that a story has over the chapters, that now all the chapters need not be published for a story to be published.

We can see that the management now goes one level down the hierarchy, from Story level to the Chapter level, in maintaining different states of entities.
The ideation can proceed via 2 alternatives.

Alternative 1

To implement the separate publication of chapters, we can evolve Chapter as a separate entity from the Story model, like shown below, similar to how Story and Topic are separate entities.

Story and Story-related model overview



Thus, a chapter will also have its **own separate storage and related models, its own different backend services and fetchers file, endpoints, etc**.

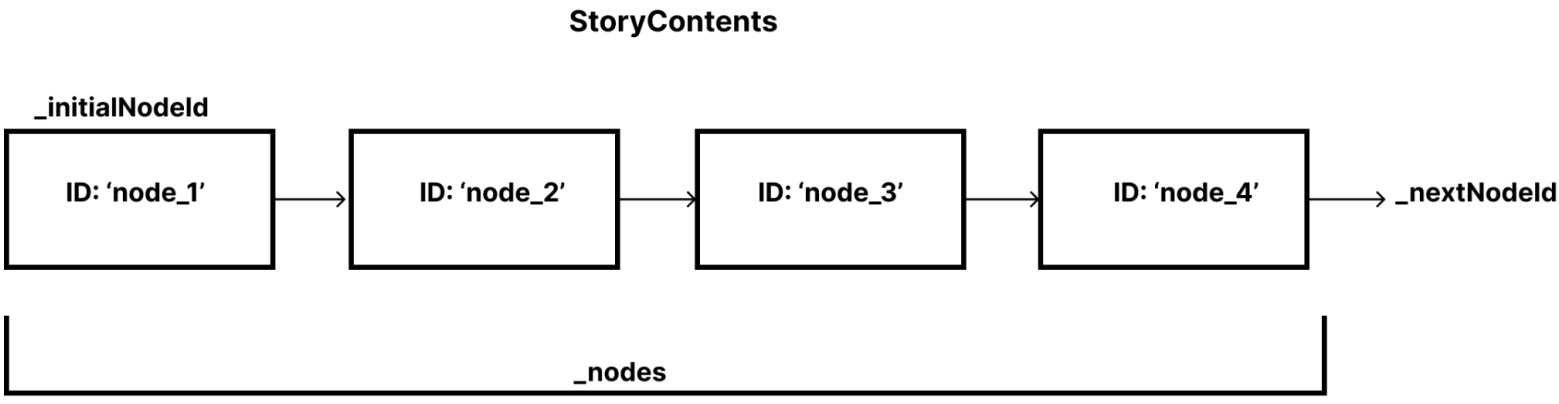
But this approach is filled with numerous disadvantages:

- Complete Refactoring of existing codebase, highly-error prone
  Most importantly, just this model architectural change will require a tremendous transformation in the existing codebase. A large number of different endpoints, functions and services that use the Topic, Story and Chapter related features have to be now rewritten in a refactored way for each functionality they serve, however small it may be, because **now the handling of chapters takes place in a separate manner via its own services functions and endpoints**. Even the distant features, which somehow are linked to this change, though are completely unrelated to the project have to be first rectified, which is not feasible as per the project duration, given the main solutions that have to be worked upon too. This vast refactoring will moreover leave a large room for errors and can in fact destabilize the present smoothly running application.

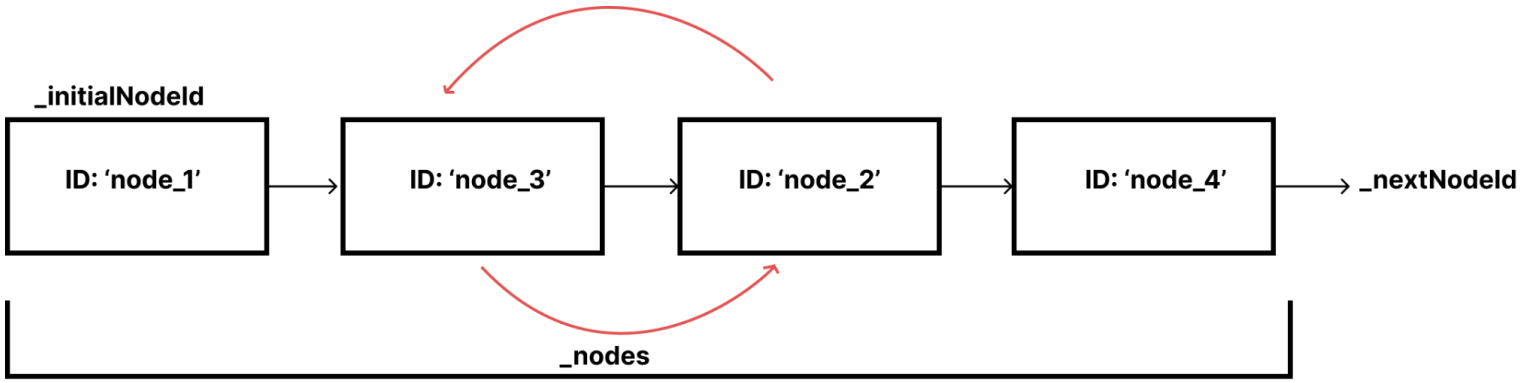
Alternative 2

The preferred alternative is to devise a solution which does not alter the above model architecture, but effectively handles the individual management of chapters.
In my proposed solution, we can introduce all of these features by adding just new fields to the existing models and writing additional functions and services to use these fields. I will also need to create some new models that will serve some “child features” in the project. But the core architecture and hierarchy of the Chapter and Story will be maintained intact, hence the disadvantages highlighted in the first approach have been eliminated.

Firstly, since chapters are no longer identical (they are classified into 3 categories); this can be done by just adding a **status** property to the StoryNode model, which will only be either of the 3 values (Draft/Ready To Publish/Publish). Publishing, unpublishing or saving as “Ready To Publish” can be performed just by modifying this status property of the chapter, like we modify any other chapter property (title, description, etc). Now, a story is marked as published when at least one of its chapters is published.
Secondly, the **pipeline structure is maintained via the StoryContents Model**.

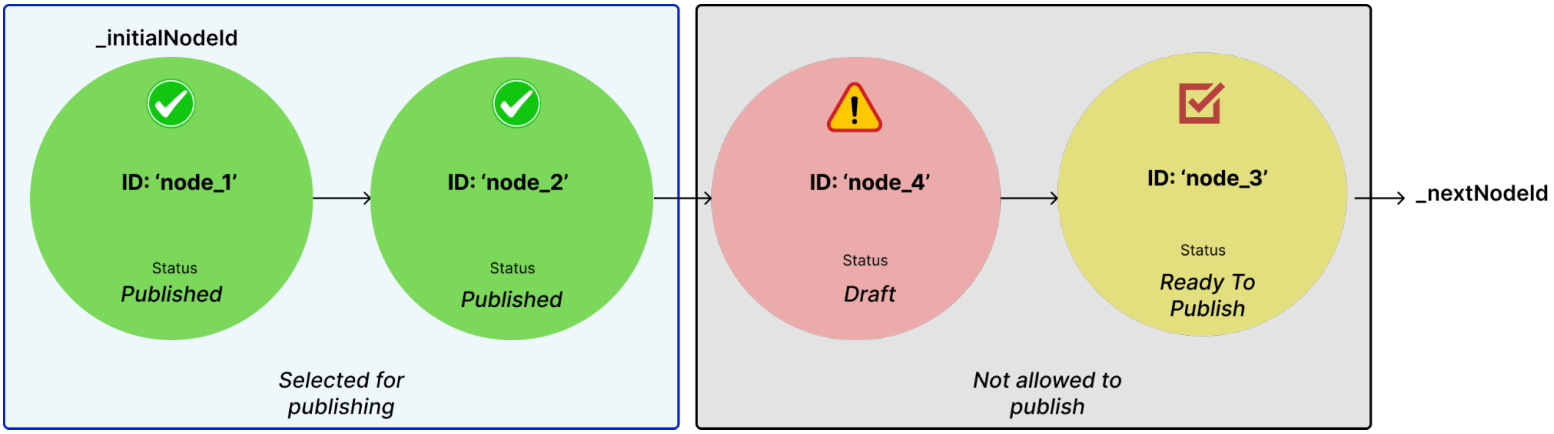


From the above structure, we can easily see that it is very similar to a linked list. Thus a queue of chapters is maintained in sequence. When we rearrange the queue, following happens:



The linked list structure maintains the new queue of chapters after rearrangement.

We can thus use this linked list type pipeline structure to publish the first selected chapters in sequence. Diagrammatically, this can be represented as:

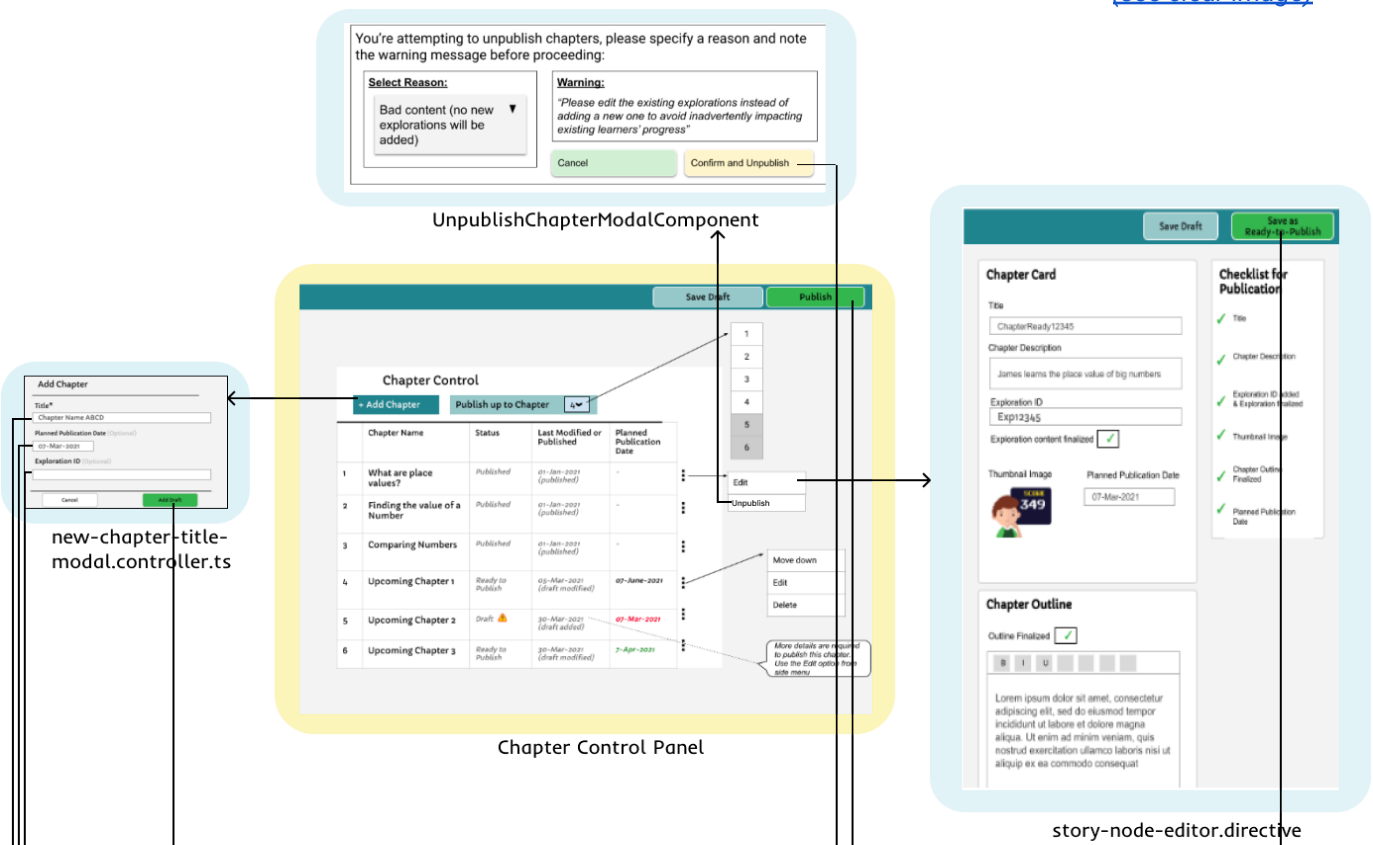


Decision 1:

We will proceed with the 2nd alternative, because it causes much less refactoring in the existing functionalities since only new fields to the existing models are added and this is simple to implement too.

A high level flowchart of how the chapters are maintained serially and the various operations on the chapter control panel that take place for serially launching a chapter, is described below. Further intricate details are mentioned in the Implementation Approach section.

[(see clear image)]



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

	Alternative 1: Treat Chapter as a separate entity from Story.	**Alternative 2:** To keep the current Story and Chapter model architecture and add new fields to existing models
Complexity and Efforts	Very complex We will need to identify all the functions, services, endpoints that involve the Chapter and Story relation, and first rewrite them as per the new model structure.	Less complex We do not need to refactor existing functions, services and endpoints (except at very few places), and the main features of the project can be worked upon.
Errors	Highly prone to errors. Any distant feature, that is using the Story and Chapter relation may get missed out, hence resulting in a broken existing feature instead.	Not prone to errors, as existing features, unrelated to the project are not touched.

Cumulative changes in model structures

The detailed explanation behind each of the changes in a model is covered in the implementation approach section.
The link to each part in the document, where the field is described and used, is given in the Sl No. column.

Frontend Models

StoryNode (and also StoryNodeBackendDict)

Sl No	**Field Name**	**Type**	**Description**
1.	_status	string	Choice of strings - Draft/ Ready To Publish/ Published - used to store the chapter's publication status. It is "Draft" for a newly created node.
2.	_plannedPublicationDateMsecs	number	Stores planned publication date of chapter. It is null for an unpublished and a newly created node
3.	_lastModifiedMsecs	number	Stores when the chapter was last modified.

4.	_firstPublicationDateMsecs	number	Stores when the chapter was first published. It is null when chapter status is not “Published”
5.	_unpublishingReason	string	Stores the reason for unpublishing a chapter. It can take values - “BAD_CONTENT” or “CHAPTER_NEEDS_SPLITTING”, as per the table. When a chapter is not in unpublished state, this field is null.

StorySummary (and also StorySummaryBackendDict)

Sl No	Field Name	Type	Description
1.	_publishedChaptersCount	number	Count of total published chapters in the story
2.	_totalChaptersCount	number	Count of total chapters added in the story
3.	_upcomingChaptersCount	number	Count of chapters with planned publication date within 14 days
4.	_overdueChaptersCount	number	Count of chapters with planned publication date behind the current date.
5.	_visitedChapterTitles	string[]	Titles of chapters that have been once opened by a learner.

CreatorTopicSummmary (and CreatorTopicSummmaryBackendDict)

Sl No	Field Name	Type	Description
1	totalUpcomingChapters	number	Total count of upcoming launches in the topic.
2.	totalOverdueChapters	number	Total count of chapter launches behind schedule in the topic.
3.	publishedStoriesChaptersCount	Object list	Each object element in the list represents a story of that topic and has 2 fields” ● Count of published chapters in the story ● Count of total added chapters in the story
4.		boolea	Determines if the topic has newly published

	newlyPublishedChapterExist	n	chapters.

Backend Models

StoryNode (and also StoryNodeDict)

Sl No	Field Name	Type	Description
1.	status	string	Choice between strings - Draft/ Ready To Publish/ Published - used to store the chapter's publication status. It is "Draft" for a newly created node.
2.	planned_publication_date	datetime.datetime	Stores planned publication date of chapter. It is None for an unpublished and a newly created node.
3.	last_modified	datetime.datetime	Stores when the chapter was last modified.
4.	first_publication_date	datetime.datetime	Stores when the chapter was published first. It is None when the chapter status is not "Published". Also it does not indicate that the chapter is currently in published state, it could have been published earlier and then unpublished later.
5.	unpublishing_reason	string	Stores the reason for unpublishing a chapter. It can take values - "BAD_CONTENT" or "CHAPTER_NEEDS_SPLITTING", as per the table. When a chapter is not unpublished, it is None.

Storage Models

UserSubscriptionsModel

Sl No	Field Name	Type	Description
1.	topic_ids	List of	Stores the topic ids to which a learner is subscribed,

		strings	notifying them when new chapters are released

StoryProgressModel

Sl No	Field Name	Type	Description
1.	visited_node_ids	List of strings	IDs of the nodes in the story that have been visited once by the user.
2.	unpublished_and_notified_node_ids	List of strings	IDs of those nodes in all of the topic's stories, which have been unpublished, and the unpublished notification has been shown to the learner.

Risks and mitigations

Potential Risk	Mitigation
Longer gaps between new chapter releases might cause learners to lose interest.	• Notify the learners via intermediate emails with teasers regarding upcoming launches. Also new content releases must be notified to the subscribed users via email. • Visiting frequency of learners can be analyzed to determine time difference between consecutive launches.
Complication of translation process, due to increased communication workload and errors between curriculum and translations teams, and increased time delays, for translated contents to be available to non-English users.	A notification feature can be developed to share the chapters queue structure to the Translations Team.

Implementation Approach

In this section, I have separately elaborated the complete list of changes I will be making in each of the 5 aspects, namely **Storage Models, Domain Objects, User Flows(Controllers and Services), Web Frontend Changes and Documentation**, for each feature of the project.

1. Chapter Control Panel Feature in Story Editor Page (UI Mock)

To implement this feature, first we need to have new fields in the chapter (StoryNode) models.

Storage Model Layer Changes

Chapters are stored in the story_contents field of the StoryModel in json format.
This would require back-populating the new fields for the existing stories and their chapters.

- **status**: For the published stories, all chapters will have status “Published”, and for non-published stories, the chapters will have status as “Draft”.
- **unpublishing_reason_id**: It will be null for all chapters since no chapter has been unpublished till now.
- **publication_date**: Only for the chapters belonging to published stories (for other chapters, it will be null). It will be set to the later of the two dates from the commit history of the StoryModel, either when the story was published or when the chapter appeared in the story for the first time.
- **last_modified**: It can be set to the date when the chapter was last changed, based on the commit history of its Story Model.
- **planned_publication_date**: For chapters of the published stories, it will be set to the publication_date, else for the remaining chapters, it will be None.

Domain Objects

Backend Domain Objects Changes

story_domain.py:

Changes to existing domain model - **StoryNode** (and **StoryNodeDict**) - with new fields:

- status (string): choice - Draft/Ready To Publish/Published
- planned_publication_date (Date-Time): planned date of chapter publication
- last_modified (Date-Time): date/time when the chapter was last modified
- first_publication_date (date-time): date when the chapter was first published
- unpublishing_reason (string): Reason as per the table (PRD), selected by the curriculum admin to unpublish the chapter.

Existing StoryNode backend domain model in story_domain.py

```
class StoryNode:
    """Domain object describing a node in the exploration graph of a
    story.
    """

    def __init__(
        self,
        node_id: str,
        title: str,
        description: str,
        thumbnail_filename: Optional[str]
```

So, when we edit these new properties, we also need to modify the existing class - **StoryChange** - with the following in STORY_NODE_PROPERTIES list:

- STORY_NODE_PROPERTY_STATUS
- STORY_NODE_PROPERTY_PLANNED_PUBLICATION_DATE
- STORY_NODE_PROPERTY_LAST_MODIFIED
- STORY_NODE_PROPERTY_FIRST_PUBLICATION_DATE
- STORY_NODE_PROPERTY_UNPUBLISHING_REASON

We also have to add these new classes corresponding to the above properties:

- UpdateStoryNodePropertyStatusDict
- UpdateStoryNodePropertyPlannedPublicationDateDict
- UpdateStoryNodePropertyLastModifiedDict
- UpdateStoryNodePropertyFirstPublicationDate
- UpdateStoryNodePropertyUnpublishingReasonDict

Frontend Domain Objects Changes

story-node.model.ts:

- Similar changes to existing model - **StoryNode** (and **StoryNodeBackendDict**) - with new fields:
  - _status
  - _plannedPublicationDateMsecs
  - _lastModifiedMsecs
  - _firstPublicationDateMsecs
  - _unpublishingReason

Existing StoryNode frontend model in story-node.model.ts

```
export class StoryNode {
  _id: string;
  _title: string;
  _description: string;
  _destinationNodeIds: string[];
  _prerequisiteSkillIds: string[];
  _acquiredSkillIds: string[];
  _outline: string;
  _outlineIsFinalized: boolean;
  _explorationId: string | null;
  _thumbnailBgColor: string | null;
  _thumbnailFilename: string | null;
```

User Flows (Controllers and Services)

There are 3 major operations that can be performed in the chapter control panel table (except Add Chapter, which is implemented as the next feature)

story-editor.directive.ts

Kebab Menu Operations (Move Up/Move Down)

We will need 2 new functions to move a chapter up or down in chapter sequence(apart from drag and drop):

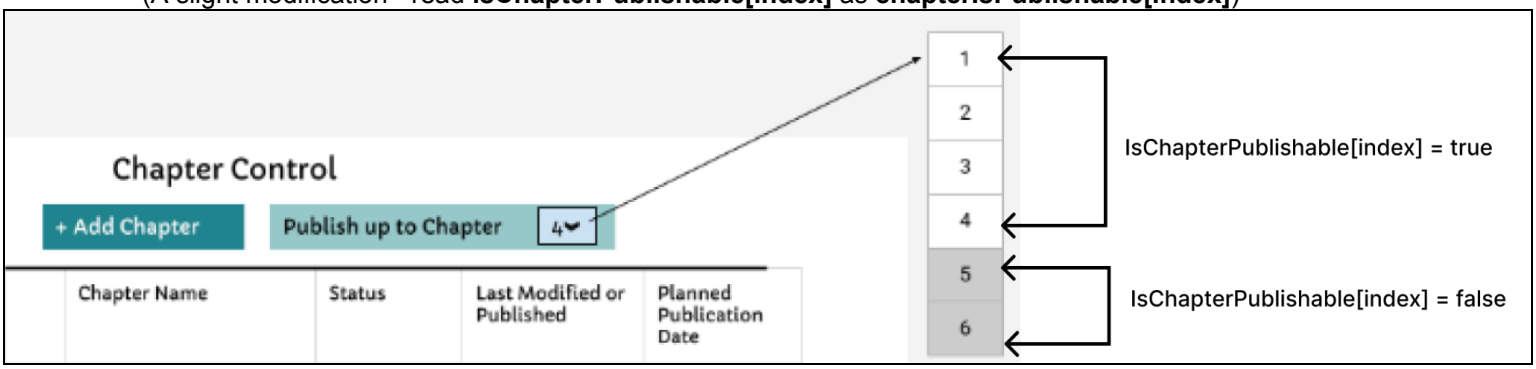
- onMoveChapterUp(index, nodeId): calls the existing function rearrangeNodeInStory(index-1)
- onMoveChapterDown(index, nodeId): calls rearrangeNodeInStory(index+1)

Publish Upto Chapter Operation

We need 2 new variables to handle the Publish Upto Chapter X dropdown:

- chapterIsPublishable[index] (boolean): It indicates if a chapter (given by linearNodesList[index]) is publishable. When false, that index will be disabled in the dropdown.
  It is true if the chapter has status "Published" or "Ready To Publish" and chapterIsPublishable[index-1] (if index is non zero) == true. Else it is false

(A slight modification - read **isChapterPublishable[index]** as **chapterIsPublishable[index]**)



- selectedChapterIndexToPublishUpToInDropdown (integer): The value selected in the Publish Upto Chapter dropdown - 1. If say option 4 is selected, then its value is 3, indicating index 3 of the array. Its default value is the index of the last published chapter, or if there is no published chapter, then it is 0.

We will also need a new function - getPublishableChaptersIndices() - to initialize the array chapterIsPublishable[] whenever the Chapter Control Panel is loaded or updated.

In this section, only the dropdown operation that takes place in the chapter control panel table is described. How the chapter is published is covered separately in the 5th feature.

Unpublish Chapter

For this, we will have a function - $scope.unpublishChapter(nodeId) - that opens a new modal **UnpublishChapterModalComponent** (using NgbModal.open()) on clicking unpublish option in the navbar (This option appears in the navbar if StoryEditorStateService.chaptersAreBeingPublished is false, if chapter index selected in the dropdown is lesser than the last published chapter index). Upon success callback, the properties of the chapter will get updated by calling the following functions (defined inside the next feature) of *story-update.service.ts*:

- setStoryNodeStatus() is called with "Draft" to mark the chapter as draft.
- setStoryNodePlannedPublicationDate() is called with null to remove the planned publication date.
- setStoryNodeUnpublishingReason() is called with the reason selected in unpublishing modal dropdown ("BAD_CONTENT" for the first option, "CHAPTER_NEEDS_SPLITTING" for the second option)

Note: The user will only be allowed to unpublish a suffix of the last published chapter in the story, to prevent any discontinuity in learning on unpublishing any in between published chapter. That is, users can select any item in the dropdown, and if it is less than the last published chapter number, then all the chapters after the selected chapter to the last published chapter will be unpublished.
Finally, call a new defined function - **saveChapter()** in the *story-editor-state.service.ts* that further calls saveStory() to unpublish and save the new status of the chapter, by making a put request to the EditableStoryDataHandler class (url **/story_editor_handler/data/<story_id>**).

Also, if all the chapters in the story are getting unpublished, we need to change the story from Published to Unpublished by calling the existing function changeStoryPublicationStatus() in *story-editor-state.service.ts.* This story will now be unavailable for learners, and will also not be seen in the topic viewer page.(mock)

In the *story_domain.py*, there will be an enum class - **ChapterUnpublishingReasons**.

```
class ChapterUnpublishingReasons(enum.Enum):
    """Enum for unpublishing reasons"""

    BAD_CONTENT = 'Bad Content'
    CHAPTER_NEEDS_SPLITTING = 'Chapter Needs Splitting'
```

tt

In the backend, the unpublishing_reason can be used to store the right Enum member value in the storage model.

When fetching the data during GET endpoint calls, the backend StoryNode model will contain the **enum name** (CHAPTER_NEEDS_SPLITTING/BAD_CONTENT) based on the **value** stored in the storage using - UnpublishingReasons(model.unpublishing_reason).name. This can be passed to the frontend.

Note: Unpublishing operation is changing the status property of each of the unselected published chapters in the dropdown to "Draft" (along with changing planned publication date, publication date and unpublishing reason) so like editing all other properties (title, description, etc) are saved using saveStory(), so this operation will also be done using saveStory(). In later sections, publishing and saving draft buttons will also use the same function only (due to similar reason).

Web frontend changes

These are the visible frontend (UI) changes corresponding to the operations listed above.

story-editor.directive.html:

Chapter Control Panel Table Display

The list of chapters displayed in the chapter control panel is given by linearNodesList[], which is an existing array of frontend model StoryNode instances.
Since, now the StoryNode model has the new properties, thus "Status"," Last Modified/ Published" and "Planned Publication Date" columns can also be filled in the table.

Kebab Menu Operations:

- If linearNodesList[index] has status != 'Published', display Edit, Move Up, Move Down, Delete options in the Kebab Menu. However for the first unpublished chapter, there will only be an option to move down, and for the last unpublished chapter, there will only be an option to move up.
- If linearNodesList[index] has status == "Published", only display Edit, Unpublish options.
- On clicking Move Up, call onMoveChapterUp(index,nodeId), defined in the ts file.
- On clicking Move Down, call onMoveChapterDown(index,nodeId), defined in the ts file.
- On clicking Delete, call deleteNode() (Existing function)

Publish Upto Chapter Dropdown:

- It displays a list of indices, with those grayed out that are not "Ready To Publish" in consecutive sequence with "Published" ones (that is chapterIsPublishable[index] is false).

- On selecting a value, it is stored in the variable selectedChapterIndexToPublishUpToInDropdown (defined in its ts file above)

Unpublish Chapter

- "Unpublish" option appears inside the navbar if the item selected in the Publish Upto Chapter dropdown is already published, that is all the published chapters after that selected item will be unpublished. On clicking, it opens a new modal component named **UnpublishChapterModalComponent** to select the reason for unpublishing. It also contains the list of consequences upon unpublishing the chapter as a warning. On confirming, the chapter is added to the last position in the queue.

Documentation changes

None

2. Add Chapter Feature (UI Mock)

Storage Model Layer Changes

None

Domain Objects

change.model.ts:

Inside StoryNodePropertyChange, we must have the following change interfaces corresponding to the new StoryNode fields.

- interface StoryNodeStatusChange
- interface StoryNodePlannedPublicationDateChange
- Interface StoryNodeLastModifiedDateTimeChange
- Interface StoryNodeFirstlPublicationDateChange
- Interface StoryNodeUnpublishingReasonChange

See the image below for better clarity.

User Flows (Controllers and Services)

story-update.service.ts:

To update the new StoryNode fields and register them as applied changes, following new functions will be used in the class StoryUpdateService.

- setStoryNodeStatus()**:** To update the chapter's status (Draft/Ready To Publish/Published)

- setStoryNodePlannedPublicationDate(): To update the chapter's planned publication date
- setStoryNodeLastModifiedDateTime(): To update the chapter's last modified date/time
- setStoryNodeFirstPublicationDate(): To update the chapter's first publication date
- setStoryNodeUnpublishingReason(): To update the reason for unpublishing a chapter

Note: the function setStoryNodeLastModifiedDateTime() is called inside every story node property changing function (including those that already exist in the codebase).

Existing StoryNodePropertyChange and some of the interfaces

```
interface StoryNodePrerequisiteSkillsChange {
  'cmd': 'update_story_node_property';
  'property_name': 'prerequisite_skill_ids';
  'new_value': string[];
  'old_value': string[];
  'node_id': string;
}

interface StoryNodeAcequiredSkillsChange {
  'cmd': 'update_story_node_property';
  'property_name': 'acquired_skill_ids';
  'new_value': string[];
  'old_value': string[];
  'node_id': string;
}

type StoryNodePropertyChange = (
  StoryNodeOutlineChange |
  StoryNodeTitleChange |
  StoryNodeDescriptionChange |
  StoryNodeThumbnailFilenameChange |
  StoryNodeThumbnailBgColorChange |
  StoryNodeExplorationIdChange |
  StoryNodeDestinationIdsChange |
  StoryNodePrerequisiteSkillsChange |
  StoryNodeAcequiredSkillsChange);
```

new-chapter-title-modal.controller.ts:

Now the chapter creation modal consists of 3 fields - Title, Thumbnail Image and Exploration ID. We already have the $scope.updateTitle(), $scope.updateThumbnailFilename(), $scope.updateThumbnailBgColor() and $scope.updateExplorationId() functions to set the title, thumbnail and exploration ID respectively.

For setting the initial chapter status to "Draft", we will have the following update function.

- $scope.updateChapterStatus(): To update chapter status to "Draft", when chapter is first created

It will call its corresponding setter function in *story-update.service.ts* just defined above.

Web frontend changes

new-chapter-title-modal.template.html:

The modal now contains fields to enter Chapter Title and Exploration ID (Optional), apart from the existing thumbnail image field. That is, the only frontend change made in the modal is making the exploration ID field optional to fill during chapter creation.

Documentation changes

None

3. Chapter Editor Page (UI Mock)

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

story-node-editor.directive.ts:

The chapter editor page now accepts one more field, Planned Publication Date, initially an empty string, so we need to have a function to update this property of the chapter as well.

- $scope.updatePlannedPublicationDate(): To update the planned publication date field. It calls the setStoryNodePlannedPublicationDate() function of the *story-update.service.ts,* defined above.

To implement the checklist feature, a variable is maintained,for each of the checklist points, that gets updated on updating that field.

Web frontend changes

story-node-editor.directive.html:

- New form UI as described in the mock above, with an added field - Planned Publication Date.

- New checklist points, as specified in the PRD, have to be included in the checklist div, and each of the requirements is checked, when the corresponding variable of the field is non-empty.

Documentation changes

None

4. Save as “Ready To Publish” Feature

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

story-editor-state.service.ts:

We need to enable the Save as Ready To Publish button only if the checklist in the chapter editor page is completely checked, as described above. To transfer this information of checklist from *story-node-editor.directive.ts* to *story-editor-navbar.component.ts*, we need to declare a boolean variable - currentNodeIsPublishable - inside StoryEditorStateService, which is initially false, and also define its setter(setCurrentNodeAsPublishable()) and getter(isCurrentNodePublishable()) functions.

story-node-editor.directive.ts:

Declare a flag variable, to check if all the checklist variables (defined above) are fulfilled, and whenever the flag is updated (which happens when any of those checklist variables are changed), the value of StoryEditorStateService.currentNodeIsPublishable is set to the value of flag, by calling setCurrentNodeAsPublishable(flag).
Thus, when all the checklists variables are non empty, the flag is set to true, therefore currentNodeIsPublishable is also set to true.

story-editor-navbar-component.ts:

To activate the “Save as Ready To Publish” button in navbar component, if

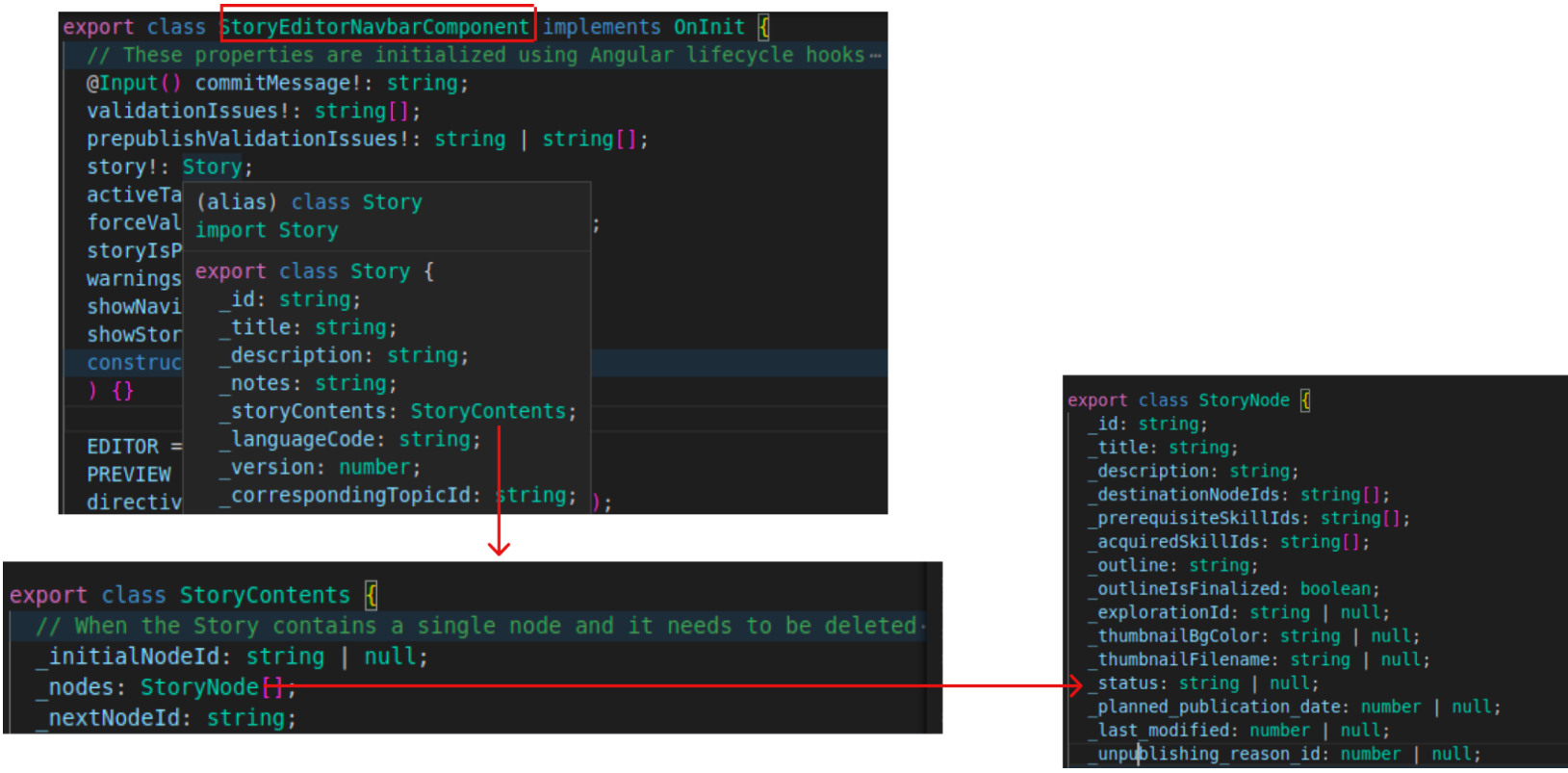
- StoryEditorStateService.isCurrentNodePublishable() is true and
- Chapter status is not already “Published” or “Ready To Publish” (get chapterId using StoryEditorNavigationService.getChapterId() and find its chapter (StoryNode instance) in

the Story instance (currently exists inside the navbar component), then get its status property), see flowchart below for clarity,

- All changes are committed, that is, it is clickable only after Save Draft. (UndoRedoService.getChangeCount() == 0)

a new variable - readyToPublishIsEnabled **-** is set to true.

How to retrieve chapter status in the *story-editor-navbar.component.ts*



To save as "Ready To Publish", as described in a note above, we just need to change the status property of StoryNode to “Ready To Publish”.

So, create a new function in this file *(story-editor-navbar-component.ts)* - changeChapterStatus() - which changes the status of the chapters. In this case, it calls the setStoryNodeStatus() of *story-update.service.ts* with "Ready To Publish", then calls saveChapter() of *story-editor-state.service.ts* that calls saveStory() to save the change, by making a put request to the EditableStoryDataHandler class via url **/story_editor_handler/data/<story_id>.**
See pseudocode of this function below.

Web frontend changes

story-editor-navbar.component.html:

- A new button - Save as “Ready To Publish” - is added, and it is rendered if StoryEditorNavigationService.getActiveTab() == ‘chapter_editor’ and the chapter status is Draft. The different buttons displayed in case of different status chapters in the chapter editor page are shown in this image.
- The button is disabled if the variable readyToPublishIsEnabled (described above in its ts file) is false.
- On click, call changeChapterStatus() in its ts file with the arguments “Ready To Publish” and chapter index.

Documentation changes

None

5. Publish Chapter Feature

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

story-editor-state.service.ts:

We see that the Publish button utility is defined in the *story-editor-navbar.component.ts*, but the selection of chapters to publish happens inside the *story-editor.directive.ts*, so we need to send the value of selectedChapterIndexToPublishUpToInDropdown(defined above) to the navbar component.
So now we declare two variables - **selectedChapterIndexToPublishUpToInDropdown** (integer) and **newChapterPublicationIsDisabled** (boolean) - in the **StoryEditorStateService** class.
For selectedChapterIndexToPublishUpToInDropdown, we define setter **setSelectedChapterIndexToPublishUpToInDropdown()** and getter **getSelectedChapterIndexToPublishUpToInDropdown()**. For newChapterPublicationIsDisabled, we define setter **setNewChapterPublicationIsDisabled()** and getter **getNewChapterPublicationIsDisabled()**. We also need another boolean -

chaptersAreBeingPublished, which is set to true if the selectedChapterIndexToPublishUpToInDropdown is greater than or equal to the last published chapter index, indicating new chapters are being published, else it is false, indicating that some existing chapters are to be unpublished. It will also have the setter **setChaptersAreBeingPublished()** and getter **AreChaptersBeingPublished().**

Thus when selectedChapterIndexToPublishUpToInDropdown in *story-editor.directive.ts* is set upon dropdown selection, it also sets the *StoryEditorStateService*.selectedChapterIndexToPublishUpToInDropdown with the same value. Now this can be accessed in the navbar component.

We also set the value of newChapterPublicationIsDisabled, if the condition (given in next section) is satisfied in the editor component. It is used to disable the Publish button in the navbar component.

story-editor.directive.ts:

Disable Publish button on navbar, if selectedChapterIndexToPublishUpToInDropdown is 0 and chapterIsPublishable[selectedChapterIndexToPublishUpToInDropdown] is false, (that is, if all chapters are in draft state) to indicate no publishable chapters.
Then call StoryEditorStateService.setNewChapterPublicationIsDisabled() with true.

When selectedChapterIndexToPublishUpToInDropdown is set (on selecting a value from Publish Upto Chapter dropdown), store it in StoryEditorStateService.selectedChapterIndexToPublishUpToInDropdown by calling setSelectedChapterIndexToPublishUpToInDropdown()**.**

story-editor-navbar.component.ts:

To disable the Publish button here, declare a boolean variable - publishButtonIsDisabled - and set it true if StoryEditorStateService.getNewChapterPublicationIsDisabled() is true. So the Publish button is actually disabled when, apart from existing conditions, publishButtonIsDisabled is true.

If **StoryEditorStateService.chaptersAreBeingPublished** is true, then the button shows "Publish".

To handle the "Publish" button click, call changeChapterStatus() (same function in Save as Ready To Publish above).
In this case, when the status passed is "Published", then for **each of the nodes from first unpublished index to the selectedChapterIndexToPublishUpToInDropdown** (received from StoryEditorStateService.getSelectedChapterIndexToPublishUpToInDropdown()), we call the following functions of *story-update.service.ts***:**

- setStoryNodeStatus() with parameter "Published"

- setStoryNodeUnpublishingReasonId() with value null, to indicate the story is published, so that if any unpublishing reason exists (if the chapter is getting re-published), it is removed now.
- setStoryNodeFirstPublicationDate() with the current date using Date object, only if the field firstPublicationDate is null.

Finally call saveChapter() of *story-editor-state.service.ts* to save the change, by making a put request to the EditableStoryDataHandler class via url **/story_editor_handler/data/<story_id>.**

One more change that needs to be made here is that, if the story is published (that is any chapter of it is in published state) then we change “Save Draft” button to “Save Changes”" to also save any changes made in published parts.

A basic pseudocode implementation of changeChapterStatus() in *story-editor-navbar.component.ts*

(In the image below, read setStoryNodePublicationDate as setStoryNodeFirstPublicationDate)

```
changeChapterStatus(newStatus: string, index: number) {
  if(newStatus == 'Ready To Publish') {
    this.storyUpdateService.setStoryNodeStatus(this.story, nodeId, newStatus)
  }
  else if (newStatus == 'Published') {
    get the first story node in this.story.story_contents with Status == 'Ready To Publish'

    for each story node from the found node till index {
      this.storyUpdateService.setStoryNodeStatus(this.story, nodeId, newStatus)
      this.storyUpdateService.setStoryNodePublicationDate(this.story, nodeId, currentDate)
      this.storyUpdateService.setStoryNodeUnpublishingReasonId(this.story, nodeId, null)
    }
  }
  this.storyUpdateService.saveChapter();
}
```

Also, if the first element of linearNodesList in *story-editor.directive.ts* did not have status == "Published" (that is, first time in a story, a chapter is getting published), then also call publishStory() (existing function in this file), to mark the story as published.

Web frontend changes

story-editor-navbar.component.html:

“Publish” button is rendered if StoryEditorNavigationService.getActiveTab() == ‘story_editor’. The button is disabled if the variable publishButtonIsDisabled(defined above in its ts file) or any of the existing disable conditions is true. On click, call changeChapterStatus() with the arguments “Publish” and selectedChapterIndexToPublishUpToInDropdown.

Documentation changes

None

6. Save Draft Feature (See different buttons for different status in chapter editor page)

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

story-editor-navbar.component.ts

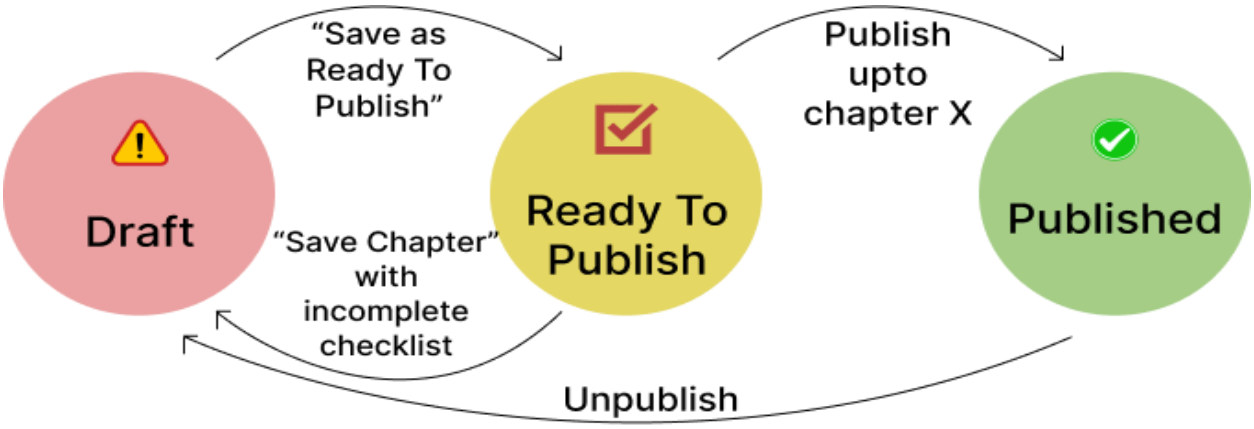
On opening chapter editor page, there are 3 possibilities:

- If the chapter’s existing status (see above how to find) is “Draft”, then no changes in frontend.

- If the chapter’s existing status is “Ready To Publish”, then check the checklist flag (defined above). If the flag is false (checklist not fulfilled), then upon clicking the “**Save Chapter**” button, a modal pops up, warning the change of status to Draft, on confirming which, we change its status to “Draft” by calling setStoryNodeStatus() of *story-update.service.ts* with “Draft”.
  This is how a Ready To Publish chapter can be changed back to status Draft. If the flag is true, then status is not changed, and we can directly call saveChapter() of *story-editor-state.service.ts*.
  There will also be a navbar button “Change To Draft”, on clicking which, the chapter status changes to “Draft” by calling setStoryNodeStatus() of *story-update.service.ts* with “Draft”, then saveChapter() of *story-editor-state.service.ts*.

- If the chapter's existing status is "Published", then also check the checklist flag. If the flag is true, then simply save the edited properties of the published chapter. But if false, **then the "Publish Changes" button is disabled**, with a tooltip message, to either fulfill the checklist, or unpublish the chapter to save these changes. This is because a published chapter cannot have an unfulfilled checklist, nor can it be converted to Draft, without formally unpublishing it.

Diagram representing how to convert chapter from one status to another



story_services.py:

def apply_change_list(): Further if elif conditions have to be added to save changes in the new StoryNode properties, that is, status, planned_publication_date, last_modified, first_publication_date and unpublishing_reason, defined earlier in *story_domain.py*. (see image below)

Web frontend changes

None

Documentation changes

None

Here we need to add more elif conditions inside the elif (change.cmd == story_domain.CMD_UPDATE_STORY_NODE_PROPERTY) corresponding to new properties in StoryNode class in story_domain.py

```
# Repository SAVE and DELETE methods.
def apply_change_list(
    story_id: str, change_list: List[story_domain.StoryChange]
) -> Tuple[story_domain.Story, List[str], List[str]]:
    """Applies a changelist to a story and returns the result. …
    story = story_fetchers.get_story_by_id(story_id)
    exp_ids_in_old_story = story.story_contents.get_all_linked_exp_ids()
    try:
        for change in change_list:
            if not isinstance(change, story_domain.StoryChange):
                raise Exception('Expected change to be of type StoryChange')
            if change.cmd == story_domain.CMD_ADD_STORY_NODE: …
            elif change.cmd == story_domain.CMD_DELETE_STORY_NODE: …
            elif (change.cmd ==
                  story_domain.CMD_UPDATE_STORY_NODE_OUTLINE_STATUS): …
            elif change.cmd == story_domain.CMD_UPDATE_STORY_NODE_PROPERTY:
                if (change.property_name ==
                        story_domain.STORY_NODE_PROPERTY_OUTLINE): …
```

7. Story Control Panel in Topic Editor Page(UI Mock)

Storage Model Layer Changes

None

Domain Objects

story-summary.model.ts:

The story data displayed in the story control panel table is of type StorySummary.
So add following new number-type fields to frontend model **StorySummary** (and **StorySummaryBackendDict**):

- _publishedChaptersCount: total number of chapters in story with status == “Published”
- _totalChaptersCount: total number of chapters in story
- _upcomingLaunchesCount: total number of chapters planned to be launched within 14 days
- _overdueChaptersCount: total number of unpublished chapters with plannedPublicationDate < currentDate

These will be needed for displaying in the Story Control Panel in Topic Editor Page.

Existing StorySummary Frontend Model

```
export class StorySummary {
  constructor(
    private _id: string,
    private _title: string,
    private _nodeTitles: string[],
    private _thumbnailFilename: string,
    private _thumbnailBgColor: string,
    private _description: string,
    private _storyIsPublished: boolean,
    private _completedNodeTitles: string[],
    private _urlFragment: string,
    private _allNodes: StoryNode[],
    private _topicName: string | undefined,
    private _topicUrlFragment: string | undefined,
    private _classroomUrlFragment: string | undefined
  ) {}
```

User Flows (Controllers and Services)

We need to display Title, Story State, Chapters Published/Added and Chapter Publication Notifications.

topic_editor.py:

In the endpoint, from where story control panel data is fetched, that is TopicEditorStoryHandler, def get() (url: **/topic_editor_story_handler/<topic_id>**), also get the published_chapters_count, total_chapters_count, upcoming_chapters_count, overdue_chapters_count in response, for displaying in the Story Control Panel. (see the snippet below, modified from the existing endpoint, can be tracked by line numbers)

```
149         for summary in canonical_story_summary_dicts:
150             story = story_fetchers.get_story_by_id(summary['id'])
151             nodes = story.story_contents.nodes
152             total_chapters_count = len(nodes)
153             published_chapters_count = len([node for node in nodes if
154                                             node.status == 'Published'])
155             upcoming_chpaters_count = len([node for node in nodes if
156                                            ((node.planned_publication_date -
157                                              date.today()).days()<14 and
158                                              node.planned_publication_date >
159                                              date.today() and
160                                              node.status != "Published")])
161             overdue_chapters_count = len([node for node in nodes if
162                                           node.planned_publication_date < date.today()
163                                           and node.status != 'Published'])
164             summary.update({
165                 'total_chapters_count': total_chapters_count,
166                 'published_chapters_count': published_chapters_count,
167                 'upcoming_chpaters_count': upcoming_chpaters_count,
168                 'overdue_chapters_count': overdue_chapters_count
169             })
```

topic-editor-stories-list.component.ts:

It gets - storySummaries - fetched from the above endpoint. Now each element of the storySummaries, is a StorySummary frontend model instance, and hence with the new fields added above in the StorySummary model, it contains the required information for the 4 columns to be displayed in the panel table.

Web frontend changes

topic-editor-stories-list.component.html:

Four columns (Title, Story State, Chapters (Published/Added), Chapter Publication Notifications) will be included in the Story Control Panel, filled with the data in the storySummaries instances.

Documentation changes

None

8. Topics and Skills Dashboard Page(UI Mock)

Storage Model Layer Changes

None

Domain Objects

creator-topic-summary.model.ts:

The topic data displayed in this dashboard page is of type CreatorTopicSummary. So we must add following fields to the frontend model - **CreatorTopicSummmary** (and **CreatorTopicSummmaryBackendDict**)

- totalUpcomingChapters
- totalOverdueChapters
- publishedStoriesChaptersCount (list of story objects with 2 fields: count of published chapters, count of total chapters in the story)

Existing CreatorTopicSummary Frontend Model

```
export class CreatorTopicSummary {
  constructor(
      public id: string,
      public name: string,
      public canonicalStoryCount: number,
      public subtopicCount: number,
      public totalSkillCount: number,
      public totalPublishedNodeCount: number,
      public uncategorizedSkillCount: number,
      public languageCode: string,
      public description: string,
      public version: number,
      public additionalStoryCount: number,
      public topicModelCreatedOn: number,
      public topicModelLastUpdated: number,
      public canEditTopic: boolean,
      public isPublished: boolean,
      public classroom: string | undefined,
      public thumbnailFilename: string,
      public thumbnailBgColor: string,
      public urlFragment: string) { }
```

User Flows (Controllers and Services)

topics_and_skills_dashboard.py:

Inside the endpoint from where the topics and skills dashboard data is fetched, that is TopicsAndSkillsDashboardPageDataHandler def get() (endpoint url: **/topics_and_skills_dashboard/data/**), each element of topic_summary_dicts must also include the three additional fields(upcoming_chapters_count, overdue_chapters_count and published_stories_chapter_counts). It can be coded as follows (modified in the existing code).

(**A slight modification to the image below** - **upcoming_chapters_count** and **overdue_chapters_count** should be passed separately and not inside another object named

```
123         for topic_summary in topic_summary_dicts:
124             topic = topic_fetchers.get_topic_by_id(topic_summary['id'])
125             upcoming_chapters_count = 0
126             overdue_chapters_count = 0
127             published_stories_chapters_count = []
128             for story_reference in topic.canonical_story_references:
129                 story = story_fetchers.get_story_by_id(story_reference.story_id)
130                 nodes = story.story_contents.nodes
131                 total_chapters_count = len(nodes)
132                 published_chapters_count = len([node for node in nodes if
133                                                 node.status == 'Published'])
134                 upcoming_chapters_count += len([node for node in nodes if
135                                                 ((node.planned_publication_date -
136                                                     date.today()).days()<14 and
137                                                     node.planned_publication_date >
138                                                     date.today() and
139                                                     node.status != "Published")])
140                 overdue_chapters_count += len([node for node in nodes if
141                                                 node.planned_publication_date < date.today()
142                                                 and node.status != 'Published'])
143                 published_stories_chapters_count.push({
144                     'total_chapters_count': total_chapters_count,
145                     'published_chapters_count': published_chapters_count
146                 })
147             notification_chapters_count = {
148                 'upcoming_chapters_count': upcoming_chapters_count,
149                 'overdue_chapters_count': overdue_chapters_count
150             }
151             topic_summary.update({
152                 'notification_chapters_count': notification_chapters_count,
153                 'published_stories_chapters_count': published_stories_chapters_count
154             })
```

notifcation_chapters_count, and **published_stories_chapters_counts** should be replaced with **published_stories_chapter_counts)**

topics-and-skills-dashboard-page.component.ts:

It has the (existing) variable totalTopicSummaries (of type CreatorTopicSummary[])**,** that contains the list of topic data to be displayed, which is filtered, paginated and passed to the topic-list component as another variable - displayedTopicSummaries, and thus now the list contains the new fields totalUpcomingChapters, totalOverdueChapters and publishedStoriesChaptersCount for each topicSummary, that will be used to fill the respective columns.

Web frontend changes

topics-list.component.html:

Additional Columns for **“**Added Stories” and “Chapter Publication Notifications” introduced, and Published Stories more detailed.
Now we can fill the “Added Stories” column with the topicSummary’s canonicalStoryCount (already existing field in CreatorTopicSummary, see image of the model above). Fill the Chapter Publication Notifications column with the values of the new fields totalUpcomingChapters and totalOverdueChapters, and fill the Published Stories column with the value of the new field publishedStoriesChaptersCount.

Filtering and Sorting Feature

topics-and-skills-dashboard-filter.model.ts:

As default sorting is based on total number of chapter launch notifications, the existing function reset() must have this.sort = most upcoming launches + most launches behind schedule (represented as constants defined in next section)

topics-and-skills-dashboard-page.constants.ts:

Modify constants and enums for filtering based on chapter’s published status and new sorting options (Most upcoming launches and Most launches behind schedule).

topics-and-skills-dashboard-page.service.ts:

To sort the topics list as per new criteria, in getFilteredTopics(), also add switch cases for upcoming and behind schedule launches.

Documentation changes

None

9. Curriculum Admin Email Feature

Storage Model Layer Changes

No changes

Domain Objects

None

User Flows (Controllers and Services)

Mailing Curriculum Admin

We can schedule the weekly mail to curriculum admins about the upcoming launches and behind schedule launches with the help of Google Cloud Scheduler, where a cron job can be created and scheduled for a fixed time in the week (see docs for how), with the endpoint url **/cron/mail/curriculum_admins/chapter_publication_notfications/**.

A new handler class CronMailChapterPublicationsNotificationsHandler in *cron.py* collects all curriculum admin ids and published topicIds, checks if the topics have any story nodes with either of the 2 needed criteria (behind schedule launches, or planned publication date in the next 14 days and status not published) and forms two dicts of stories (one for behind schedule, another for upcoming), which contain the chapter names that are behind schedule/upcoming in the respective dicts. Then it calls a function - send_reminder_mail_to_notify_curriculum_admins() in *email_manager.py* with the curriculum_admin_ids and the two dicts. The function send_reminder_mail_to_notify_curriculum_admins() then forms the links of the story, and adds the chapters in an email body template, that can be sent using mailgun service (existing).

Web frontend changes

None

Documentation changes

None

10. Story Tab in Topic Viewer Page(UI Mock)

Storage Model Layer Changes

UserSubscriptionsModel: New field added - topic_ids - topic subscription list of a learner.

Domain Objects

None

User Flows (Controllers and Services)

topic_viewer.py:

In the backend response, we must not pass any “Draft” chapter to be displayed on the story tab of the topic viewer page.
So while fetching the story data from the handler class - TopicPageDataHandler def get(), in all_node_dicts of canonical_story_dicts, only add chapters with status "Published" and "Ready To Publish".

topic-viewer-backend-api.service.ts:

In the story tab of the topic viewer page, the user must either see the unsubscribe or subscribe button, depending on if they are already subscribed or not, otherwise if there are no “Ready To Publish” or “Draft” chapters in the topic, then the button is not displayed. We need this from the backend.
So, response from _fetchTopicData() (obtained from TopicPageDataHandler in *topic_viewer.py*) should also contain a boolean/None parameter user_is_subscribed (similar to ProfileHandler in *profile.py*) and pass it to *topic-viewer-stories-list.component.ts,* which displays the story tab.

The parameter user_is_subscribed is set to true based on whether the topic_id is present in the topic_ids of the UserSubscriptionsModel associated with the user. Also if the topic has no upcoming chapters (that is no chapter in any of its stories is in Ready To Publish or Draft state), also pass – topic_has_upcoming_chapters as false, else true.

(A slight modification to the image below – read **learner_topic_ids** as **topic_ids**)

```
is_already_subscribed = False
subscribed_learner_topic_ids = subscription_services.get_all_topics_subscribed_to_learner(self.user_id)
if topic.id in subscribed_learner_topic_ids:
    is_already_subscribed = True
```

The get_all_topics_subscribed_to_learner() is defined in *subscription_services.py* as follows:

(A slight modification to the image below – read **learner_topic_ids** as **topic_ids**)

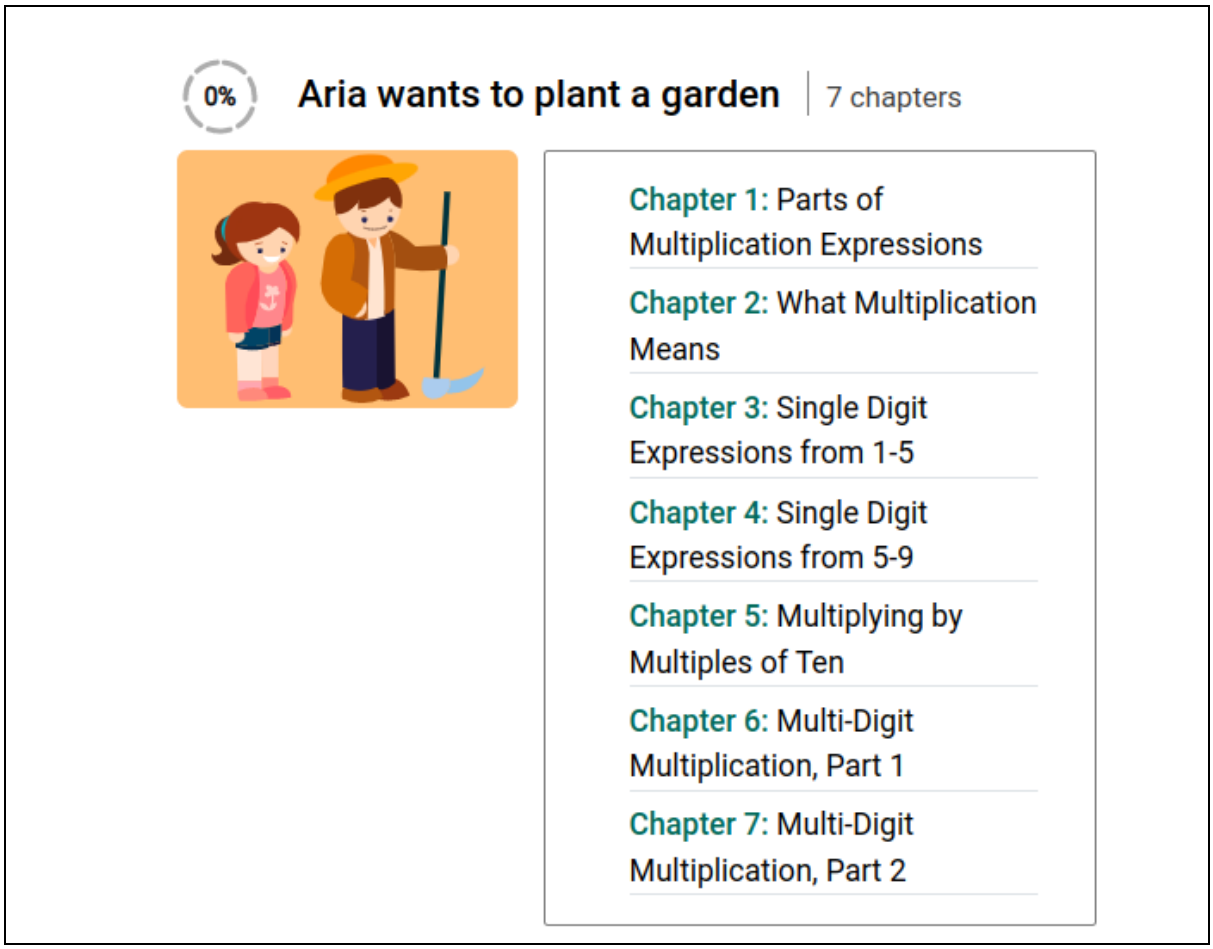
```python
def get_all_topics_subscribed_to_learner(user_id: str) -> List[str]:
    subscribers_model = user_models.UserSubscribersModel.get(
        user_id, strict=False)

    if subscribers_model:
        topic_ids: List[str] = subscribers_model.learner_topic_ids
        return topic_ids
    else:
        return []
```

story-summary-tile.component.ts:

This component is used to list chapter names, along with the story thumbnail, title, description and progress. Additionally declare 2 variables:

- availableNodeCount (count of published chapters)
- comingSoonNodeCount (count of ready to publish chapters)

Also, the existing variable - storyProgress - must be calculated by replacing nodeCount with availableNodeCount.



existing story summary tile component in the story tab of topic viewer page

Web frontend changes

story-summary-tile.component.html:

Add a title of the existing section "Available" (containing only the StoryNodes with status == "Published") and display availableNodeCount beside the heading

If any StoryNode instance inside the _allNodes field of the StorySummary has status == "Ready To Publish", create a new section "Coming Soon", and add only these story nodes titles, and display comingSoonNodeCount beside the heading

topic-viewer-stories-list.component.html:

Subscribe button beside the heading, on click opens **SubscriptionConfirmationModalComponent**, and passes userIsSubscribed (defined in its ts

file) to it. **If topic_has_upcoming_chapters is false, then no Subscribe/Unsubscribe button is shown.** That is, the learner does not see the option to Subscribe/Unsubscribe for the topics in which no more content is to be released. If they are already subscribed to such topics, then they will remain subscribed, so that if some release occurs later on, then they can be notified.

SubscriptionConfirmationModalComponent (UI Mock):

- If user is logged in, (can be obtained by using UserService.getUserInfoAsync(), userInfo.isLoggedIn()) and userIsSubscribed is false, get the email using getEmail() of UserInfo class. On OK click, a function - subscribeToTopicId() - is called which calls backendApiService and calls endpoint **/topic_subscription_handler/**. A put request is sent to a backend handler class, that adds the topic_id to the topic_ids list of UserSubscriptionsModel instance associated with the user_id.
- If logged in and userIsSubscribed is true, show Unsubscribe button, which calls another function - unsubscribeToTopicId() - and in the backend unsubscribes to the the topic by removing the topic_id from the topic_ids of the UserSubscriptionsModel.
- If not logged in, show the Create Account button, which redirects to the sign up page on click.

Documentation changes

None

11. End Card of Last Available Chapter in a Story(UI Mock)

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

StoryExpectationsDataHandler
New endpoint class (url: **/story_expectations_data_handler/**), used to provide all data to be shown in the end card of a story. It defines a get function, in which the story is obtained from the story_url_fragment (passed in the request).
The total number of upcoming story nodes of the story, which have their status as “Ready To Publish”, is used to set - upcoming_chapters_count**.**

The topics_ids list of UserSubscriptionsModel is used to check if the learner is already subscribed and sets - user_is_subscribed.(see code below). Also if the topic has no upcoming chapters (that is no chapter in any of its stories is in Ready To Publish or Draft state), also pass – topic_has_upcoming_chapters as false, else true.

story-expectations-data-backend-api.service.ts:

Defines function to make http get call to StoryExpectationsDataHandler, with request data storyUrlFragment (using urlParams of UrlService), to get upcoming_chapters_count, user_is_subscribed.

(A slight modification to the image below - read **StoryEndCardDataHandler** as **StoryExpectationsDataHandler,** another parameter in response – topic_has_upcoming_chapters, described as above)

```python
class StoryEndCardDataHandler(base.BaseHandler([Dict[str,str]],Dict[str,str])):
    """Provides the data to be displayed on the end card of a story's last published chapter"""
    def get(self):
        assert self.normalized_payload is not None
        story_url_fragment = self.normalized_payload['story_url_fragment']
        story = story_fetchers.get_story_by_url_fragment(story_url_fragment)
        nodes = story.story_contents.nodes
        upcoming_chapters_count = len([node for node in nodes if
                                       node.status == "Ready To Publish"])

        subscribed_topic_ids = subscription_services.get_all_topics_subscribed_to_learner(
            self.user_id
        )
        user_is_subscribed = story.corresponding_topic_id in subscribed_topic_ids

        self.values.update({
            'upcoming_chapters_count': upcoming_chapters_count,
            'user_is_subscribed': user_is_subscribed
        })
```

conversation-skin.component.ts:

We need to make sure that the displayed card is the **last card of the story**, and based on that we will display the new end card, with information as mentioned in the PRD. The endpoint call to decide whether it is the last chapter of the story (stored in the boolean – chapterIsLastInItsStory), and if true then fetching of end card data takes place in the init() function when the exploration is loaded. We will need

- New variable - upcomingChaptersCount - set from the above backend api service response.upcoming_chapters_count, is used to display the number of upcoming chapters in the end card.

- New boolean variable - hasSubtopics - is set to true when topicViewerBackendApiService.fetchTopicDataAsync() is called and subtopics.length>0. It determines if the revision card link should be displayed or not.
- classroomSuggestionLink and revisionSuggestionLink can be obtained using classroomUrlFragment and topicUrlFragment, which we can get using UrlService. These are the links to be displayed as recommendations on the end card.
- New boolean variable - userIsSubscribed - set using the above backend api service's response.user_is_subscribed, it determines whether to show Subscribe/Unsubscribe on the button if it is true or false. Also if response.topic_has_upcoming_chapters is false, then none of the two buttons will be shown.

Web frontend changes

conversation-skin.component.html:

If

- isOnTerminalCard() is true &&
- isCurrentCardAtEndOfTranscript() (that is, at last card of exploration) is true &&
- inStoryMode is true (that is, it is opened inside a story) &&
- chapterIsLastInItsStory is true

(that is, the last card of the last chapter of the story) then display the div of the end card.

The end card div contains upcomingChaptersCount, classroomSuggestionLink, revisionSuggestionLink (if hasSubtopics is true) as per UI Mock.

Subscription feature

- If topicHasUpcomingChapters is false, no Subscribe/Unsubscribe button is shown, else -
- Display Subscribe button, if userIsSubscribed is false
- Display Unsubscribe button, if userIsSubscribed is true
- On clicking Subscribe/Unsubscribe, similar action takes place as in the topic viewer page above.

Documentation changes

None

12. Recently Published Chapters Identifier Feature

We need the learners to identify the newly published chapters. It has 2 conditions:

- The chapter has been published in the past 28 days from the current date
- The chapter has not been opened/visited yet by the learner.

For the first condition, we have the first_publication_date field in the StoryNode. But for the next condition, to keep track if the user has visited a chapter, we need to have a new field. **This is**

very similar to how the information whether a user has completed a chapter is stored, using the existing StoryProgressModel.

Storage Model Layer Changes

New field - **visited_node_ids** - in the StoryProgressModel is introduced.
This field is used to keep track of which chapters of a story have been visited once by a learner, so that the New Chapter identifier is not displayed again on the visited chapters.

Note: In implementation, it is very similar to the existing field **completed_node_ids**, that keeps **track of chapters completed by learners.** It will also have a service function to record when a new chapter is opened(on making a backend request to record the chapter as visited), and update the visited_nodes_ids, if the chapter id is not present in the list. Also, if a visited chapter is unpublished, then it is removed from this list, to maintain coherency with the current state of data, that is, only the currently published chapters must be listed in it.

```
def record_visited_node(
        user_id: string, story_id: string, node_id: string
) -> None:
    visited_model = user_models.StoryProgessModel.get_or_create(
        user_id, story_id)

    if node_id not in visited_model.visited_node_ids:
        visited_model.visited_node_ids.append(node_id)
        visited_model.update_timestamps()
        visited_model.put()
```

Classroom Page(UI Mock**)**

Domain Objects

creator-topic-summary.model.ts:

The topic tiles displayed in the classroom page are of type CreatorTopicSummary.
So, add another new field newlyPublishedChaptersExist to **CreatorTopicSummmary** and **CreatorTopicSummmaryBackendDict**. (two fields have already been added to it before)

User Flows (Controllers and Services)

classroom.py:

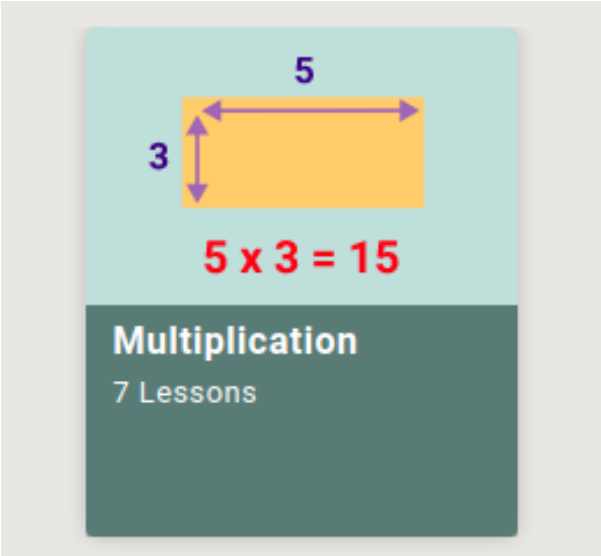
To know whether there are new chapters in the topic, we must also fetch “newlyPublishedChaptersExist” for the CreatorTopicSummary, while fetching the classroom page data. So, inside response of ClassroomDataHandler def get() (endpoint url: **/classroom_data_handler/<classroom_url_fragment/**), pass another parameter - newly_published_chapters_exist - in each dict of topic_summary_dicts, set to true if current date - first_publication_date of **any** of its published chapters is < 28 days and that chapter is not visited before(if its node_id is not present in the visited_node_ids of the StoryProgressModel instance associated with the user and the story)

Web frontend changes

topic-summary-tile.component.html:

The topicSummary (which is of type CreatorTopicSummary) obtained from the above response is passed to this component to display a topic summary tile in the classroom page. Add the label in red "New Chapter" if topicSummary.newlyPublishedChaptersExist is true.

existing topic summary tile in the classroom page



Topic Viewer Page(UI Mock)

Domain Objects

story-summary.model.ts:

Add a new field - _visitedChapterTitles - to **StorySummary** (and **StorySummaryBackendDict**). Also define a new function isNodeVisited(nodeTitle) that checks if the node with the title is present in the visitedChapterTitles array, and returns true/false accordingly.

This is again very similar to the existing field _completedNodeTitles and function isNodeCompleted(nodeTitle), which are responsible for tracking which chapters are completed.

```
isNodeCompleted(nodeTitle: string): boolean {
  return (this._completedNodeTitles.indexOf(nodeTitle) !== -1)
}
```

Existing function isNodeCompleted() in *story-summary.model.ts*

User Flows (Controllers and Services)

topic-viewer-backend-api.service.ts:

When topic viewer page data is fetched from _fetchTopicData(), the response should also contain visited_node_titles for each story in the canonical_story_dicts of the topic. These canonical_story_dicts are stored as StorySummary instances in the frontend.
(The parameter visited_node_titles is calculated in the TopicPageDataHandler get function of *topic_viewer.py*, similar to how completed_node_titles is calculated there).

Web frontend changes

story-summary-tile.component.html:

“New” tag must be added if storySummary.isNodeVisited(nodeTitle) is false and current date - storynode._first_publication_date < 28 days && storynode.status == “Published”, where storynode is the StoryNode instance associated with the loop variable nodeTitle.

“Continue where you left off” Section in Learner Dashboard Page(UI Mock)

Domain Objects

None

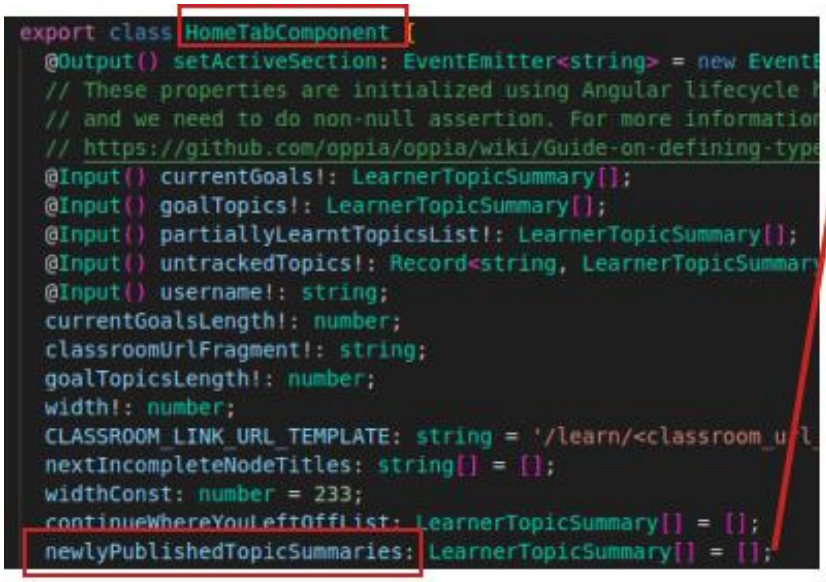
User Flows (Controllers and Services)

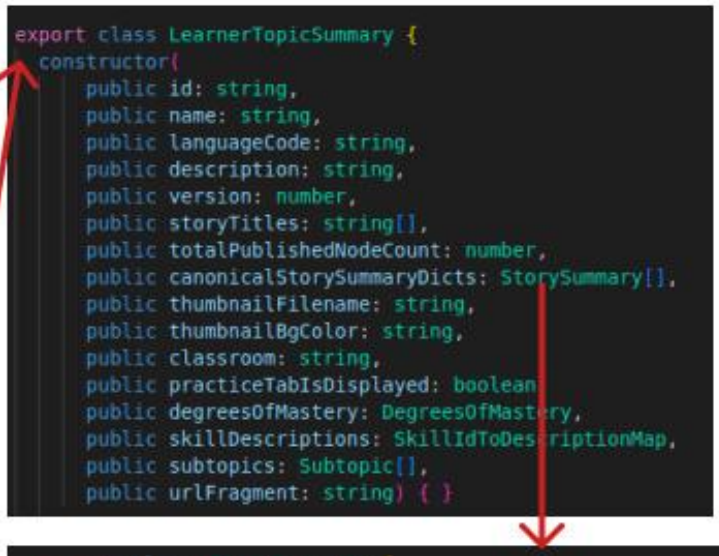
Currently, “Continue where you left off” section only displays the list of **all** story cards which have been set as goals or belong to partially learnt topics. (**Note: only the “Published” story nodes must be passed in these topics’ stories from backend response.**)
Now we also need to display those **learnt topics, which have newly published chapters.**

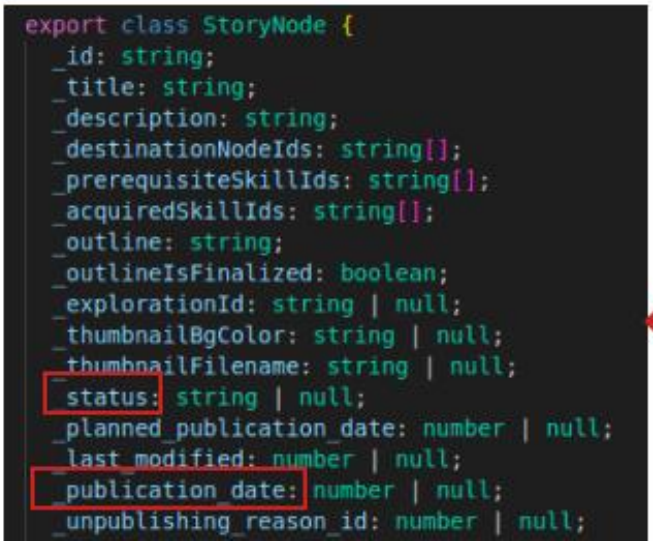
home-tab.component.ts:

Hence, this component also receives learntTopics from the parent component (*learner-dashboard-page.component.ts*).

Now we will **separately** display the topics with newly published chapters. So create a new array - newlyPublishedTopicSummaries (of type LearnerTopicSummary[]).
Each element of this array is a LearnerTopicSummary instance, and its canonicalStorySummaryDicts field (of type StorySummary[]) consists of a number of StorySummary instances, each of which will contain **only the first node** in its _allNodes list, having status == “Published”, title not present in “visitedChapterTitles” and current date - _firstlPublicationDateMsecs < 28 days. (see this flowchart that represents the description)







Code of how the newlyPublishedTopicSummaries array will be created

```
for (var learntTopic of this.learntTopicsList) {
  let newStorySummaryList: StorySummary[]= [];
  for (var storySummary of learntTopic.canonicalStorySummaryDicts) {
    let nodes = storySummary.getAllNodes();
    let firstNewlyPublishedNodeFound = False;

    for (let i = 0;i < nodes.length && !firstNewlyPublishedNodeFound; i++) {
      let currentDate = new Date().geTime();
      let dateDiff = (currentDate - nodes[i].getPublicationDate())/(1000*60*60*24);

      if(nodes[i].getStatus() == "Published" && dateDiff < 28
      && !storySummary.isNodeVisited(nodes[i].getTitle())) {
        firstNewlyPublishedNodeFound = true;

        newStorySummaryList.push(new StorySummary(
          storySummary.getId(),
          storySummary.getTitle(),
          storySummary.getNodeTitles(),
          storySummary.getThumbnailFilename(),
          storySummary.getThumbnailBgColor(),
          storySummary.getDescription(),
          storySummary.isStoryPublished(),
          storySummary.getCompletedNodeTitles(),
          storySummary.getUrlFragment(),
          [nodes[i]],
          storySummary.getTopicName(),
          storySummary.getTopicUrlFragment(),
          storySummary.getClassroomUrlFragment(),
          storySummary.getVisitedNodeTitles()
        ))
      }
    }
  }
  learntTopic.canonicalStorySummaryDicts = newStorySummaryList;
  this.newlyPublishedTopicSummaries.push(learntTopic);
}
```

Web frontend changes

home-tab.component.html:

The story cards of the partially learnt topics should have links to the chapter from where the learner left off, rather than directly jumping to the newly published chapter, so we do not display the label and the new chapter information on these topics.

For the learnt topics with newly published chapters, create a **separate div for loop**, that displays oppia-learner-story-summary-tile component, but this loop will be on the newly constructed array (newlyPublishedTopicSummaries in the above ts file).

The main reasons behind separately displaying the topics with newly published chapters are:

- A parameter (hasNewlyPublishedChapters) can also be passed to the oppia-learner-story-summary-tile component to display thumbnail of the first newly published chapter card, as given in PRD, (and **not the story thumbnail**) and a "New Chapters Available" label will also be added to these tiles.
- They can be displayed at the beginning of the Continue where you left off section, before the partially learnt topics.

Documentation changes

None

13. Email to Subscribed Learners(UI Mock)

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

Whenever chapters are published, that is if in apply_change_list() of *story_services.py,* story node status property is being changed to “Published” for any of the story node, get all the users subscribed to the topic (UserSubscriptionModel instances which have this chapter’s topic_id in the topic_ids), and add all the newly published nodes link in the email template, attach the template to the email body and send mail to the the users with the help of mailgun service.

Web frontend changes

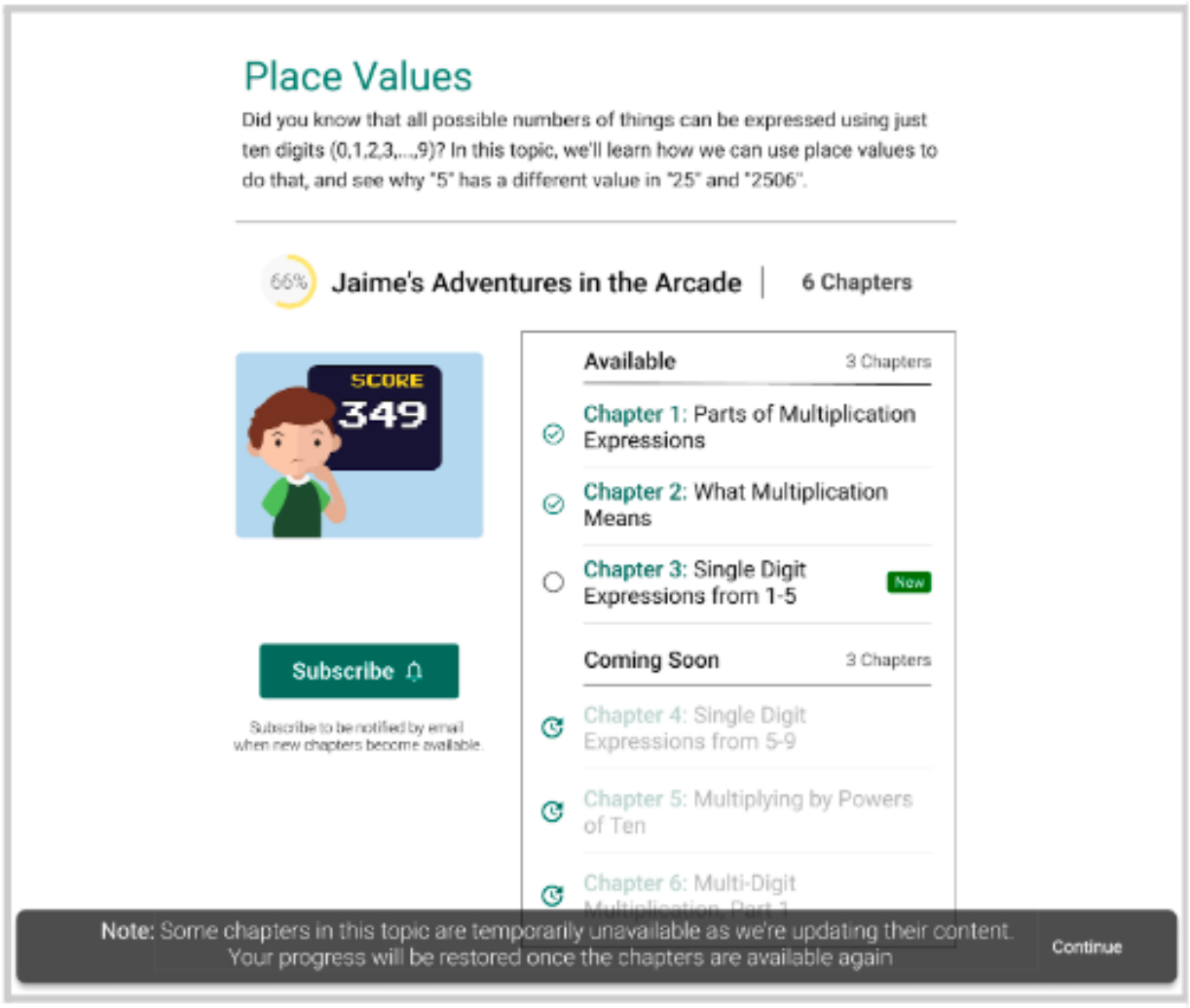
None

Documentation changes

None

14. Unpublished Chapters Notification Modal to Learners

We need to show a notification modal in the topic viewer page when some chapters have been unpublished.



(*taken from PRD*)

This message will be shown **only for the first time**, when a chapter is unpublished and not yet notified to the learner in the topic viewer page. The text of the message is dependent on the unpublishing reason selected by the curriculum admin while unpublishing, given in the PRD table.

Storage Model Layer Changes

To track if a chapter's unpublishing has been notified to the learner, we need a new field - **unpublished_and_notified_node_ids** in the existing model StoryProgressModel. This field is used to keep track of all the story node ids of the story that have been unpublished and they have been notified to the user.
When an unpublished chapter is published back, and if it is present in the unpublished_and_notified_node_ids field of any instance, it is deleted from the list.

Domain Objects

None

User Flows (Controllers and Services)

topic-viewer-backend-api.service.ts:

Response from _fetchTopicData(), that gets the information in the story tab of the topic viewer page, should also contain a message string - unpublishing_notification_message and pass it to *topic-viewer-stories-list.component.ts.* It is the message displayed to the learners based on the reason for unpublishing, as specified in th PRD table. To check whether a chapter has been attempted (for "Bad Content" reason), we can check if the chapter_id is present in the completed_node_ids of the StoryProgressModel.
These unnotified nodes will now get added to the unpublished_and_notified_node_ids list.

If unpublishing_notification_message is None, then no notification is displayed.

Web frontend changes

topic-viewer-stories-list.component.html:

Show the modal (**UnpublishedChaptersNotificationsModalComponent**), if from the backend response, unpublishing_notification_message is not None.

UnpublishedChaptersNotificationsModalComponent
The message displayed is determined based on the backend response's unpublishing_notification_message.

Documentation changes

None

15. Translation Contributor Dashboard Changes

Storage Model Layer Changes

None

Domain Objects

None

User Flows (Controllers and Services)

In the translation contributor's dashboard, only the published lessons will be shown. For this, in the backend endpoint, ContributionOpportunitiesHandler **def get()**, which further calls get_translation_opportunities() in *opportunity_services.py*, we need to filter only those chapters in the opportunity_summaries list which have status == "Published".

Web frontend changes

Only the published chapters are available for translation.

Documentation changes

None

Testing Plan

Acceptance testing plan

#	Test name	Initial setup step	Step	Expectation
1.	Serial Chapter Feature Test	1 story in a topic, having ● 2 published chapters (A, B) ● 1 draft chapter (C) ● 1 Ready To Publish chapter (D) in above	Open the chapter control panel in story editor page	The new columns of status, last modified and planned publication date are visible.
			Click on the Kebab Menu icon for all the 4 chapters in sequence.	Chapter A only has the option to edit. Chapter B has the options to edit and unpublish Chapter C has the options to Move down, Edit, Delete. Chapter D has the options to Move up, Edit, Delete.
			Test kebab menu operations to move up and down. Move	● The draft chapter C moves to the 4th row and the Ready To Publish

		sequence	down the draft chapter C from the 3rd row to the 4th row. Move it up again to the 3rd row.	chapter D moves to the 3rd row. ● On moving the draft chapter up again, the sequence reverts to original.
			Unpublish the 2nd published chapter B with reason “Bad Content”.	● The page reloads. Chapter B’s status has changed to “Draft”, and its planned publication date is empty. ● Navigate to the topic viewer page for the topic. Inside the story, only the first chapter A is available for learning. A notification modal at the bottom displays the learner message corresponding to the “Bad Content” unpublishing reason. ● In the contributor dashboard page, for this topic, only the published chapter A is available for translation.
			Edit the Ready To Publish chapter D in the chapter editor page. Remove the planned publication date value and click “Save Chapter”. Click “Confirm” in the modal that pops up, confirming the status change of the chapter to “Draft”.	● On confirming, D’s status in chapter control panel is changed to Draft and the planned publication date is empty.
			Edit the same chapter D, now Draft, in Chapter Editor Page , by refilling the planned publication date with some past date - “1 Jan 2000” and save it as Ready To Publish.	● All the checklist points are marked in the chapter editor page. ● “Save as Ready To Publish” button is enabled in the navbar. ● After clicking on it, in the chapter control panel the chapter D’s status is shown as “Ready To Publish” and the planned publication date is “1 Jan 2000”.
			Re-edit the Ready To Publish chapter D in the chapter editor page, by changing the planned publication date from current date to “2 Jan 2000”, and click on “Save Chapter”.	● Chapter D’s status is retained as Ready To Publish in the chapter control panel. ● The planned publication date changes to “2 Jan 2000”.
			Rearrange the Ready To Publish chapter D to the 2nd row by drag and drop. Select “2” in the Publish Upto Chapter dropdown and publish it.	● The dropdown items from the first “Draft” chapter onwards (that is, chapters B and C) disabled. ● On Publishing, the status of the chapter D in the Chapter Control Panel changed to “Published”.

				• Navigate to the topic viewer page for that topic. Inside the story, now both chapters A and D are available for learning. D chapter has a “New” label. • In the contributor dashboard page, for this topic, the chapters A and D are available for translation.
			Edit the 2nd published chapter D in the chapter editor page. Remove the description field.	• The explorationId and planned publication date fields are non editable. • The “Publish Changes” button is disabled and not clickable.
			Refill the description with some other text.	The “Publish Changes” button is now clickable. On clicking the changes get saved.
2.	Tests to check serial chapter changes on learner side	1 story in a topic, with 2 “Published” chapters, with 1 recently published, 2 “Ready to Publish” chapters, and 1 unpublished chapter.	Open story tab of topic viewer page	An unpublished modal, with the unpublishing message, appears. There is a “New” tag beside the recently published chapter. There are 2 chapter titles in the ‘Coming Soon’ section.
			Subscribe to the topic.	The subscription confirmation modal opens with the user email id (if logged in). After clicking on Ok, now the button in the topic viewer page shows “Unsubscribe”.
			Complete the two published chapters in the topic.	The end card appears, with no. of upcoming chapters in topic displayed as 2, classroom and revision tab links and **unsubscribe** option present (user has already subscribed, so now unsubscribe)

Implementation Plan

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

I will utilize the community bonding period from May 4 till May 28 to develop better communication with my mentors, working on the precise project execution plan under their guidance and suggestions, understanding the various intricacies involving the project thoroughly and also taking care of any side undesirable consequences from the proposed changes.

Milestone 1

Key objective for this milestone:
Completing the Oppia team facing requirements, that includes:

- Serialising the chapter launch mechanism
- UI to track the story-wise chapter publication status and schedule.
- Email reminders that are sent to curriculum admins about upcoming and behind schedule launches.

All user flows will work on both desktop and mobile.
Each PR deadline includes the time to add the corresponding unit and acceptance tests.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
1	Add feature flag – **serialize_chapter_curriculum_admin_view** for all the milestone 1 tasks and providing the learners only the “Published” chapters to learn.	-	3 June	6 June
2	Add chapter control panel UI with all changes in models and frontend functions and also back-populate the new fields in storage model levels. After this, the user can view the modified chapter control panel table in the story editor page.	-	10 June	15 June
3	Add a new field – planned publication date and make associated function changes for changing the status and last modified fields of chapter during a new chapter creation, as described in the 2nd feature of my Implementation Approach.-After this, the user can create a new chapter as per the new chapter management system.	2	13 June	17 June
4	Modify chapter Editor Page UI and checklist feature. After this, the user can also edit the new “planned publication date” field in the chapter editor page and see the new checklist conditions get fulfilled on filling all the fields.	2, 3	17 June	21 June
5	Allow saving chapter as Ready To Publish feature. After this the user can save a draft chapter as Ready To Publish if the checklist is	2, 3, 4	19 June	22 June

	fulfilled.			
6	Allow saving Draft and publishing a chapter function. After this, the user can see different navbar buttons for different status chapters (described later in the doc) and can click on them to save draft or publish the chapter.	2	24 June	27 June
7	Modify the Story Control Panel in the Topic Editor Page. After this, the user will see the 2 additional columns - **Chapters (Published/Added)** and **Chapter Publication Notifications**, filled with their values in the Story Control Panel in Topic Editor Page.	2	26 June	28 June
8	Modify the Topics and Skills Dashboard. After this, the user can see the 2 additional columns - **Published Stories** and **Chapter Publication Notifications**, filled with their values, and also filter and sort the table based on new filtering and sorting options.	2	30 June	4 July
9.	Add curriculum admin reminder email feature. After this, the curriculum admins will be weekly emailed about the upcoming and behind schedule chapters.	2	5 July	7 July
10.	Demo to the PM for Milestone 1		7 July	
11.	Buffer time for bug fixes or other recommendations from milestone 1			

Milestone 2

Key objective for this milestone:
Completing the learner facing requirements, that include:

- Notifying the learner about upcoming launches.
- Identifying the newly launched and unvisited chapters to the learners in different pages, like topic viewer page, learner dashboard page and classroom page
- Subscription feature of learner to a topic
- Email notifications to the learners about the new chapters in the topics in which they have subscribed.

All user flows will work on both desktop and mobile.

Here also, each PR deadline includes the time to add the corresponding unit and e2e tests.

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
12.	Add feature flag - **serialize_chapter_learner_view** for all the learner facing changes resulting from the milestone 1 tasks.		18 July	20 July
13	Modify the Story tab in a topic-viewer page. After this, the learner can also view the upcoming chapters in the topic viewer page, but in grayed out format.	2	26 July	30 July
14	Add end card of the topic, with various information and links. After this, the user upon completion of all available topic chapters will see an end card, described in the mock.	2	5 August	10 August
15	Add “recently published” labels in learner dashboard, classroom page and topic viewer page. After this the learner will be able to identify the newly published content in the **learner dashboard Continue where you left off**, in the **classroom page** and also in the **topic viewer page**.	2	15 August	22 August
16	Add Learner subscription feature and emailing feature. After this, the learner will be able to subscribe to a topic and will receive emails about new launches.	2, 13	24 August	30 August
17	Create a new unpublishing message modal in the topic viewer page. After this the learner will be able to see a modal in the topic viewer page, informing about the unpublished chapters and an appropriate message.	2, 13	30 August	4 September
18.	Demo to the PM for complete project		6 September	
19	Buffer time for bug fixes or other recommendations from milestone 2 and the full project			

Launch Plan

The project can be launched in two phases, firstly, the curriculum admin side and chapter model level changes and secondly, the learner facing changes, and we will need a feature flag for each of the 2 phases, namely

1. **serialize_chapter_curriculum_admin_view**: This includes making storage model level changes to chapters, new features in the story editor, chapter editor and topic dashboard, and providing the learners only the “Published” chapters to learn.

2. **serialize_chapter_learner_view**: This includes the new frontend changes for learner side to reflect the features above.

Note: the second feature is dependent on the first feature, and cannot exist independently. Each feature can be launched following the wiki guidelines. The feature can be tested on the custom server, the buffer time kept in the above implementation plan at the end is for the bugs to be reported and fixed. After the bugs are resolved and the feature is finally approved, the feature will be ready to be announced to the public after filling this form.

Future Work*(referred from out of scope goals in PRD)*

***Note**: This section is mainly for reference (since it is understood that items in this section will not be part of the GSoC project). Proposals will primarily be evaluated based on the implementation plan above.*

After the end of the project, I can work on integrating the new serial chapter feature with the Translations team, as mentioned in the PRD, so that the new chapter publication notifications are also displayed in the Translations Admin dashboard and the team can accordingly translate the new content.