Moodle Essay PDF Annotator: A Standalone Plugin

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Nideesh N and VM Sreeram
(112001028 and 112001051)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Moodle Essay PDF Annotator: A Standalone Plugin**" is a bonafide work of **Nideesh N and VM Sreeram** (**Roll No. 112001028 and 112001051**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Jasine Babu
Associate Professor
Department of Computer Science & Engineering
Indian Institute of Technology Palakkad

Contents

List of Figures

Chapter 1

Introduction

Moodle is one of the world's most widely used open-source Learning Management System (LMS) platforms [1]. Moodle is provided under GNU General Public License and it is provided freely as open source. It supports more than 120 languages, has a huge developer base, extensive documentation, and user support. It is used at several educational institutions and organisations including IIT Palakkad. The primary implementation language used for Moodle is PHP. Moodle's design is modular, and its capabilities can be expanded through modules, referred to as plugins.

Moodle has "quiz" and "assignment" modules that allow teachers to evaluate performance of students by assessing answers shared over the LMS. Annotation is a very useful and convenient feature for the purpose of evaluation. The assignment module currently comes with a PDF annotation plugin. However, this plugin uses deprecated libraries and is no longer maintained. At present Moodle has neither a built-in feature nor a plugin to annotate uploaded PDF filetype answers to essay-type questions in quiz. There have been two previous attempts by Tausif Iqbal and Vishal Rao [2] and Parvathy S. Kumar and Asha Jose [3] of IIT Palakkad to address this limitation. The basic tool developed has been integrated with Moodle and is working fine. As pointed out by Moodle developer community [4], it does not qualify as a plugin since modifications have been carried out to the core Moodle code. The feedback also suggested to address security aspects before releasing the tool as a plugin. We also received some useful feedback from those who have used the aforementioned tool.

1.1 Objectives

The objectives for this project are:

- Develop a standalone Moodle plugin that allows the teacher to annotate the files uploaded as attachments by a student in a quiz.
- Address the security concerns raised by the Moodle developer community as feedback to the previous work.

1.2 Overview of the work

In this project we have reused code from the past works [2, 3]. The major tasks involved in our work are the following:

Making the tool standalone Earlier works [2, 3] involved changes to core Moodle code. However, the standard way to extend the functionality of Moodle is to have a standalone plugin. So work has been done to detach the code from core into a standalone plugin. This is discussed in detail in Chapter 4.

Addressing security concerns The past works did not consider addressing security aspects like capability and authentication checks for viewing pages and performing grading actions through annotations. Further, sanitization of POST parameters was also not done. These were raised as issues by the Moodle developer community. Enhancements were made to address the security guidelines. This is discussed in detail in Section 4.2.1.

Support for backup and restore For any activity plugin for Moodle, it is mandatory to have support for backup and restore. When a quiz is backed up, the annotated files should also get backed up so that they can be restored later. This feature was not supported by the earlier versions of the tool. In the current version, suitable changes are made to mark the annotated files for backup. This is discussed in detail in Chapter 5.

Aligning HTML, JavaScript, and CSS for compatibility According to Moodle development guidelines instead of plain HTML files, Mustache templates must be used, and JavaScript files should be modularized. Further, all the CSS styles used for plugin should reside in a single file. Adherence to these guidelines is discussed in Chapter 6.

Automated testing In the previous versions of the tool, only manual testing was done. Moodle uses the Behat [5] framework for performing automated tests. In the current version, we have included Behat tests for testing the functionalities in various scenarios. As Behat tests do not work when an exception is thrown, those scenarios were tested manually. We also tested the plugin in various Moodle and PHP versions. This is discussed in detail in Chapter 7. The bugs found during testing and their corresponding fixes are discussed in Chapter 8.

1.3 About our plugin

We have completed the implementation of the annotation tool as a standalone plugin. The repository containing our code and installation help are available at `https://github.com/vmsreeram/moodle-qtype_essayannotate`.

Chapter 2

Basic Setup and Familiarization Experiments with Moodle

In this chapter, we discuss the setup used for development and understanding concepts of Moodle relevant for plugin development. We will also explore the annotation feature in Assignment module towards the end of this chapter.

2.1 Setup used

Our basic setup of Moodle has the following:

- Webserver (Apache)
- Database (MySQL)
- PHP (version 8.1)

We started our development using Moodle version 4.1, which has long term support as the base code. For familiarizing with Moodle code base easily, we set up a local PHP debugger using the extension "PHP Debug" by Xdebug in the VS Code IDE [6]. This setup offers essential debugging features such as setting breakpoints, continuing execution, stepping over, stepping into, and stepping out of code. We can conveniently access the call stack and view the values of both local and global variables using this extension. This method of debugging was not used in the previous works [2] [3] on this problem.

2.2 Understanding the Moodle system

Roles, capabilities, and contexts are key concepts to understand before developing plugins as they determine user permissions and interactions.

2.2.1 Roles

A **role** is like a label for a user's status. It signifies what a user can do based on permissions they have for specific capabilities in a given context. Some default standard roles are Administrator, Teacher, Student, and Guest.

2.2.2 Capabilities

A **capability** in Moodle represents a specific permission or action that a user can perform. Each capability is identified by a unique string. For instance, `mod/quiz:manage` is the capability to manage quizzes. Capabilities are crucial for defining the roles and permissions within the Moodle platform.

Examples of Capabilities:

- `mod/forum:createattachment` - The capability to create attachments in a forum.
- `mod/quiz:grade` - The capability to grade quiz.
- `moodle/course:update` - The capability to update course settings.

2.2.3 Contexts

The **context** in Moodle refers to the space or environment in which a capability is evaluated for a specific role. Contexts help determine the scope of a user's permissions within the Moodle system. It is possible that a user can have different roles in different contexts simultaneously. Some contexts may also encapsulate others, forming a hierarchical structure.

Examples of Contexts:

- `course` - This is the context of an entire course, where users can have roles like teacher, student, or non-editing teacher.
- `module` - Represents the context of a specific activity module within a course, such as a quiz or forum.
- `system` - Refers to the overall site context, influencing global permissions and settings.

To illustrate, consider a user who is a teacher (`role`) in the `course` context for one course and a student (`role`) in the `course` context for another course. Additionally, the user may have administrative privileges (`role`) in the `system` context, allowing them to configure global settings.

2.3 Moodle plugins

Moodle plugins enhance the functionality of Moodle, allowing users and organizations to customize and extend its features beyond the core system provided by Moodle HQ.

2.3.1 Types of Moodle Plugins

Moodle offers various types of plugins to extend its core functionality. Important plugin types that are relevant to this Project are the following:

- **Block**: Block plugins allow us to show additional information in various parts of Moodle like in Dashboard, Course, etc. All the block plugins are present inside `/blocks` directory. Each block plugin should be present inside as a subdirectory with a mandatory `block_pluginname.php` file which will be used to render the block.

- **Assignment feedback**: Assignment feedback plugins are used to provide feedback on a user's assignment submission. They are located in `/mod/assign/feedback` directory. Moodle, by default, comes with four feedback plugins. The plugin responsible for PDF annotation feature is `editpdf`.

- **Quiz report**: It is used to analyze and display the results of quizzes. Additionally it can be used to add new behaviour to the quiz module. All quiz report plugins are present in `/mod/quiz/report`.

- **Question types**: It provides the various types of questions like Calculated, Multiple choice, Short answer, Numerical, etc. These question types are used by the quiz module. Any new type of question should be added inside `/question/type` directory.

2.3.2 The common files required for a Moodle plugin

The common files required for a Moodle plugin are the following [7].

- `version.php`: It contains the meta data about the plugin. It includes information such as the version number of plugin, list of dependencies, and the minimum Moodle version required. This is a mandatory file for every plugin.

- `lang/en/plugintype_pluginname.php`: Every plugin must have a set of language strings with at least one translation in English. This is a mandatory file for every plugin.

- `settings.php`: This file will contain the settings for the plugin that can be configured by the Administrator. This is an optional file.

- `db/access.php`: This file has the access control rules of the plugin. This is an optional file.

- `db/install.xml`: This file is used to define new database tables required for the plugin during its initial installation. This is an optional file.

2.4 Development of sample plugins for familiarization

By developing three plugins, a local plugin for notifying users, and two block plugins (one to list all users and the other to display active and hidden courses), we gained familiarity with various aspects of plugin development in Moodle. Through this process, we learned about integrating new tables into the database, designing user interfaces for plugin functionality, extending core classes like the block base class, and utilizing AJAX calls for dynamic content loading. We also explored how to perform database queries using global variables and manage dependencies between PHP scripts and JavaScript files.

2.5 Annotation feature in the assignment module

The Assignment module in Moodle is equipped with a plugin used for feedback annotation. It is called `editpdf`. The relevant code for this plugin is located in the directory `mod/assign/editpdf` and uses `yui.js` library. However, this is deprecated and no longer being maintained.

Our first objective is to convert the previous work [3] into a plugin. Towards this, we studied and analysed the design and working of the `editpdf` plugin. We found the implementation of the plugin to be complicated and sub-optimal to serve as a base for adapting it for developing Quiz plugin.

Chapter 3

Exploring Options for Standalone Plugin for Annotation in Quiz

The quiz activity module is used for creating and managing quiz assessments within Moodle LMS. It provides various features to design and deliver quizzes. Moodle's quiz module supports a variety of question types, including multiple-choice, true/false, short answer, essay, matching, and more. Out of these, here we deal with the essay question type, in which students can submit document and image files as answer. Currently there is no plugin which allows the teacher to annotate on the submission files like the one available for assignment.

3.1 Review of the past work implemented for quiz

The previous work [2, 3] has made annotation possible but the code makes some changes to the core Moodle and is not a plugin. The changes are made in `mod/quiz/comment.php` and `question/type/essay/renderer.php`. The option to annotate the submission files is available in `mod/quiz/comment.php` as a button. On clicking this button, it is redirected to `mod/quiz/annotator.php`. The overall flow of control to real the annotator is shown in Figure 3.1.

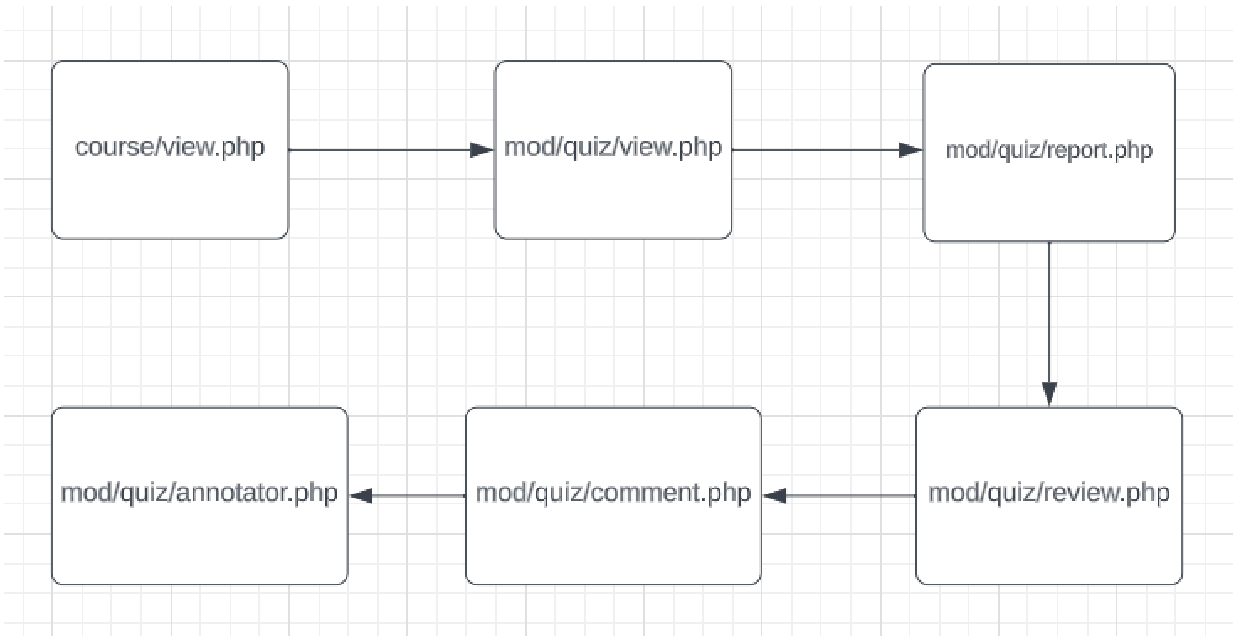


Figure 3.1 Flow to reach the PDF annotator in quiz

The solution uses FabricJS for annotations. The annotations are serialized with the help of FabricJS and then given to FPDF/FPDI along with the original PDF to get the annotated PDF. This annotated PDF file is stored in the database. The button was added in `mod/quiz/comment.php` by effecting necessary changes directly in the code. An alternate approach is required to develop a plugin that is acceptable to the developer community. We can extend the renderer of the essay question type to add this button because `mod/quiz/comment.php` uses this renderer. The rest of the code used by `mod/quiz/annotator.php` can be reused because it is independent of the Moodle code.

3.2 Potential ways to make a standalone plugin for annotation in quiz

We chose to explore four ways to make a standalone plugin for annotation in quiz after analyzing many plugin types. They are discussed here.

3.2.1 Creating a new question type that supports submission annotations

Question type plugins allow developers to create specialized question types beyond the standard ones provided by default in Moodle. We looked into one such plugin [8] that provides the additional functionality of autograde for essay type questions. After looking into its source code, we figured out that the standard question type, `essay`, was replicated and this custom functionality was added to it. So a similar approach can be taken.

This approach involving code duplication is not a robust solution. For instance, a flaw that is rectified in the original essay question type will have to be handled independently in the new question type plugin.

3.2.2 Creating a new quiz report plugin

Quiz report plugin can be used to add new functionality to the quiz module like providing custom reports on quiz data. Quiz report plugins should extend the `quiz_default_report` class and should implement `display()` method. For users to see the report of the new plugin, they should go to `mod/quiz/report.php` with query parameter `mode` (the name of the report plugin) and `id` (the context id). Looking at the functionality provided by the report plugin, we find it not very useful because, we would have to modify the renderer of essay question type to bring the Annotate button in `comment.php`.

3.2.3 Creating a local plugin

To add a new functionality in Moodle, the acceptable procedure is to develop a standard plugin, for example like Block, Activity, etc. But if the functionality does not fit into any of the standard plugin types, then local plugins are to be used. Local plugins are to be located in the `/local` directory. We saw a few examples of local plugins and concluded that we will consider making a local plugin only if no other viable means exist.

3.2.4 Creating question behaviour plugin

The question behavior type plugin allows for customization of how questions behave during assessments or quizzes. Question behaviour plugins are a part of the broader question engine in Moodle, which manages the presentation and behaviour of questions in various quizzes and exams. These plugins provide a way to control how questions are displayed, how they interact among one another, and how they are scored. The new question behaviour plugin renderer should extend the `qbehaviour_renderer` to make changes to the HTML output. To change the block of output present in the feedback area of the question, the method `feedback()` should be overridden. We tried to add an Annotate button in the Feedback area. The issue with question behavior is that this will be applied across all the questions in the quiz. However for questions of type essay, the custom behaviour will not take effect because its behavior is set to `question/behaviour/manualgraded` in the core code. So this type of plugin will not serve our purpose.

3.3 Conclusion

We were not able to find an acceptable way to directly extend `comment.php`. We found that `comment.php` is calling the renderer of the essay question type to list all the file submissions. So we decided to make a new question type with Annotate button beside every file. We also explored other plugin types like quiz report plugin, local plugin and question behaviour plugin.

Chapter 4

Design Changes to Create a Standalone Plugin

We started with duplicating the `essay` question type plugin. In this new plugin, we created a new folder called `Annotator` in which we placed all the code files responsible for the annotating logic.

4.1 Detaching from quiz

Since the past work [3] involved modifications to Moodle core code (inside the module Quiz) it has to be detached from core and made into a Moodle plugin.

4.1.1 Moving the interface to our plugin

The changes present in `mod/quiz/comment.php` is now incorporated in the `renderer.php` of the plugin, which is responsible for rendering a button to annotate beside every uploaded file. The button is actually an HTML link tag with reference to `annotator.php` with parameters attempt id, slot, and file number. The snippet below is the code for adding the button:

```
$url = new moodle_url("$CFG->wwwroot" . get_string('annotator_url',
        'qtype_essayannotate'),
        ["attempt" => $attemptid, "slot" => $slot, "fileno" => $filenum]);
$out .= html_writer::link($url, get_string('annotate_button_label',
        'qtype_essayannotate'),
        ['class' => 'btn btn-secondary qtype_essayannotate_annotatebtn']);
```

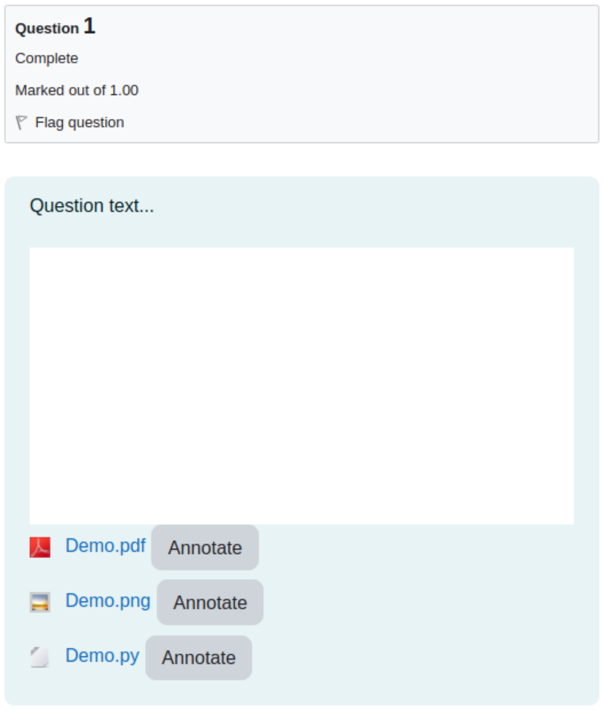


Figure 4.1 Annotate buttons beside each uploaded file in `comment.php`

4.1.2 Setting the page context for the annotator

Since `annotator.php` file was located in `mod/quiz`, the page context was already set. As we have moved the `annotator.php` to `question/type/essaynew`, we have to set the page context manually as it is being called from quiz context. To set the page context, context module id (`cmid`) is required. If `cmid` is not present in the POST request to `annotator.php` as a parameter, we query the database for `cmid` using the below query.

```
$sql = "SELECT cm.id AS cmid
        FROM {quiz_attempts} qa
        JOIN {course_modules} cm ON qa.quiz = cm.instance AND
        cm.module = (SELECT id FROM {modules} WHERE name = 'quiz')
        WHERE qa.id = :attemptid";
```

Below is the code for setting the page context after finding the `cmid`.

```
if (!empty($cmid)) {
    $cm = get_coursemodule_from_id('quiz', $cmid);
    $context = context_module::instance($cm->id);
    $PAGE->set_context($context);
}
```

4.2 Design improvements

4.2.1 Security aspects

Restricting access

In Moodle, the `require_login()` function is part of the Moodle API and is used to ensure that a user is logged in before accessing a particular page or resource. This was suggested by the Moodle developers community as an essential aspect to be addressed in the plugin. Accordingly, we have added `require_login()` calls in `annotator.php` and `upload.php`.

If an unintended user somehow gets to know the URL to annotate, for example like `[moodlebaseurl]/question/type/essayannotate/annotator/annotator.php?attempt=2&slot=1&fileno=1`, then she can annotate this file. To prevent this from happening we have added the following capability validation in `annotator.php`:

```
require_capability('mod/quiz:grade', $PAGE->context);
```

The capability `‘mod/quiz:grade’` is granted only to teachers and users with roles similar to editing TAs for the particular module. Similarly, to prevent unauthorised access to annotation saving logic in `upload.php` we chose to use the same capability check.

User data sanitization

Superglobals like `$_POST` should not be accessed directly, wrappers like `required_param()` with correct type declared should be used to sanitize input [9]. If the user data is not properly sanitized, malicious code or scripts can be injected into the application. Earlier we were directly accessing the superglobals, which we have now fixed.

4.2.2 Annotate button visibility control

To make sure the Annotate button is only displayed to users who have the capability to grade the quiz and display it only in the page where an attempt can be graded, we did the following to add the button:

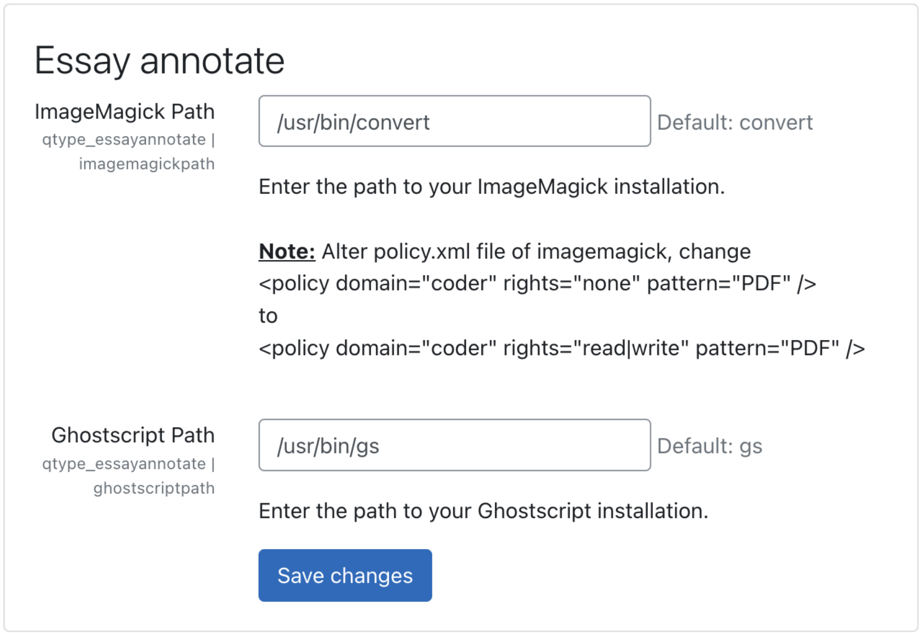
```
// Display Annotate button to teachers only in comment.php
// and not in review.php
if (has_capability(‘mod/quiz:grade’, $this->page->context) &&
    $options->manualcomment == question_display_options::EDITABLE) {
    // Display the button
}
```

4.2.3 Delayed accessibility to students

We noticed that students were able to see corrected documents even before grading of the question. So we added a condition that will display the corrected documents when the question is graded or the user has the capability to manage the quiz.

4.2.4 Settings page

Since the parser provided by FPDI only supports PDF versions up to 1.4, the past work uses Ghostscript to downgrade higher PDF versions to 1.4. It used ImageMagick to convert image and text files to PDF. Both of these are executed via shell. There was no way for admin to set paths to these executables. It is a general practice for plugins to get the required configurations from admin through a settings page. So we added a settings page (Figure 4.2) for the plugin in which the admin user can specify the paths to the executables.



Essay annotate

ImageMagick Path
qtype_essayannotate | imagemagickpath

/usr/bin/convert   Default: convert

Enter the path to your ImageMagick installation.

Note: Alter policy.xml file of imagemagick, change
<policy domain="coder" rights="none" pattern="PDF" />
to
<policy domain="coder" rights="read|write" pattern="PDF" />

Ghostscript Path
qtype_essayannotate | ghostscriptpath

/usr/bin/gs   Default: gs

Enter the path to your Ghostscript installation.

Figure 4.2 Admin settings page where paths to the excecutables of Ghostscript and ImageMagick can be specified

4.3 Adherence to plugin development guidelines

Before submission of the developed plugin to the Moodle plugin directory, certain guidelines are to be followed. These guidelines are given as plugin contribution checklist [9]. Our efforts in adhering to these guidelines are discussed below.

4.3.1 Backup and restore functionalities

Moodle insists that all activity module plugins should support backup and restore functionalities. The quiz module already implements these. However, for the new question type that we are creating, the annotated files were not getting backed up. Hence we have added this functionality. This is discussed in detail in Chapter 5.

4.3.2 Usage of language API

Moodle plugin development guidelines require that instead of hard-coding text in the code, the language strings should defined in a separate language file. The strings defined in these files can be accessed by using the `get_string()` function provided by Moodle's language API. In our plugin, the file `lang/en/qtype_essayannotate.php` contains the mapping of language string identifiers to actual text strings in the English language. These language strings are used throughout the question type plugin to display various labels, instructions, error messages, and other text elements.

4.3.3 Disabling browser access to internal pages

PHP files in plugins are either browser-accessible or internal files. Coding guidelines require that internal files should have the following code at the beginning:

```
defined('MOODLE_INTERNAL') || die();
```

This will prevent access of internal files from the browser.

Browser-accessible files must include `config.php`. By including `config.php`, the browser-accessible page becomes part of the Moodle application and can access Moodle's APIs, libraries, and other resources.

We included `config.php` in `upload.php` and `annotator.php` since these are the browser-accessible files in our plugin. Internal files in our plugin were also handled accordingly.

4.3.4 Reducing shell command usage and sanitizing

Moodle recommends using PHP functions wherever possible instead of `shell_exec()` commands. It also recommends that before using `shell_exec()` to execute the commands, they need to be escaped using `escapeshellcmd`.

The past work used `shell_exec()` to remove a file from `EssayPDF` using `rm`. We have changed this to use the PHP function `unlink()` instead. We found that PHP offers an extension `imagick` to create and modify images using the ImageMagick library. However, upon testing, we found that it is not supported for certain PHP versions that are supported by Moodle. There is no PHP function available for Ghostscript. Hence, we are using `shell_exec()` itself to execute Ghostscript and ImageMagick.

4.3.5 Adherence to Moodle coding style

All the PHP, JavaScript, and CSS code present in plugin should follow Moodle coding styles. There are various tools to check if code follows Moodle coding guidelines. One such tool for PHP files is a plugin, Code-checker [10]. It is used to analyze and review code written for Moodle plugins. It performs static code analysis to identify potential issues, adherence to coding standards, and best practices. We ran the tool against our code and made changes for necessary compliance.

4.3.6 Declaring third party libraries

According to Moodle plugin development guidelines, details of all third party libraries used in a plugin should be declared in the file `thirdpartylibs.xml` [11], which in turn is used to generate ignore file configuration for linting tools. This has been done.

Moodle also suggests not to add third party libraries that are already shipped in core. So we have removed third party libraries like JQuery, Bootstrap CSS, and Font Awesome CSS from the plugin. We also upgraded the remaining libraries to their latest versions. The third party libraries used in our plugin are the following.

- Fabric JS - JavaScript Canvas Library, SVG-to-Canvas (and canvas-to-SVG) Parser
- FPDF - FPDF is a PHP class which allows to generate PDF files with pure PHP
- FPDI - Collection of PHP classes facilitating developers to read pages from existing PDF documents and use them as templates in FPDF
- Alpha PDF - The AlphaPDF is an extended class on FPDI that is used for setting transparency
- PDF JS - PDF.js is a Portable Document Format (PDF) viewer that is built with HTML5
- PDF JS Annotations - Wrapper for PDF JS to add annotations. Suitable modifications were done to this code.

4.3.7 Converting HTML, JavaScript, and CSS to required formats

According to Moodle development guidelines Mustache templates must be used for rendering HTML output instead of plain HTML. Further, JavaScript files should be modularized. We have chosen Asynchronous Module Definition (AMD) format for this purpose. All the CSS styles used for a plugin should reside in a single `styles.css` file in the root of the plugin. The details are discussed in Chapter 6.

Chapter 5

Backup and Restore of Annotated Files

For an activity plugin in Moodle, implementing backup and restore functionality is crucial and it is listed as an approval blocker in the Moodle plugin contribution checklist [9]. The backup and restore [12] functionality will ensure that question data and the annotated files can be safely exported, preserved, and migrated between different Moodle installations or versions, enabling seamless content sharing, disaster recovery, and long-term maintenance of assessment resources.

This essential feature was not present in the previous works [2, 3]. Addition of this functionality is a major contribution of this project. In this chapter, we discuss the details of our work towards this.

5.1 Familiarisation with core backup code

We did a detailed debugging of Moodle's core backup code by going through the code execution during backup and restore of a course.

In Moodle's backup system, the backup plan is composed of multiple tasks and steps, each responsible for handling specific aspects of the backup process [13]. These tasks and steps are organized and executed in a predefined order to ensure a systematic backup operation. The order in which the backup is to be performed is defined in a type of step called structure step. The structure relevant to this plugin is `activity` → `quiz` → `quiz attempts` → `quiz attempt` → `question usage` → `question attempts` → `question attempt` → `steps` → `step`.

We inferred from our explorations that, in order to backup the annotated files added by our plugin, a new step has to be added to the question attempt on saving the annotated file. We did that and tried to backup and restore a course to check the functionality. We found that they were not being backed up as there is no link/relation between the question attempt and the annotated files.

The following are the important parts used in the logic to backup the question attachments:

- Question attempt – It represents the student's response to a particular question. `mdl_question_attempts` table stores the information about each individual question attempt within a quiz attempt.

- Question attempt steps – It refers to all interactions and actions taken by both students and teachers during the life cycle of a question attempt. Each of these steps/actions are stored in the `mdl_question_attempt_steps` table. When a student answers a question or attaches a file to the question it is considered as a step. Additionally, this table also stores the actions performed by teachers during the grading or manual review process, such as manual grading or comment additions.

- Question attempt step data – It refers to the detailed information associated with each individual step/action taken by a student or teacher. For the step in which student uploads file attachments as a response, corresponding to each attachment uploaded, there will be an entry added in the step data table. Similarly when the teacher gives feedback to the question attempt, the comment provided and the grade given will be added to the step data table. These are stored in the `mdl_question_attempt_step_data` table.

- Question usage by activity – The collection of all `question_attempts` that make up that quiz attempt is represented by a class called `question_usage_by_activity`.

- `mdl_files` – The `mdl_files` table is a core database table in Moodle that stores information about files uploaded to or used in the LMS. Files are conceptually stored in file areas. The file area being used to store the file attachments by students and the PDF annotated by teachers is `response_attachments`.

To understand how a step is being added we debugged into the code execution for various scenarios such as student uploading a file, student submitting a quiz attempt, and teacher giving feedback. Below are the functions used in creation of step:

- `question_usage_by_activity::process_action` – It takes question slot and an array of key-value pairs, `$submitteddata` as arguments. This function adds a step to the question attempt corresponding to the question slot passed. The step data for the newly added step will be taken from `$submitteddata`.

- `question_engine::save_questions_usage_by_activity` – The argument for this function is `question_usage_by_activity`. This function saves it to the database, thereby saving our new step.

- `question_engine::load_questions_usage_by_activity` – It takes `usage_id` as argument, and loads and returns the corresponding `question_usage_by_activity`.

5.2 Backup logic and implementation

In this section we first discuss the logic used to mark annotated files to be backed up and then the implementation details.

5.2.1 Backup logic

While backing up a quiz, every question attempt gets backed up along with corresponding file attachments by the student. A file can be uniquely identifed by `filearea`, `contextid`, and `itemid`. We found that the `itemid` of the file attachments by student is same as the `attemptstepid` of the question attempt step. When a question attempt is backed up, all the corresponding question attempt steps also get backed up. During that time, all files in the filearea `response_attachments` with `item_id = attemptstepid` are also marked for backup.

The snippet below is taken from the function `process_nested_element()` responsible for actually backing up the file attachment.

```
foreach ($area as $filearea => $info) {
        $contextid = !is_null($info->contextid)
                         ? $info->contextid
                         : $this->get_var(backup::VAR_CONTEXTID);
        $itemid    = !is_null($info->element)
                         ? $info->element->get_value()
                         : null;
        backup_structure_dbops::annotate_files($backupid,
                         $contextid, $component, $filearea,
                         $itemid, $this->progress);
}
```

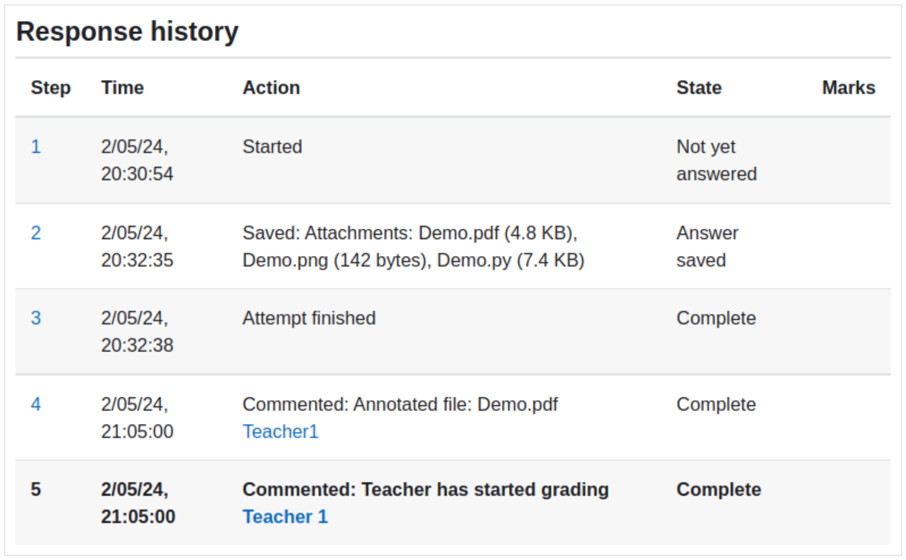
In this code, `$info` is an `stdClass` that has two properties – `contextid` and `element`. The `element` stores the `attemptstepid`. The first two lines inside the `foreach` loop is responsible for fetching the `contextid` and `itemid` of the file to be backed up. The third line adds backup id database record for all files in the given file area with the given `contextid` and `itemid`, thereby marking all these files to be backed up.

5.2.2 Implementation

On saving of our annotated file, as discussed in Section 5.1, we need to add a step to the corresponding question attempt. The saving is handled by `upload.php` and hence the logic for adding the step is to be added here. Each time the annotation is saved we add the step. Let us call this step, an annotation step.

We loaded the `question_usage_by_activity`, `$quba` using the `$usage_id` which we get as URL param for `upload.php`. We defined `$submitteddata` to be a one element array with the key as `"-comment"` and value as `"Annotated file:  <filename>"`. Then we called `question_usage_by_activity::process_action` on `$quba` to create the step. Then we save it to the database. Then we get the attempt step id of first annotation step from `$quba`, which will be the `$itemid` for the annotated file to be saved. This ensures the backing up of the annotated file.

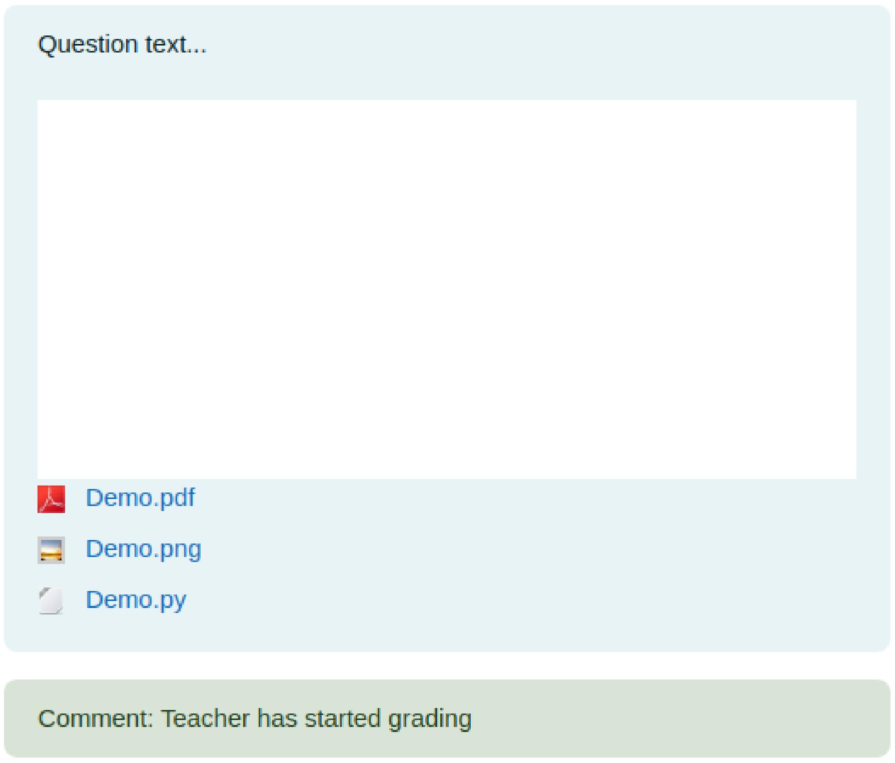
All `question_attempt_steps` are shown to the teacher in the Response History table of the question attempt. The annotation step for each annotation performed by the teacher will be added to the table. The last question attempt step with `-comment` in its data will be shown to the student as teacher's comment. So to hide this from the student we added another step with `-comment`, `"Teacher has started grading"` (Fig. 5.1). If the teacher has already given some comment manually, that will be added after each annotation step instead of the default comment.



Response history

Step	Time	Action	State	Marks
1	2/05/24, 20:30:54	Started	Not yet answered	
2	2/05/24, 20:32:35	Saved: Attachments: Demo.pdf (4.8 KB), Demo.png (142 bytes), Demo.py (7.4 KB)	Answer saved	
3	2/05/24, 20:32:38	Attempt finished	Complete	
4	2/05/24, 21:05:00	Commented: Annotated file: Demo.pdf Teacher1	Complete	
5	**2/05/24, 21:05:00**	**Commented: Teacher has started grading Teacher 1**	**Complete**	

Figure 5.1 The Response History table after the first annotation by a teacher.



Question text...

Demo.pdf

Demo.png

Demo.py

Comment: Teacher has started grading

Figure 5.2 Student's view after the first annotation by a teacher.

Chapter 6

Aligning HTML, JavaScript, and CSS for Moodle Compatibility

In the past work all the JavaScript files were added using script tag in `index.html` file. The JavaScript files namely `pdfannotate.js` and `clickhandlers.js` were in vanilla JavaScript format and third party JavaScript files were added using script, referencing library files hosted on a Content Delivery Network (CDN). Similarly there were two CSS files added - `styles.css` and `pdfannotate.css` and third party CSS files like Bootstrap and Font Awesome were added in `index.html` using link tags with href pointing to library files hosted on CDN. However according to Moodle development guidelines such practices are not acceptable.

6.1 Converting HTML to Mustache template

Moodle makes use of the Mustache template system to render its HTML output [14]. It allows dynamic content rendering. Variables can be inserted directly into the template, and their values are substituted during rendering.

The `index.html` of the past work should be converted into a Mustache template format to ensure compatibility. This file should be present in the `templates/` directory of the plugin. We converted the `index.html` to `annotator.mustache` with HTML tags, Mustache tags, and placeholders. Placeholder values are given in `annotatorMustacheConfig.php`. We now render HTML from the template using

```
$output->render_from_template('qtype_essayannotate/annotator',$data);
```

where `$output` is the renderer of the plugin and `$data` is the `stdClass` containing values to the placeholders in the Mustache template.

6.2 Converting vanilla JavaScript to AMD format

Moodle development guidelines insists that all JavaScript used by plugin should be present in `amd/src` folder of the plugin and should be in ESM/AMD format where ESM stands for ECMAScript Module and AMD for Asynchronous Module Definition. Therefore, third-party JavaScript files referenced by CDN links need to be locally added to the `amd/src` folder and the JavaScript files `pdfannotate.js` and `clickhandlers.js` are to be converted to ESM/AMD format. Since a third-party library FabricJS is not available in ESM format we decided to stick with the AMD format.

6.2.1 JavaScript in AMD format

Syntax of JavaScript code in AMD format looks like this [15] -

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

FabricJS, PDFjs, and JQuery are the dependencies of `pdfannotate.js`, which in AMD format will look like:

```
define(
    [‘qtype_essayannotate/fabric’, ‘qtype_essayannotate/pdf’, ‘jquery’],
    function(fabric, pdfjsLib, $) {
        /* JavaScript code of past work */
        return {
            init:function(/*parameters to initialize*/){
                /*Initializing variables*/
            }
        }
    }
);
```

6.2.2 Including from PHP

The following function call is used to include JavaScript from PHP:

```
    $PAGE->requires->js_call_amd(‘qtype_essayannotate/pdfannotate’,
                                ‘init’,[/* params to init function*/]);
```

For this call to happen it is required to add header and footer to the page along with the output of the Mustache template, as shown below:

```
     echo $output->header();
     echo $output->render_from_template(...);
     echo $output->footer();
```

In production environment Moodle serves the JavaScript files from `amd/build` directory which consists of the minified files. Minification is a process of reducing the size of JavaScript files by removing unnecessary characters such as comments, whitespace, and newlines, without affecting functionality. This practice significantly reduces file sizes, leading to faster load times and improved performance, especially in web applications with large JavaScript codebases.

To minifiy the files present in `amd/src`, Moodle suggests using `grunt` [16]. Grunt will also perform linting of JavaScript files.

6.3 CSS

All the CSS required for the plugin should be present in `styles.css` file. All `styles.css` files from all plugins are concatenated, cached, and served as a one big resource on all pages of the given Moodle installation [17]. Since there are two CSS files, we merged them and removed third party CSS as they are already used in core Moodle. As CSS from every plugin is concatenated the CSS selectors in `styles.css` should be properly namespaced so that it will not affect other pages. For example, `.messagebox` will become `.qtype_essayannotate_messagebox`.

In the past work CSS was added to the `index.html` using `<link>` tags, now we don't have to include it as Moodle automatically takes all `styles.css` from the root directory of plugin.

Chapter 7

Testing the Plugin

In this chapter, we discuss how we used Behat for automated testing of the functionalities of the plugin, and how we manually tested various scenarios.

7.1 Automated testing using Behat

Behat [5] is a popular behavior-driven development framework for PHP. It allows developers to write human-readable acceptance tests in a language called Gherkin. Behat takes a set of Features, Scenarios, and Steps, and uses these to step through actions and test results using a real web browser. The details of our test cases for Behat testing can be found in `tests/behat` directory of our repository [18].

7.1.1 Setting up Behat for Moodle

The following are required for Behat testing:

- Any recent OS with supported version of Moodle installed.
- A recent web browser, we use Chrome and Firefox.
- A WebDriver implemented for the browser being used, which are chromedriver and geckodriver respectively.

- A new dataroot area has to be created for the use of Behat.
- The path to dataroot, wwwroot, and the prefix for database tables used by behat has to be set in `config.php`. Our setup is configured as follows:

```
$CFG->behat_wwwroot = ‘http://127.0.0.1’;

$CFG->behat_prefix = ‘bht_’;

$CFG->behat_dataroot = ‘/var/moodledata/behat_dataroot’;
```

- After setting the configurations, Behat can be initialized by using

```
php admin/tool/behat/cli/init.php
```

The acceptance test site can be found at http://127.0.0.1 and the production site can be found at http://localhost.

7.1.2 Writing acceptance tests

Behat tests are located in `tests/behat` directory of the plugin. In Behat, each test is represented using a feature file. Feature files are written in Gherkin syntax. Each feature file represents a specific functionality or feature of the plugin being tested and contains multiple scenarios defined using `Given`, `When`, and `Then` steps to describe the preconditions, actions, and expected outcomes of the behavior being tested. The following is an example feature file with two scenarios for testing user authentication [19]:

```
Feature: User Authentication


Scenario: Successful login
  Given I am on the Moodle login page
```

```
    When I enter valid credentials

    And I click the login button

    Then I should be logged in successfully


  Scenario: Failed login with invalid credentials

    Given I am on the Moodle login page

    When I enter invalid credentials

    And I click the login button

    Then I should see an error message
```

The code corresponding to the steps in the feature file are stored in files called step definitions. These files contain the actual PHP code that implements the behavior described in the feature files. That file for our plugin is `tests/behat/behat_qtype_essayannotate.php`. Step definition files are usually organized within a directory structure specified in the Behat configuration file (`behat.yml`). Below is an example code for a step:

```
/**

   @When /^I click the "(.*)" button$/

*/

public function i_click_on_the_button($button) {

  $this->getSession()->getPage()->pressButton($button);

}
```

In Behat, step definitions are linked to the steps in feature files using regular expressions. The step definition is annotated with a regular expression within a comment block. Behat matches this regular expression against the steps in the feature files to determine which step definition to use.

7.1.3 Testing the PDF annotation feature using Behat

To test the PDF annotation feature of the plugin we added `.feature` files. The feature files have four scenarios:

- Student uploads a PDF file and teacher is able to annotate.
- Student uploads a PNG file and teacher is able to annotate.
- Student can see the annotated file only after the question is graded.
- When the quiz is backed up, the annotated files also gets backed up.

In these scenarios we also check if the annotation step and the step to mask it from student have been added to the response history.

The test fixtures include a PDF and a PNG file which are stored in the `tests/fixtures` folder. The step responsible for drawing annotation on the PDF is:

```
And I annotate the pdf
```

The code definition for this step is:

```
/**
 * @When I annotate the pdf
 */
public function i_annotate_the_pdf(){

    $js = "
    var pathString = ‘M 100 100 Q 125 125 150 150 Q 175 175 200 200
                      L 200 200’;

    var path = new fabric.Path(pathString, {
        fill: ‘’,
            stroke: ‘black’,
            strokeWidth: 2,
            strokeLineCap: ‘round’,
            strokeLineJoin: ‘round’
        });
        pdf.fabricObjects[0]._objects.push(path)
        pdf.fabricObjects[0].renderAll()
        ";
        $this->execute_script($js);
        sleep(3);
    }
```

The `pdf` in the above code is object of `PDFAnnotate` and is created in `clickhandler.js`. `pdf.fabricObjects` is an array containing FabricJS objects representing elements of the PDF document. The path representing the annotation is created using FabricJS and pushed into the array and `renderAll()` method is called to render the changes made to the canvas.

7.2 Unit testing

We made modifications to the unit tests originally written for `essay`, so that they work for our plugin. We are planning to add additional test cases covering more scenarios.

7.3 Testing scenarios manually

7.3.1 Incorrect paths to ImageMagick and Ghostscript

If ImageMagick is not installed or incorrect path is given in settings page of the plugin, when teacher tries to annotate a file with mime type `image` or `text`, an exception is thrown. Similarly for Ghostscript when teacher tries to annotate a PDF file with version greater than 1.4, saving will not be successful.

7.3.2 Unsupported file types

If student uploads a file with mime type `pdf` but its extension is not `.pdf` an exception (unsupported file) will be thrown. Similarly if extension of the file is `.pdf` but the mime type is not `pdf` the same exception will be thrown. For non PDF files with mime type other than `text` or `image`, the same exception will be thrown.

7.3.3 File size near permissible limit

Permissible limit is the minimum of `upload_max_filesize`, `post_max_size`, `memory_limit` (defined in `php.ini` file), and `$CFG->maxbytes` (defined in `config.php`). If the size of file exceeds this limit after annotation, teacher will not be able to save the file.

7.3.4 Working of tools

We tested the UI tools used for annotation, namely freehand drawing, highlight box, and text annotation, to ensure their functionality. We verified that all objects created within the current annotation window are selectable and deletable using the delete tool.

7.3.5 Working of plugin in multiple PHP and Moodle versions

We created test Moodle servers with Moodle versions 4.0 to 4.4 and PHP versions 7.4 and 8.1. One of the test server uses PostgreSQL for database and others use MySQL. This was done to ensure that our plugin is cross-db compatible. It is listed as an approval blocker in the Moodle plugin contribution checklist [9].

We installed the plugin in these servers following standard installation procedure of making a ZIP file of the code repository and uploading it as admin in the Site administration → Plugins → Install plugins. We installed the dependencies in the servers and gave paths to their executables in the settings page of the plugin. We created test users, courses, and quizzes with `essayannotate` questions and tested working of the plugin across various scenarios.

Chapter 8

Bug Fixes

In this chapter, we discuss the fixes for bugs encountered during Behat and manual testing.

8.1 Masked exception messages

Earlier, variable names like `$attemptid` were included in some exception messages. This is a vulnerability as it reveals unnecessary information to possibly malicious users. So we removed such unnecessary information from the exception messages.

8.2 `$attemptid` being wrongly set

In the production environment, `$attemptid` is same as `$qa->get_usage_id()`. While running of Behat tests, we found that this is not the case in the testing environment. We fixed it by getting the `$attemptid` directly from the URL parameters.

8.3 `EssayPDF` directory

Earlier, we wanted the admin to manually create the `EssayPDF` directory with necessary permission in the file system. But during running of Behat tests, we found that after each test, the environment is reset and the `EssayPDF` in the Behat's dataroot was getting deleted. We fixed it by creating the directory, if it does not already exist, using a PHP function `mkdir()`.

8.4 Attachments with the same file name for different questions

While manually testing, we found unexpected behaviour when two possibly different files with the same file name are uploaded as attachments to different questions in the same quiz. When one of the file is annotated, the other file with same file name was also being shown as annotated. This was because both of them were pointing to the same annotated file record in the Moodle file system.

We first tried to fix it by appending question number to the uploaded file name so as to make the file names distinct thereby creating two different file records.

We found a more general fix by adding question `usage_id` and `slot_id` to the file info array used to create the file record of the corresponding annotated file.

8.5 Message displayed on clicking save after annotation

Earlier we were showing a JavaScript window alert saying whether the save operation has been successful or not. We couldn't find a way to capture this in Behat testing. So we changed the alert to a custom function `showMessage`. This function displays the message on the screen using HTML `<p>` tag which disappears after 3 seconds.

8.6 Concurrent annotation of question attempts

When more than one student-attached file was being annotated concurrently by one or more teachers, it led to unexpected behavior. The teacher's annotation on the current file would be saved with the latest file opened for annotation, appearing as a background. This issue occurred because the past work used a single dummy file - a copy of the attached file in the `EssayPDF` directory of the `moodledata` folder. The annotations were added on top of this file while saving the annotation. This dummy file would be overwritten every time a new file was opened for annotation.

To ensure that every file opened for annotation would have a unique dummy file, we decided to include the attempt id of the quiz, the question number (slot), and the user id in the name of the dummy file. For example, if the attempt id is 1, the slot is 2, and the user id is 3, `dummy.pdf` would become `dummy1$2$3.pdf`.

When multiple annotating sessions of the same file occur concurrently, by one or more teachers, the latest saved annotations will overwrite all previously saved annotations in the same session.

8.7 Incorrect mime type

During manual testing, it was observed that when the extension of a ZIP file is modified to `.pdf` and uploaded, no error is thrown, but the Annotator UI remains blank. This was because `annotator.php` was using the file extension to determine if the file is a PDF or not. When the extension of a non-PDF file was modified to `.pdf` and uploaded, it was considered as a PDF, and the plugin attempted to render it in the Annotator UI. However, PDF.js, the library responsible for rendering the file in the UI, expects the file to be a PDF. Since the actual file was not a PDF, nothing was rendered for annotation. To fix this issue, we decided to use the PHP function `mime_content_type()`, instead of relying on the file extension, to get the mime type of the file.

Also, when the extension of an image or text was modified to `.pdf` and uploaded, annotation was permitted. However, the saved annotated file would not get listed in the Corrected Documents section. This is because another function `get_mime_type_description()` is used in `renderer.php` when listing the annotated files. The mime type is needed for this because if the mime type of the attached file is not "PDF" but is "image" or "text", the filename is appended with `"_to_pdf"`. To retrieve the file back from the file system in `renderer.php`, we need to know the correct filename. The `get_mimetype_description()`

function uses the `mimetype` field present in the `mdl_files` table, which is derived from the file extension and has nothing to do with the contents of the file. Hence, if we allow image or text files with the `.pdf` extension to be annotated, the renderer will look for the annotated file with the same filename, whereas the annotated file would have been saved with filename appended with `"_to_pdf"`. The PHP function `mime_content_type()` can not be used in renderer to get the mime type because this function requires the file, whose mime type is queried for, to be in the file system, and saving and deleting a file solely for the purpose of getting the correct mime type is an overhead that should be avoided in a renderer.

To resolve this issue, we implemented the following solution: If there is an inconsistency between the mime type and extension when either of them indicates the file is a PDF, we reject the file for annotation. Now, both the mime type and extension should indicate that the file is a PDF for it to be considered as a PDF file. Files with the mime type "image" or "text" will be converted to PDF using ImageMagick and are then allowed to proceed. Files with any other mime type will not be supported.

8.8 Filenames containing whitespace

It was observed that image files with whitespace in the filename were not rendered in the Annotator UI. This was because the existing work uses `shell_exec` with this command to convert it to PDF format:

```
convert <srcfilename> <destfilename>
```

This will not work as intended if the filename contains whitespace, as the shell will treat it as multiple arguments instead of a single argument. To fix this issue, we embedded the file names in single quotes, like this:

```
convert '<srcfilename>' '<destfilename>'
```

By enclosing the filenames with single quotes, they are treated as a single argument, ensuring that files with whitespace in their names are handled correctly.

8.9 Images with non-standard resolutions getting cropped after annotation

After annotating an uploaded image of dimension 963×1280 pixels we observed that the annotated PDF was trimmed to resolution 595×841 (A4). To fix this, we passed an argument (`-page a4`) in the `convert` command so that the converted PDF is scaled to A4 resolution.

8.10 PDF thumbnail in corrected documents

We found that the link for the PDF thumbnail in the corrected documents section was broken. This issue occurred because the absolute path to the thumbnail was hardcoded. We fixed this by using the `pix_icon()` function of the core renderer, which generates the correct path to the thumbnail dynamically.

8.11 Choice of secondary button

Moodle documentation states that secondary buttons should be used for buttons that are always visible, and there should be only one primary button in each UI component [20]. Hence, we chose to make our Annotate button a secondary button to adhere to Moodle's user interface guidelines.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

We completed the development of a standalone Moodle plugin for annotation of files uploaded as attachments by a student in a quiz, extending the past work [2, 3]. Now, like any other Moodle plugin, our plugin can be downloaded as a ZIP file and installed by the admin. The code and installation steps are made available in this repository:

`https://github.com/vmsreeram/moodle-qtype_essayannotate`

An important contribution in the present work is the development and implementation of backup and restore functionalities for the annotated files. Security concerns raised by the Moodle developer community about the past work were addressed. Feedback comments from users of the tool were taken into consideration during the development.

The guidelines and coding style for Moodle plugin development were adhered to. We tested the functionality of the plugin, both manually and using automated tests, for various scenarios. Thorough code review was done and the plugin will be submitted to Moodle plugin directory shortly.

9.2 Future work

While this plugin for quiz annotations evolved into its present form, we could identify scope for further improvements to its capabilities as well as new functionalities that would increase its usefulness. However, considering the time and resource constraints in this project, we had to limit the focus of our efforts towards completing the development of the plugin as per the initially set objectives.

An important enhancement in functionality would be the capability to edit past annotations. In the current work, the base PDF file and the annotation layer are combined into a new PDF file while saving it. If a teacher wants to annotate the file in a later session, new annotations can be done only on top of this and annotations from earlier sessions cannot be edited. A solution could be to save the base file and the annotations separately, so that annotations from previous sessions could be edited.

The assignment module is an activity module in Moodle, similar to the quiz module. There is a builtin annotator in the assignment module, but built using YUI JavaScript library which is deprecated in Moodle [21]. Thus there is a need for a new annotation tool. With some effort, it should be possible to reuse existing code and take an approach similar to ours to release an assignment feedback plugin.

Bibliography

[1] Moodle Pty Ltd. Moodle open source learning platform. URL https://moodle.org. Accessed 01-Oct-2023.

[2] Tausif Iqbal and Vishal Rao. Quiz annotator plugin that enables teacher to annotate essay type question in moodle. 2021. URL https://github.com/TausifIqbal/moodle_quiz_annotator. Accessed 01-Oct-2023.

[3] Parvathy S. Kumar and Asha Jose. Pdf annotator for moodle quiz. 2023. URL https://github.com/Parvathy-S-Kumar/Moodle_Quiz_PDF_Annotator. Accessed 01-Oct-2023.

[4] Moodle tracker. Apply pdf annotation for grading essay or written response quiz questions. URL https://tracker.moodle.org/browse/MDL-65872. Accessed 01-Oct-2023.

[5] Moodle Pty Ltd. Running acceptance tests. URL https://moodledev.io/general/development/tools/behat/running. Accessed 10-March-2024.

[6] Derick Rethans. Xdebug - debugger and profiler tool for php. URL https://xdebug.org. Accessed 01-Oct-2023.

[7] Moodle Developer Resources. Common files, . URL https://moodledev.io/docs/apis/commonfiles. Accessed 01-Oct-2023.

[8] Gordon Bateson. The essay (autograde) question type. URL https://github.com/gbateson/moodle-qtype_essayautograde. Accessed 07-Nov-2023.

[9] Moodle Developer Resources and Documentation. Plugin contribution checklist. URL https://moodledev.io/general/community/plugincontribution/checklist. Accessed 10-March-2024.

[10] Lafuente et al. Code-checker. URL https://moodle.org/plugins/local_codechecker. Accessed 03-May-2024.

[11] Moodle Pty Ltd. Plugin with third party libraries, . URL https://moodledev.io/general/community/plugincontribution/thirdpartylibraries. Accessed 03-May-2024.

[12] Moodle Developer Resources. Backup 2.0 for developers, . URL https://docs.moodle.org/dev/Backup_2.0_for_developers. Accessed 10-March-2024.

[13] Moodle Developer Resources. Backup API, . URL https://moodledev.io/docs/apis/subsystems/backup. Accessed 10-March-2024.

[14] Moodle Developer Resources. Templates, . URL https://moodledev.io/docs/guides/templates. Accessed 10-March-2024.

[15] Addy Osmani. Writing Modular JavaScript With AMD, CommonJS and ES Harmony. URL https://addyosmani.com/writing-modular-js/. Accessed 03-May-2024.

[16] Moodle Pty Ltd. NodeJS and Grunt, . URL https://moodledev.io/general/development/tools/nodejs. Accessed 03-May-2024.

[17] Moodle Pty Ltd. CSS Styles, . URL https://moodledev.io/general/community/plugincontribution/checklist#css-styles. Accessed 03-May-2024.

[18] Nideesh N and VM Sreeram. Essay annotate. URL https://github.com/vmsreeram/moodle-qtype_essayannotate. Accessed 04-May-2024.

[19] Anshita Bhasin. Writing Feature Files with Cucumber: A Beginner's Guide. URL https://medium.com/@anshita.bhasin/writing-feature-files-with-cucumber-a-beginners-guide-d3748aadd00c. Accessed 04-May-2024.

[20] Moodle Pty Ltd. UI Component Library. URL https://componentlibrary.moodle.com/admin/tool/componentlibrary/docspage.php/moodle/components/buttons. Accessed 03-May-2024.

[21] Moodle Pty Ltd. Improve Annotation - Moodle docs. URL https://docs.moodle.org/dev/Improve_Annotation. Accessed 04-May-2024.