Design, Development, and Deployment of TA Buddy: An AI-Assisted Autograding System for Programming Assignments

Thesis to be submitted in partial fulfillment of the requirements for the degree

of

Master of Technology

by

Saurav Chaudhary
23M0838

Under the guidance of

Prof. Varsha Apte

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
India - 400076

CERTIFICATE

This is to certify that we have examined the thesis entitled **Design, Development, and Deployment of TA Buddy: An AI-Assisted Autograding System for Programming Assignments**, submitted by **Saurav Chaudhary**(Roll Number: *23M0838*) a postgraduate student of **Department of Computer Science and Engineering, IIT Bombay** in partial fulfillment for the award of degree of Master of Technology. We hereby accord our approval of it as a study carried out and presented in a manner required for its acceptance in partial fulfillment for the Post Graduate Degree for which it has been submitted. The thesis has fulfilled all the requirements as per the regulations of the Institute and has reached the standard needed for submission.

Prof. Varsha Apte

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Place: IIT Bombay
Date: 24 October,2024

ACKNOWLEDGEMENTS

I would like to extend my deepest gratitude to **Prof. Varsha Apte**, Department of Computer Science and Engineering, IIT Bombay, for her consistent guidance and support throughout this Phase 1 of my MTP. Her consistent availability, willingness to help, and insightful feedback were invaluable. Despite the multiple setbacks encountered along the way, her encouragement always pushed me forward, which has been crucial in the progress made so far. The valuable discussions and her motivation to explore new ideas have greatly shaped the direction of this work.

I would also like to express my heartfelt thanks to Naga Kalyani Goda, my teammate, for her dedication and support throughout this phase. I am also grateful to Chaitanya Shinge and Yogesh Mandlik for their invaluable assistance in various aspects of the project.

Saurav Chaudhary
IIT Bombay
Date: 24 October,2024

ABSTRACT

The rise of MOOCs post-pandemic has emphasized the need for efficient grading methods, especially for programming assignments in large classes. Traditional auto-graders rely on test-case-based grading, which compares a student's code output to a reference output, but this approach falls short in academic settings. In educational contexts, it's essential to inspect the source code to award partial marks for partially correct code and assess code quality based on specific criteria. However, manual code inspection becomes time-consuming in large enrollments, making advanced autograding tools necessary.

This project aims to develop an an AI Assistant for source code based grading on *EvalPro* website. Evalpro, an integral component of the Bodhitree e-learning platform, is dedicated to managing programming assignments. The goal is to integrate this AI Assistant which we have named **TA Buddy** into Evalpro for criterion based grading of programming assignments. TA Buddy is powered by Code LLama, a large language model especially trained for code related tasks, which we fine-tuned using a graded programs dataset. Given a problem statement, student code submissions and a grading rubric, TA Buddy can be asked to generate suggested grades, i.e. ratings for the various rubric criteria, for each submission. These AI-suggested grades streamline the grading process, allowing human TAs to either accept or modify the grades, thereby enhancing overall grading efficiency while maintaining the quality of evaluations.

To successfully integrate TA Buddy into the BodhiTree platform, we designed, developed, and deployed a comprehensive AI-driven web interface that streamlines the process of code evaluation, data collection, and continuous model improvement(retraining). The system relies on three pipelines: Inference, Data Collection, and Retraining. Furthermore, we will discuss the field trials conducted to validate TA Buddy-1.0 and review user feedback and results to confirm its effectiveness.

Keywords: TA Buddy, Inference, Data Collector, ReTraining, EvalPro

Contents

List of Figures

Chapter 1

Introduction

In programming courses, student learning is usually evaluated through programming assignments [1].Programming assignments play a crucial role in programming courses. It provides hands-on experience, helps tutor to assess students, provides better understanding of subject matter and allows one deep dive into technical concepts. The instructor gives a programming problem to solve, with some sample test cases to help students understand how the program should work. Each test case is a pair of input with its expected output.

For large classes, many programming courses use autograders, which are tools that automate the process of checking if the student's program works correctly [2]. These autograders compile and run the program, then compare its output for each test case with the expected result. This basic process of testcase-based autograding is shown in Figure 1.1.

A large number of programming autograders are available for instructors to use, and many institutions build their own custom autograders that implement this basic functionality, with minor variations in features [3, 4].But most of the automated grading systems are primitive and test-case based. These primitive autograders are biased and inefficient in educational contexts, instructors may require specific assessments, like evaluating algorithms, checking the use of certain data structures, distinguishing between iterative and recursive implementations, and avoiding reliance on particular packages or built-in functions. We term these instructions as *coding criteria*. These are given by instructors primarily so that a specific learning goal of programming can be evaluated. For *e.g.*, the aim may be to implement functionality using a specific algorithm - *e.g.*, finding prime numbers using the Sieve of Eratosthenes, sorting using

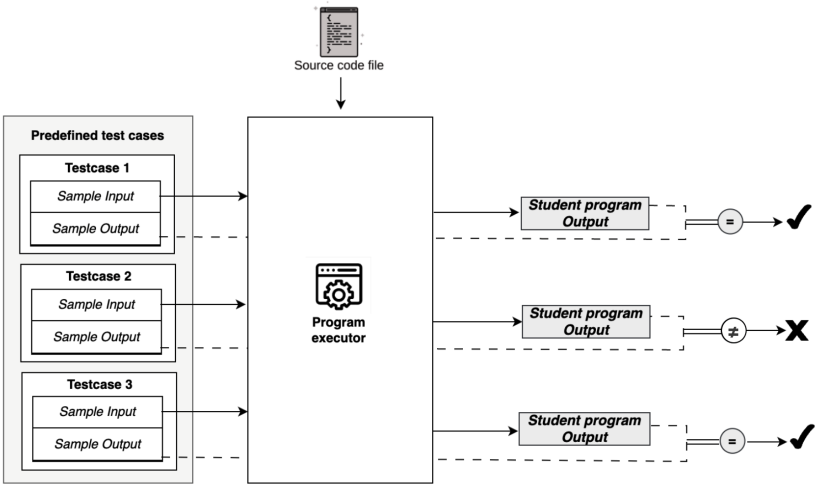

Figure 1.1: Overview of Test-case based evaluation for a student submitted source code

insertion sort, and so on. There may also be quality criteria such as the time complexity of the algorithm implemented, or whether functions have been used judiciously, or whether the code is well-commented, which may not be written explicitly in the problem statement, but appear in the evaluation guidelines. Such code quality or code criteria-based evaluation has to be done by *teaching assistants (TAs)*.Typically testcase-based autograding is used for the functionality check, and TAs subsequently look at the source code and grade it according to the instructor's grading guidelines.

An example to demonstrate the ineffectiveness of test-cased based grader for academics is shown in Figure 1.2. In Figure 1.2, there are two codes shown. The code on the left is correct implementation of bubble sort, and the code on the right is incorrect. But the test-case based autograder fails to identify it.

The most systematic way to perform this is using a *grading rubric* [5]. In Table 2.1, we present a rubric for the problem statement shown in Figure **??**. A rubric has multiple *criteria* (in rows), where each criterion has a range of *ratings*. The ratings are ordered from highest (meets the criterion fully) to lowest (does not meet the criterion at all). Here, 'A' is used to indicate the best rating. TAs are expected to select one of the ratings, and optionally explain why those ratings were chosen. The ratings typically correspond to some marks and are added up to arrive at the total code quality and criteria-based marks [6]. How these marks combine with the testcase marks is a policy that may vary with the instructor. Thus source code correctness is

Write a C++ program to implement bubble sort algorithm. You need to write bubble sort within function void bubble_Sort(int a[], int n).

```
void bubble_sort (int a[], int n)
{
  int i, j, temp, flag = 1;
  for (i = 0; i < n - 1 && flag == 1; i++){
     flag = 0;
      for (j = 0; j < n - 1 - i; j++){
          if (a[j + 1] < a[j]){
              flag = 1;
              temp = a[j + 1];
              a[j + 1] = a[j];
              a[j] = temp;
            }
       }
    }
}
```

```
void bubble_sort(int a[],int n)
{
    int i,j,k;
    for(i=1;i<n;i++){
        k=a[i];
      for(j=i-1;j>=0&&a[j]>k;j--){
       a[j+1]=a[j];
      }
      a[j+1]=k;
    }
}
```

Figure 1.2: Test-case based assessment.

a default rubric criterion that may appear in many code grading rubrics with a range of ratings.

There is another reason that source code inspection is of critical importance in an educational setting. That is to award *partial marks* to students who may have made a small mistake in their program, which results in the program failing some or all testcases. Figure 1.3 shows one such example. In an educational setting, exclusive use of a testcase-based autograder is often not desirable. If a student's submission fails due to a small output formatting mistake, or even a small logical mistake, an instructor may want to give some marks so as to not demotivate the student. Thus source code correctness is a default rubric criterion that may appear in many code grading rubrics with a range of ratings. So we use approaches based on Artificial Intelligence (AI) to auto-grade source code submitted by a student, according to a grading rubric supplied by the instructor. Specifically, our intelligent task for our AI model is as follows:

Given a programming problem statement, p, a student program submission s_p*, and a grading rubric* g_p*, comprising of* m_p *criteria* $\{C_{p1}, C_{p2}, \ldots, C_{pm_p}\}$*, where a criterion* C_{pi} *has* n_{pi} *ratings* $R_{pi1}, R_{pi2}, \ldots R_{pin_{pi}}$*; select a rating* $R_{pij}, 1 \leq j \leq n_{pi}$ *for all criteria* $C_i, i = 1, 2, \ldots, m_p$ *which reflects the most appropriate rating in that criterion for the*

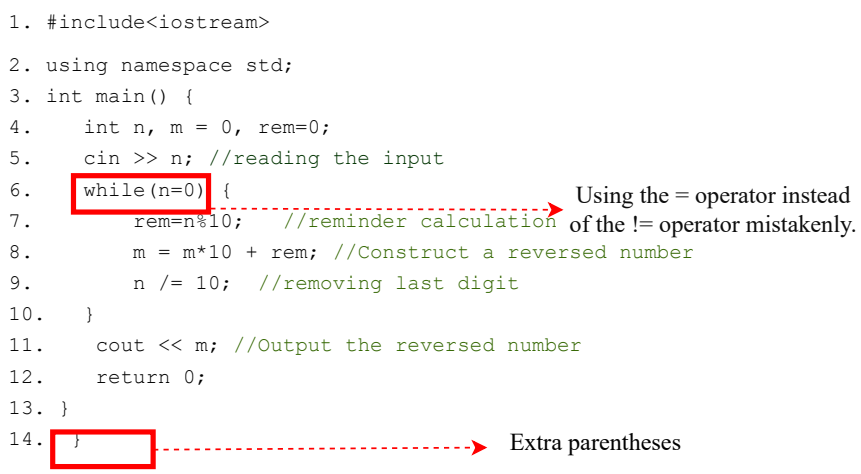

Figure 1.3: Student code with common syntax errors, instructor may consider giving partial grade even though the program will fail all testcases.

student submission s_p.

We prompt the fine-tuned CodeLlama-7B model with the grading task, and it automatically selects the most appropriate rating for each criterion in the rubric. However, manually processing all submissions (i.e., downloading them from the BodhiTree interface), exporting them, and running inference is suitable for R&D purposes but is too complex and time-consuming for real-time use.

In this thesis, we will discuss the development and integration of TA-Buddy, an AI-assisted grading tool, into the BodhiTree, marking its transition to TA-Buddy 1.0. Our primary focus is to explore how TA-Buddy enhances the grading process by automating the selection of rubric ratings for student submissions. Additionally, we will examine the design and implementation of the data collection and automated retraining pipelines, which aim to continuously improve the AI model's accuracy. We will also discuss the steps taken towards automating model selection post-retraining and outline the future development of TA-Buddy 2.0, where further automation and feature enhancements are planned to enhance grading efficiency. Lastly, we will present the user feedback of TA-Buddy contribution to streamlining the grading workflow and improving the TA experience while grading on BodhiTree platform.

1.0.1 TA Buddy in EvalPro

This involves developing a comprehensive AI-driven backend system that streamlines the process of code evaluation, data collection, and continuous model improvement, integrated with BodhiTree-Evalpro for automated grading of programming assignments. Our primary objective was to robustly integrate this system, TA-Buddy, into the existing BodhiTree-EvalPro platform for the automated grading of programming assignments.

BodhiTree, currently supports conventional test-case-based autograding alongside a manual grading system. Instructors could specify grading rubrics consisting of various criteria and ratings, which Grader TAs would use to manually evaluate student submissions by selecting ratings and providing comments for each criterion. Our task was to enhance this process by seamlessly integrating TA-Buddy, an AI-powered assistant, into BodhiTree which automates the grading process by suggesting ratings and reasoning for each criterion. The Grader TA can then either select the same rating and reasoning or select a different rating and write their own reasoning. They can also select the suggested rating but overwrite the reasoning.

Furthermore, we implemented a robust data collection pipeline to export graded submissions and feedback, ensuring that TA-Buddy can be retrained to adapt to new data points. This continuous retraining mechanism helps prevent model drift, keeping the grading system accurate and up-to-date with evolving student code patterns.

1.0.2 Contribution and Achievements in MTP Phase 1

As part of this MTP, We integrated the AI-Assisted grading tool, TA-Buddy, into the BodhiTree Interface, which streamlines the grading process by allowing instructors to auto-grade source code using AI with a single click. In addition, a manual data collection pipeline was set up to gather and store data from the BodhiTree interface to continuously improve the AI model. To enhance this process, an automated retraining pipeline was implemented on the TA-Buddy server, which checks for new data points at configurable intervals and initiates a retraining job when the data exceeds a set threshold. While model retraining is automated, switching to the best model for inference is still manual and planned for future development. This current version is marked as TA-Buddy 1.0, with full automation and further enhancements planned for TA-Buddy 2.0.

1.0.3 Report Outline

We will begin with Chapter 2, where we will describe all the major terms and concepts that will be used throughout the thesis. In Chapter 3, we will review related work and the literature around autograding systems and machine learning pipelines, which form the foundation for the development of TA Buddy. Chapter 5 will dive into the design and architecture of TA Buddy, focusing on the high-level architecture and then we will explore the three pipelines which are the heart of TA Buddy in depth.

In Chapter 6, we will describe the implementation details of various layer such as Database, Communication and Service Layer of Bodhitree and AI Server. Then we will look at Celery task which makes our task asynchronous. Chapter 7 will highlight the new features added to EvalPro in 2024, along with the AI4Code project and will also cover the bug fixes and improvements made to EvalPro.

In Chapter 8, we will present the experiments done under AI4Code project, and will explain the DPO and Dora Fine Tuning results. Chapter 9 will provide details on the field trials conducted to test TA Buddy in real-world scenarios, including the results, user feedback, and lessons learned. Finally, in Chapter 10, we will conclude the report by summarizing the key achievements and outlining future work for further improvements and research.

Chapter 2

Background

We begin with BodhiTree, a robust learning management system (LMS) that already offers numerous features (2.1). Among it the essential module is EvalPro, a programming autograder that supports programming activities and enables testcase grading of assignments (2.2). This platform not only provides automated grading but also provides subjective rubric-based manual grading through a well-designed user interface (2.3). With these capabilities, we decided to explore code-related LLMs for the subjective autograding of programming assignments (2.4). Our initial attempts at subjective grading using prompts yielded results that were average (2.5), But often resulting in outputs that were either poorly formatted or simply repeated the same problem statement (2.5.1). In search of improvement, we turned to fine-tuning these LLMs to align them with grading specific tasks (2.6). This process involved techniques such as Lora and DPO, which helped to refine the models (2.6.1, 2.6.2). Encouraged by the promising results from fine-tuning, we saw an opportunity to integrate an AI-assisted grader as a feature in EvalPro (2.6.3). To facilitate this integration seamlessly, we utilized the Django REST Framework (DRF), paving the way for a more efficient and intelligent grading process (2.7).

2.1 BodhiTree: A Learning Management System

BodhiTree[7] is an online learning platform created at IIT Bombay, aiming to make quality technical education accessible to everyone through personalized, flexible, and hands-on learning.

BodhiTree offers interactive multimedia books, including lab manuals that simulate classroom teaching. Each book contains chapters filled with interactive videos

(which include pop-up quizzes), auto-graded practice problems, reference materials, and slides. Students can access these books through a smartphone app or a web browser.

2.2 EvalPro : A Programming Autograder

In BodhiTree, we can host a course, and each course can have multiple labs that could be programming activities, virtual activities, or other types. EvalPro (Evaluation of Programming Assignment), is a module of BodhiTree which specially focuses on programming activity and it supports auto-evaluation of programming assignments. It supports multiple languages such as C,C++ and Python. It autogrades the assignment by running submitted codes on test cases provided by instructor (test-cased based) and rubric-based grading system that make use of an algorithm identification model for automated subjective evaluation.

2.3 Subjective Grading

Subjective grading is an evaluation method where the assessment of student work relies on the personal judgment and interpretation of the evaluator rather than strictly defined objective criteria. In the evaluation of programming assignments, subjective grading emphasizes that code submissions are assessed based on multiple criteria rather than solely relying on automated test cases. While automated tests can verify the correctness of a solution, they may not fully capture the nuances of code quality, such as clarity, organization, and adherence to coding standards. Evaluators often consider factors like the quality of comments and different coding criterion. Additionally, aspects such as algorithmic efficiency, the overall structure of the code play a crucial role in the assessment.

But the subjective nature of grading may introduces variability in score due to different interpretations by the TAs. So to have consistency in the grades, the widely used technique for subjective grading is via a rubric. Rubrics can help anchor the evaluation in more objective terms by specifying the detail in a structured and clear way.

Table 2.1: For Fig:2.4 Rubric for the problem *Finding all prime numbers from 2 to n*. Rating across different criteria are not comparable except for the fact that rating k is strictly greater than rating k+1 for a specific criterion.

Criteria	Rating-1 (A)	Rating-2 (B)	Rating-3 (C)
Creation of array of consecutive integers	*Correctly Done*	*Not Done*	
Multiples of p marked in array as non-prime	*Correctly Marked*	*Not Correctly Marked*	
Advance p to the non-marked number	*Correctly Advanced*	*Not Correctly Advanced*	
Comments	*Well Commented*	*Ok Commented*	*Poor or No comments*

2.3.1 Rubric

A rubric is a standard evaluation measurement scheme with clear criteria used to assess students' work consistently and fairly. A grading rubric is like a checklist that breaks down what's expected in an assignment and how it will be graded. It's helpful for teaching assistants because it makes grading easier and more consistent. Structure of a rubric is shown in Table **??**, where each row represents a criterion and column corresponding to each row represents rating.

```
Write a C++ program to find all prime numbers from 2 to n (n taken
 as input from the user).Your program should print out all prime
                      numbers from 2 to n.
```

Figure 2.1: Examples of Problem statements for Finding Prime Numbers with specific code requirements given by the instructor.

2.4 Code Related LLMs

CodeLLama proposed a family [2.2] of Large Language Model that outperforms general LLM for code specific tasks based on LLama 2 by enhancing the and increasing the capabilities of it by applying a cascade of training and fine tuning steps, which supports large input context and zero-shot instruction following ability for programming tasks. [8]

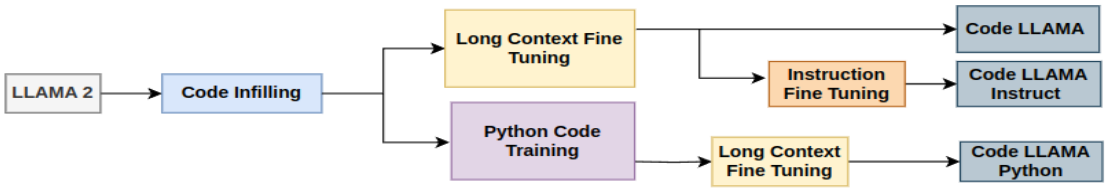

Figure 2.2: Family Of CodeLLama.

They performed **CodeTraining** on General Purpose Text and Code Data, **Code Infilling** which featured a multitask objective consisting of both Autoregressive and Causal Infilling Prediction, **Long Input Context** by modifying parameters of ROPE and **Instruction Fine-Tuning** to understand human instruction effectively.

The major focus of interest is CodeLLama-Instruct [2.3] because it has an extra fine-tuning step over dataset from LLama 2 and used the RLHF5 version. And during data annotation instead of using a forced choice they asked annotators to label the degree to which they prefer their chosen response over the alternative. And it was observed that it is better at following human instructions.

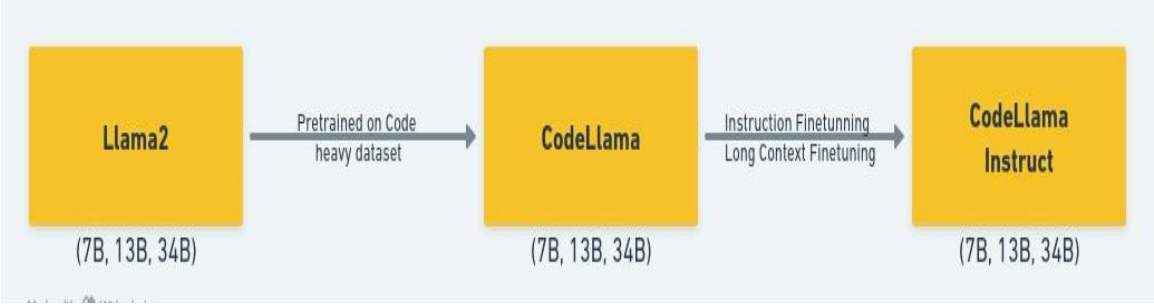

Figure 2.3: Illustration for PreTraining of CodeLLama-Instruct.

2.5 Using LLMs For Subjective Grading

As discussed in 2.3, Subjective grading can be very useful in delineating complex programming grading tasks into easier subtasks. To maintain consistency between grades, we can use a well-defined rubric. But still doing it manual is time-consuming.

To address this challenge, large language models (LLMs) can be employed to automate and assist with the grading process. A critical aspect of using LLMs effectively is crafting the right prompt. To get accurate and meaningful results from an LLM, one must create prompts that clearly define the task.

2.5.1 Prompt

In a general context, a "prompt" is simply an idea or signal given to someone to do something specific. For AI-specific purposes, a prompt is a natural language input or piece of information given to influence the behavior of a pre-trained model.

For our task, we provide LLM with two prompts and we termed them as System Prompt and Task-Specific Prompt.

2.5.1.1 System Prompt

This prompt is given at the start to define the LLM's role and align it with the overall task it needs to perform.

```
  Your task is to choose the MOST suitable option among a set of
options I provide, about a code which will also be provided.  Give
your output as a json with a single field "answer".  Do not output
 anything else.  Strictly follow this output format at any cost.
```

Figure 2.4: Example of System Prompt.

2.5.1.2 Task-Specific Prompt

This prompt is specific to a task and is provided after the system prompt.

2.6 Fine-Tuning

While prompts can be effective in guiding large language models (LLMs) to produce desired outputs, they do not always yield the expected results. In many cases, the answers provided by the model may lack accuracy or relevance. This is where fine-tuning comes into play. Fine-tuning is the process of taking a pre-trained model and training it further on a specific dataset for a particular task, allowing it to adapt and

```
### Context :
<Problem Statement>

### Code :
<Student Submission>

### Task :
Choose the option which is most suitable for the above code for the
criterion "Whether Babylonian method is implemented in a function".
Give your output as a json with a single field "answer". Do not output
anything else. Strictly follow this output format.

### Options :
A. Separate function is used and function signature is correct
B. Separate function is used but function signature is not correct
C. Totally wrong

### Response : The required output in json format is
```

Figure 2.5: Example of Task Specific Prompt

perform that task efficiently. The most common approach to fine-tuning is supervised fine-tuning, where the model is trained on a labeled dataset.

In the context of autograding, we fine-tune our model using a dataset of student code submissions, along with their correct labels. This process involves adjusting parameter of the pre-trained model. We start with a raw base model CodeLLama-7B that has been pre-trained on a large amount of code-data, and then we refine it using task-specific data to make it more specialized.

For code evaluation, we provide the base model with a graded dataset, allowing us to train the model and adjust its parameters based on this data. This fine-tuning process helps ensure that the model produces accurate evaluations and minimizes the risk of hallucinations—instances where the model generates incorrect or irrelevant outputs.

2.6.1 Lora

Large Language Models have impressive capabilities on multiple tasks due to their pre-training. However, we might need them to perform on several downstream tasks.

These capabilities are generally achieved using fine-tuning where the model's parameters are updated to achieve good performance on the downstream tasks. However, updating all the parameters at the scale of large language models is computationally very expensive and requires a lot of data.

In Low Rank Adaptation (LoRA) [9] of Large Language Models, the pre-trained weights are frozen, and the dense layers are trained indirectly by optimizing rank decomposition matrices of their change during adaptation. The weight matrices are hypothesized to have a low intrinsic rank during adaptation.

For example, consider a weight matrix $W_0 \in \mathbb{R}^{d\times k}$. The update to this matrix is represented using a low rank decomposition as follows: $\Delta W = BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank r is far less than $\min(d, k)$. The matrix W_0 is frozen, but the matrices A and B contain trainable parameters. A random Gaussian initialization for A and zero initialization for B is used.

2.6.2 DPO

Reinforcement Learning with Human Feedback (RLHF) is a technique used to align Large Language Models (LLMs) with human preferences. Traditional RLHF methods consist of two steps: first, train a reward model that reflects human preferences, and then fine-tune the LLM using Reinforcement Learning to maximize these rewards without a large shift from the original model.

Direct Preference Optimization (DPO) [10] is a technique that optimizes the language model policy by directly improving the log probabilities of responses preferred by humans. Fine-tuning LLMs using DPO has proven to be more stable and computationally less expensive compared to RLHF using Proximal Policy Optimization (PPO).

2.6.2.1 Formulation

Denote the reference Language Model policy by π_{ref} and the optimal model policy by π_θ. Suppose we have a dataset of preferences given by $D = \{x^{(i)}, y_w^{(i)}, y_l^{(i)}\}_{i=1}^N$, then the policy objective is given by:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim D}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

where β is a parameter that controls the deviation of the optimal policy from the base reference policy. The reference model is ignored as $\beta \to 0$.

2.6.3 DORA

DoRA[11] improves upon the LoRA technique by breaking down the pre-trained weights of a model into two parts: magnitude and direction. The magnitude represents the size of the weights, while the direction indicates the orientation of the weights. In DoRA, these two matrices are fine-tuned separately—magnitude is adjusted in a straightforward way like regular fine-tuning, while direction is fine-tuned using the LoRA method. After fine-tuning, the two matrices are combined back together through matrix multiplication, resulting in an updated weights matrix. This approach enhances the learning ability and stability of the training process while keeping the efficiency of inference intact, allowing DoRA to perform better than LoRA on various tasks across different model architectures.

2.7 Django REST Framework (DRF)

Django is a python based framework for building out web applications. The django rest framework [12]is a library we can use with it to help us build out API so whether you're looking to build a public API or an API to simply distribute data to your own applications the Django rest framework is a great tool. Companies like Mozilla and Eventbrite have relied on the DRF to construct their applications, and the Django Rest Framework also meets our technical requirements.

In Django, there is a feature called an "app." The concept of an app is to build a specific functionality that is independent of other functionalities. This means that the functionality present in an app behaves like a plug-and-play component. BodhiTree uses Django, which has significantly contributed to its robustness and the seamless integration of various features. This solid foundation has proven effective in supporting complex functionalities. Inspired by its success, we also plan to build our TA Buddy using Django.

Each app has its own Models, Views, and Controllers (MVC), which are the basic components that enable it to function.

- **Model**: Django models allow us to work with data at a higher level, focusing on the data rather than the queries. We define data models (classes in Python) and use objects based on those model classes to run common operations. In the app, there is a file called `models.py` which is used to create Django models and interact with databases. Here, we define data entities and blueprints for data objects.

- **View**: The view handles the logic for the user interface. Views are functions or classes executed for different URLs, containing code that handles (evaluates) requests and responses.

- **Controller**: The controller is the key component that manages how everything in the app works together. It oversees communication between `views.py` and `urls.py`. When a URL is accessed, the controller creates objects from classes defined in `views.py`, calls a function with these objects, and returns the data to the front end.

Various URLs redirect to different pages. URL-Action mappings ensure that certain results are achieved when specific URLs are entered by the user. Some of the key features of DRF include:

- **Serialization:** Converts complex data types, such as querysets and model instances, into Python data types that can easily be rendered into JSON, XML, or other content types.

- **Authentication and Permissions:** Provides customizable and built-in authentication methods such as token-based authentication, OAuth, and more, making it easier to protect APIs.

- **Viewsets and Routers:** Simplifies the mapping of URL patterns to views, reducing boilerplate code and providing clean, maintainable API routes.

- **Browsable API:** Allows developers and users to explore APIs through a web-based interface, which is particularly helpful during development.

Chapter 3

Related Work and Literature Survey

In this chapter, we will delve into the ongoing research in the field of LLMOps, which focuses on the practices, techniques, and tools essential for the operational management of large language models in production environments. We will examine the LLMOps pipeline, covering the collection and preprocessing of training data for supervised instruction tuning, the deployment of custom LLMs, and the inference pipeline for making predictions. This literature survey aims to highlight the integration and deployment of large language models (LLMs) within production settings.

As mentioned in the introduction, our primary goal is to develop a robust system that integrates this subjective grading model into the BodhiTree platform. While much research has been conducted in the broader field of MLOps, we will look at relevant MLOps papers to gather insights, with a focused emphasis on LLMOps and its specific challenges and opportunities. Additionally, we will examine related work and approaches for efficiently integrating such models.

3.1 Software Architecture and Design For ML Models

The paper "Machine Learning Software Architecture and Model Workflow: A Case of Django REST Framework" [13] by Kennedy Ochilo Hadullo and Daniel Makini Getuno, looks at the challenges of building and deploying machine learning (ML) software. The authors explain that while ML has great potential, more than 80% of ML projects never make it to production. This causes many companies to lose

money. The paper points out several reasons for these failures. These include a lack of experienced ML developers, poor collaboration between data scientists and software engineers, and a lack of guidance on using established platforms like Django REST Framework (DRF). This study review the literature relevant to ML model. Some of them are :

3.1.1 Machine Learning as a Model (MLaaM)

MLaaM is the output of writing ML algorithms that run on data and represents what was learned by the algorithm on training data. An algorithm in ML is a procedure that is run on data to create a machine learning model.

3.1.2 Machine Learning as a Service (MLaaS)

Machine Learning as a Service (MLaaS) encompasses various services that provide machine learning tools within cloud computing frameworks. One of the primary advantages of these tools is that they enable customers to quickly initiate machine learning applications without the need to install specialized software or manage their own servers. MLaaS providers facilitate the development and deployment of machine learning projects by offering capabilities such as data transformation, predictive analytics, data visualization, and advanced machine learning algorithms.

3.1.3 ML Model Software Deployment

Software deployment includes all activities that make a software system usable. It involves delivering application modules from developers to users. The methods developers use to build, test, and deploy code can impact how quickly a product responds to customer needs and the quality of those changes. In machine learning (ML), deployment means taking a trained model and making its predictions available to users. Many data scientists, especially those without software engineering experience, may not fully understand this process, while software engineers might find ML model development challenging.

The main goal of the study was to find solutions to these problems and Create a suitable Machine Learning Software Architecture deployable with Python's Django REST Framework (DRF). The authors propose a new ML software design and workflow that can help teams deploy models successfully using DRF. It consists of six

sub-architectures: the User Interface, which connects users to the system; the Django API, integrating essential files for data processing; System Configuration Files, crucial for linking components and defining settings; Serialization/De-Serialization, handling the saving and restoring of ML models; and Server and Repository, leveraging Heroku for cloud deployments. Finally, the Command Line Utility facilitates project interactions and administrative tasks.

3.2 Designing ML Pipeline with Continuous Training

In this literature survey[14], it provides information regarding the importance of continuous training of ML Models and a robust way to do it. "Designing a Machine Learning Pipeline with Continuous Training for Time Series Forecasting by Jan Koskinen" explores importance of continuous training for Time series forecasting. It involves making predictions based on historical data. One of the critical challenges in time series forecasting is managing data drift or concept drift, where the underlying data distribution changes over time, causing the model's performance to degrade. This issue makes periodic retraining essential to maintain accuracy.

It proposes a a ML pipeline capable of continuous training using an error-based trigger. It serves different functionality :

- **Functionality 1: Model Training and Deployment Pipeline**
  The pipeline has a comprehensive set of tasks, including data collection, preprocessing, model training, evaluation, and deployment.

- **Functionality 2: Error-Based Retraining Trigger**
  An error-based retraining trigger will be implemented to automatically run the pipeline when the model's performance deteriorates beyond a predefined threshold.

- **Functionality 3: Performance Monitoring Service**
  A dedicated monitoring service will be established to continuously inspect the model's performance.

- **Functionality 4: Inference Service**
  An inference service will be integrated to utilize the trained models for generating forecasts.

3.3 Automating the Training & Deployment Of Models - MLOPS

Penghao Liang[15] iscusses the growth of MLOps (Machine Learning Operations) and its significance in addressing challenges related to model deployment and performance monitoring. MLOps is an emerging field that integrates machine learning systems with traditional software development and operations methods. It aims to address challenges in deploying machine learning models from project to production. Liang reviews the evolution of MLOps and its connection to traditional software development, proposing methods to improve existing MLOps systems and enhance productivity.

In a machine learning (ML) system, deployment goes beyond simply launching an offline trained model as a predictive service. It often involves complex multi-step pipelines that automate the retraining and deployment of models, replicating the manual processes data scientists typically perform. One significant challenge in production is that machine learning models can degrade not just due to coding issues, but also because of constantly changing data statistics. Even Jan Koskinen highlighted this as a major problem. So for model to stay accurate continuous retraining is required. The author highlights the importance of automated model training and explains how a version control system can ensure transparency and repeatability in the training process. Author proposes three methodology in-order to automate the training & deployment of ML Models.

3.3.1 Automation in Model Training

Automation helps to reduce human intervention by offloading most repeatitive tasks such as data preprocessing, feature engineering and model tuning saving time and resources while improving efficacy. It ensures repeatability and consistency in model training, improving model reliability and maintainability by using version control systems and automated workflows.

In real world deployments, compressed neural networks have edge over large neural networks (DNNS) in terms of significant reduction in storage and computational requirements of the model. When compressing neural networks, various optimization techniques such as pruning, quantization, low rank approximation are used to prevent performance loss and hardware optimization techniques such as hardware accelerators and dedicated processors to further improve the efficiency and performance of the model.

Pruning aims to reduce redundant structures, slim down DNN models while maintaining model performance. Microsoft team proposed and published the OTOv2 framework in ICLR 2023 which is the industry's first automated, one stop, user friendly and versatile framework for neural network training and structural compression.

3.3.2 Integration with CI/CD Pipelines

CI/CD pipelines enable continuous automation and monitoring throughout the software development lifecycle. Machine learning workflows, however, pose unique challenges, such as managing versioned environments and containerizing components for consistency across different environments. Version control systems help manage changes, while containerization ensures the portability of machine learning workflows, facilitating efficient deployment and management.

DevSecOps ensures security at every stage of application software development life cycle(SDLC) by embedding required safety checks into CI/CD automation. During actual deployments, integrating Dynamic Application security testing (DAST) into continuous integration/ continuous deployment (CI/CD) pipelines helps to find and address security vulnerabilities as early as possible, manage and deploy machine learning workflows more efficiently, accelerating the speed and quality of software delivery. .

3.3.3 Model Deployment and Monitoring

Large AI models are typically deployed in cloud-native environments like Kubernetes (K8s), which offers flexible resource management, elastic scaling, containerized deployment, and service orchestration. K8s also supports model version management,

rolling updates, and persistent storage solutions, making it an ideal platform for deploying large-scale AI models.

Continuous monitoring of machine learning models is essential to ensure stable performance after deployment. Monitoring metrics like accuracy, latency, and throughput help detect performance degradation or anomalies, prompting timely interventions such as model retraining or parameter adjustments. Feedback loops created through continuous monitoring improve model performance and adapt to changing needs.

3.4 Traditional ML vs Generative AI Pipeline

Traditional machine learning(ML)[16] has a structured process that aim to extract meaningful features for model performance.A machine learning pipeline is a series of automated processes involving data pre-processing, feature engineering, training, tuning, and deployment. It encompasses the entire workflow from gathering raw data, feature extraction to model training, evaluation and deployement. Traditional machine learning pipeline is initiated by gathering pertinent data from diverse sources. Raw data is cleaned by addressing missing values, outliers and handle categorical variables, scaling numerical features. The goal of feature engineering is to sculpt a dataset that can optimally fuels model training. The model is then trained on the training data, and evaluated using validation set.This phase involves analysing performance metrics, fine-tuning hyperparameters, and iterating on the model or algorithm based on evaluation results.

In recent years, the field of artificial intelligence (AI) has made significant strides, overcoming past setbacks with advancements in algorithms, computing power, and data collection. Generative AI generate text, images, and audio based on existing data. It is subset of AI, that offers intelligent responses to human prompts by understanding word relationships. Generative AI aims to design effective prompts that guide AI in producing desired outputs. It emphasizes prompt engineering, fine-tuning language learning models, and deployment. This shift from ml not only transforms the approach to data processing but also showcases the versatility of Generative AI in producing more sophisticated and context-aware outputs.

Chapter 4

AI Methodologies

Artificial Intelligence (AI) is the heart of the TA Buddy tool, which help to automate and improve the grading process. In this section, we describe the work we have done in AI that has led to the creation of TA Buddy. During our seminars and R&D, we explored different ways to solve the challenges of understanding code. We used several techniques, often used for generating code, and adapted them to help with understanding code for grading assignments more efficiently. We also looked at how Large Language Models (LLMs) have evolved, and by using prompt engineering, we were able to improve their ability to understand and handle more complex tasks.

4.1 Prompt Engineering

One of the primary methods investigated was **Prompt Engineering**, which formulates grading tasks as prompts to guide LLMs, particularly **CodeLlama 7B-Instruct**. Hence we decided to formulate the task of grading as a pipeline involving prompting LLMs, where the grading task is formulated as a prompt and the LLM is expected to provide a suitable grade for the student submission provided. This model was chosen for its open-source nature, ability to handle extensive contexts (up to 100,000 tokens), and its fine-tuning for following human instructions.

4.1.1 Zero-shot Prompting

As mentioned above, since LLMs already possess a lot of knowledge regarding a wide variety of topics, we can leverage this knowledge to directly prompt without feeding them any other additional information to solve our downstream tasks. In

zero-shot prompting, we directly ask a question, the LLM is given clear instruction with no specific examples. We provide all the relevant information like the problem statement, student submission, the criterion and directly query the LLM to obtain a grade. Note that the LLM is only used for inference in this process.

Zero-shot Prompt template

```
<s>[INST] <<SYS>> Your task is to choose the MOST suitable option
among a set of options I provide, about a code which will also be
provided.  Give your output as a json with two fields :  "answer"
and "reasoning".  Do not output anything else.  Strictly follow this
output format.
<</SYS>>
### Context :  <Summarised Problem Statement>
### Code :  <Student code>
### Task :  Choose the option which is the most suitable for the
criterion <Criterion description> for the above code.  Give your
output as a json with two fields :  "answer" and "reasoning".
### Options :  <List of rating desciptions for the criterion>
### Response :  The required output in json format is :  [/INST]
```

Figure 4.1: Zero-shot prompt template

4.1.2 Few Shot Prompting

Few Shot Prompting is a method where a few similar examples are provided to achieve better results. There is no need for massive training data, while specific examples are pro- vided directly inside the prompt. So that the model can use the provided examples as a reference and can solve the asked question correctly. And few-shot prompting works because LLM have impressive in-context learning capabilities. Similar to zero-shot prompting, we again prompt the model to give its output in a json output. The following components will be provided to the model :

- Simplified problem statement
- Demonstrations
- Student submission

- Description of the criterion
- Ratings associated with the criterion (as options)

Few-shot Prompt template

```
<s>[INST] <<SYS>> Your task is to choose the MOST suitable option
among a set of options I provide, about a code which will also be
provided.  I'll give you a few examples showing how to do it.  Give
your output as a json with two fields:  "answer" and "reasoning".  Do
not output anything else.  Strictly follow this output format.
<</SYS>>
### Context :  <Summarised Problem Statement>
### Examples :
## Task:  Choose the option which is the most suitable for the
criterion <Criterion description>
<List of demonstrations> (Collection of pairs of <(Code, Actual
grade))>
### Code :  <Student code>
### Final Task:  Choose the option which is the most suitable for
the criterion <Criterion description> for the above code.  Give your
output as a json with two fields:  "answer" and "reasoning".
### Options :  <List of rating desciptions for the criterion>
### Response :  The required output in json format is :  [/INST]
```

Figure 4.2: Few-shot prompt template

4.1.3 Chain Of Thought Prompting

Large Language Model a big success but fails to tackle complex arithmetic, commonsense and symbolic reasoning task.Unlike normal prompts here we provide a chain of thoughts.

```
Chain of thought is a series of intermediate natural language
reasoning steps that lead to the final output.
```

It overcomes two major problems which impact the performance of LLMs.

1. Complex Task : Breaking the big problem into smaller sub-problems helps language model to understand the subsequent step in a better way and give us the correct answer.

2. Improvement of few shot learning : It enhances few-shot learning by providing explicit instructions or examples to the model, allowing it to quickly adapt and generalize to new tasks.

[17] show how breaking a problem into increasingly harder subproblems and solving them enables complex reasoning capabilities for the LLM. Incorporating both these methods, in our approach, we provide a few demonstrations for each criterion. For each criterion, we break the grading task into several smaller tasks that focus on different details of the code. Each answer to a small question helps with the next one, and by the end, the final answer helps the LLM decide the grade.

We used two different strategies that combine the strengths of the three methods mentioned earlier:

4.1.3.1 Strategy 1

For each code submission, we show four examples of different student work. In each example, we first summarize the functions in the student's code. Then, we summarize the entire code and check if it meets the given criterion. We believe that summarizing the code before evaluating it helps the model perform better. We demonstrate this strategy by checking the criterion, "Is an element of the array wasted?" for a circular queue assignment. The demonstration shows how we summarize each function and use those summaries to assign a grade to the submission.

4.1.3.2 Strategy 2

In this approach, we teach the LLM to ask a series of questions about the code. We break the original problem into smaller questions to determine if the specified criterion is met. It shows how we ask a series of questions to answer the main question. We chose eight submissions for the circular queue problem to evaluate the criterion, "Is an element of the array wasted?" Since this question can only have a "YES" or "NO" answer, we explicitly ask the LLM to respond with one of these options and then compare its answer to the correct one. We also check the LLM's reasoning for each submission to verify if it is correct. The results from Strategies 1 and 2 on these eight

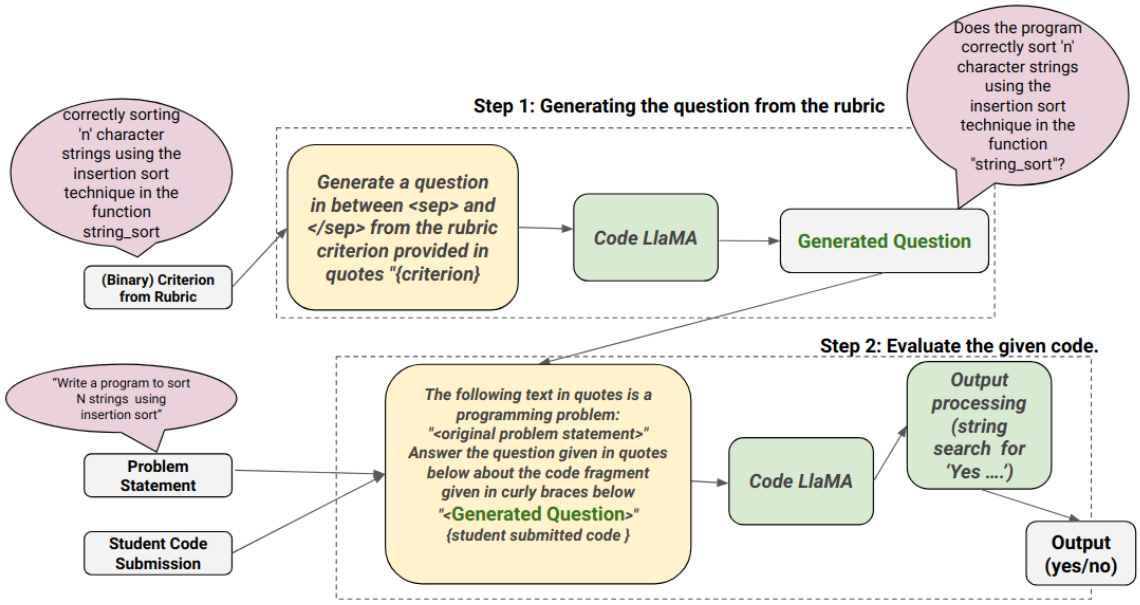

Figure 4.3: Fully automated prompt chaining pipeline for binary grading

submissions are presented in Table 4.1. In these tables, "C" stands for Correct, "W" for Wrong, and "AC" for Almost Correct. Based on these limited results, Strategy 2 performs better than Strategy 1 when using CodeLlama 34B Instruct.

An Important Observation : When we applied Strategy 2 prompts with Claude 2, an LLM by Anthropic, the model's answers and reasoning were correct for all eight student submissions. This indicates that the size of the model is a key factor in our experiments.

Table 4.1: Comparison of CoT Strategy 1 and 2 using CodeLlama 34B Instruct

Submission	**Strategy 1**		**Strategy 2**	
	Answer	*Reasoning*	*Answer*	*Reasoning*
1	W	W	W	W
2	W	W	C	C
3	C	C	C	C
4	W	W	W	W
5	C	C	C	C
6	W	W	W	C
7	C	C	C	C
8	C	W	C	AC

4.1.4 Supervised Fine-tuning

Recognizing that prompt-based approaches alone might not achieve optimal performance, the thesis delved into **Fine-tuning** techniques to adapt the LLMs more effectively. It employed **Supervised Fine-Tuning (SFT)** using **Quantized LoRA (QLoRA)** to minimize the computational demands associated with full fine-tuning. This approach utilized a dataset of prompts and their expected completions, facilitating the grading process through an efficient training mechanism.

To construct the fine-tuning dataset, we utilized ground truth TA grades to create prompts for each student submission, allowing the model to learn from expected responses.

4.1.5 Fine-tuning using DPO

Direct Preference Optimization[14], or DPO trains large language models to align with human preferences without complex reinforcement learning. It skips training a separate reward model. **Direct Preference Optimization (DPO)**, a method that optimizes the model by improving the likelihood of preferred responses compared to rejected ones. This technique involved creating preference pairs for student submissions, where the "chosen" field represented the ground truth grade, and the "rejected" field included any alternative grade.

Through these approaches, the work laid the groundwork for more accurate and efficient autograding systems, informing future research directions in automated assessment of programming assignments.

The scope of the work is just limited the implementation of model which is capable of taking input as a prompt which contains student submission , problem statement , criterion along with it ratings. And in response provides the best suited rating for the criterion. In a realworld one need to have a proper robust system so that one can easily integrate it with lms and autograding can be done.

Data format for DPO

{"prompt": "<s>[INST] <<SYS>> Your task is to choose the MOST suitable option among a set of options I provide, about a code which will also be provided. Give your output as a json with a single field 'answer'. Do not output anything else. Strictly follow this output format at any cost.
<</SYS>>
Context : Write a C++ program that determines the approximate square root of a given number S up to a desired precision epsilon. The program should take S and epsilon as input and print the value of the square root up to 5 decimal places. The Babylonian method is used to find the square root. The method starts with an initial guess and iteratively improves the guess until convergence is achieved. The program should terminate when the absolute difference between the current guess and the previous guess falls below epsilon.
Code :
#include<iostream>#include<cmath>using namespace std;double sq_root(double S, double epsilon){double x=2;while(fabs(x/2-S/(2*x))>epsilon){x = x/2+S/(2*x);}return x;}int main(){double S, epsilon;cin>>S>>epsilon;cout<<fixed; cout.precision(5);cout<<sq_root(S, epsilon);}
Task :
Choose the option which is most suitable for the above code for the criterion 'Whether babylonian method is implemented in a function'. Give your output as a json with two fields : "answer" and "reasoning". Do not output anything else. Strictly follow this output format.
Options :
A. Separate function is used and function signature is correct
B. Separate function is used but function signature is not correct
C. totally wrong
Response : The required ouptut in json format is :
[/INST] {"answer" : 'A. Separate function is used and function signature is correct'} </s>", "chosen" : {"answer" : 'A. Separate function is used and function signature is correct'}, "rejected" : {"answer" : 'C. totally wrong'} }

Figure 4.4: An example showing the data format for DPO

Chapter 5

Design and Architecture of TA Buddy

TA-BUDDY is an AI-assistant designed for the qualitative autograding of programming assignments. It is built on Code Llama-7b, a pre-trained LLM. It goes beyond checking code correctness by providing detailed evaluations based on grading rubrics, offering instructors a powerful tool for efficient assessment and significantly reduces the time spent on grading, a result which will be discussed in detail later.

To align the model to grading task, TA-BUDDY utilizes the Direct Preference Optimization (DPO) method to fine-tune the Code Llama-7b-Instruct model on a specialized dataset of problem statements, rubrics, student code submissions, and rubric-based ratings.

Like other models, TA-BUDDY also can experience model drift over time, where performance may degrade as the nature of submissions evolves. So we need a Continuous training that automatically retrains models incase of drift to adapt to changes in the data before it is redeployed[18]. So for this we require a robust data collection pipeline that continuously gathers new submissions and their corresponding ratings. Coupled with this, an automated retraining pipeline regularly updates the model by retraining with new data, ensuring TA-BUDDY maintains its accuracy and reliability over time.

5.1 High Level Architecture

In this section, we will describe the high-level architecture. The system already supported for manual grading and we introduce automated assessment of student

submissions, data export and continuous model improvement through retraining. We propose an architecture which supports inference, data collection and retraining. *The scope of intelligent grading is limited to Instructor's/TA so this feature is only available to them.*

The Fig 5.1, illustrates the structure of Intelligent Grading System of BodhiTree. The frontend interface for instructors and students is implemented in React, while the Django backend handles the server-side logic and communication with other components. Instructors/TA's either can initiate the Intelligent Grading or Manually grade.

For AI-Assisted,*(Represented With Blue Line)* once the instructor submits the Intelligent Grading Request a celery task is inititated where the grading task is queued and request is sent to AI Server. Once the grading completes, the result is sent back from AI Server to BodhiTree Backend. On backend it populates back the result in DB.

For Manually Grading,*(Represented With Green Line)* the Instructor's/TA's can use the BodhiTree Rubric Grading Interface to grade the student submission. These graded data can act as a ground truth results for the student submission which can be exported to AI Server via the Export Data Button on BodhiTree Interface which triggers the exportData task in the Bodhitree Backend and as a result data is exported to the AISever. *(Represented with Orange Line)*

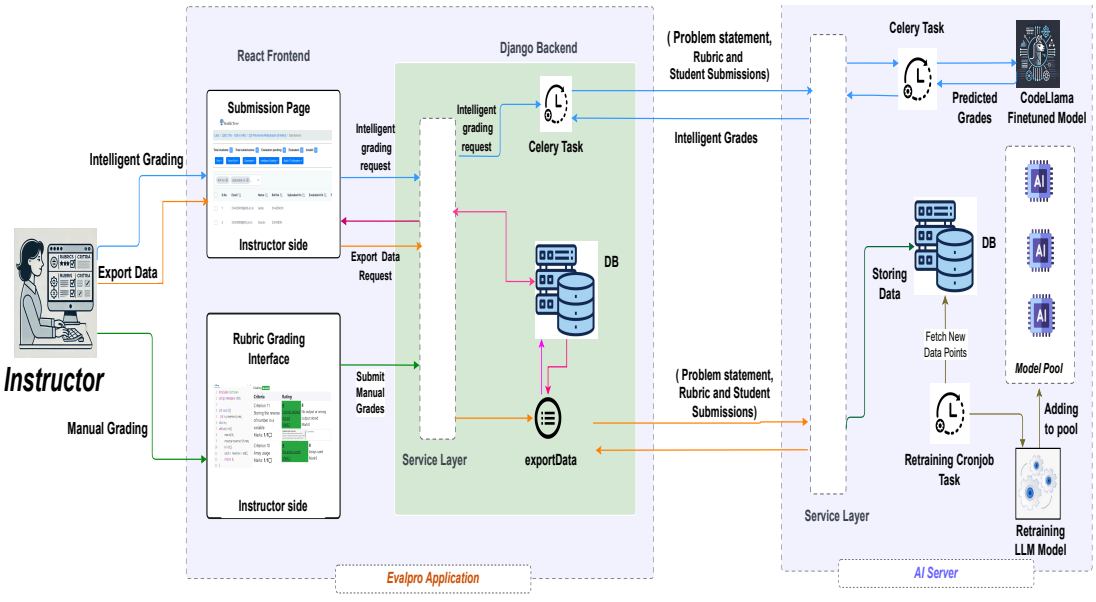

Figure 5.1: High Level Architecture

From the architecture, we can observe that there are three crucial tasks Inference, Data Collection & Retraining. For each tasks, we implement a separate pipeline so that the entire system is robust and resilient. In the subsequent sub-sections we will see the architecture of these pipeline and in the next section we will explore this pipelines in detail.

5.1.1 Inference

The figure below illustrates the architecture of the TA Buddy AI Server interacting with the BodhiTree platform to perform intelligent grading of student submissions. Given a Problem Statement, Criterion and a Submission, this pipeline can suggest AI Rating and Comment for it. It begins when an instructor clicks the "Intelligent Grading" button on the frontend, which triggers an API call to the backend. The backend extracts relevant data (problem statement, rubric, and student submissions) from the database. A Celery task is initiated to process this data and send it to the AI Server.

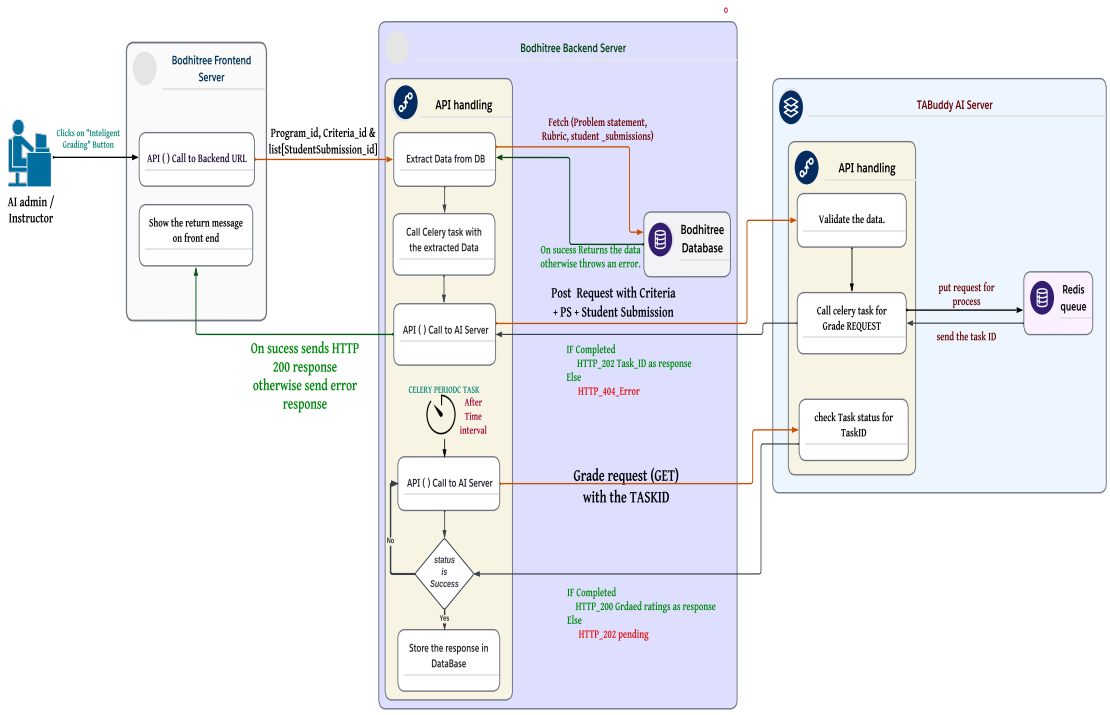

Figure 5.2: Architecture Of Inference Pipeline

Within the AI Server, also a Celery task begins which accepts the grading request puts it in redis queue and returns the task id. The AI Server performs the evaluation task. After each certrain interval bodhitree backend uses the task id and checks for results. If the grading task is complete, the AI Server sends a graded response back to the backend, which is stored in the database and displayed to the user. If the task is still pending, a periodic Celery task checks the status until completion.

5.1.2 Data Collector

The figure below illustrates how data is collected and processed in BodhiTree and then sent to AI server. When an instructor clicks the "Export All" button on the frontend, it sends a request to the backend. The backend fetches the required data (such as problem statements, rubrics, and student submissions) from the BodhiTree database, formats it into JSON, and sends it to the AI server. The AI server validates, processes, and stores the data in its database (MongoDB). If the process is successful, the AI server returns a confirmation; otherwise, it sends an error.

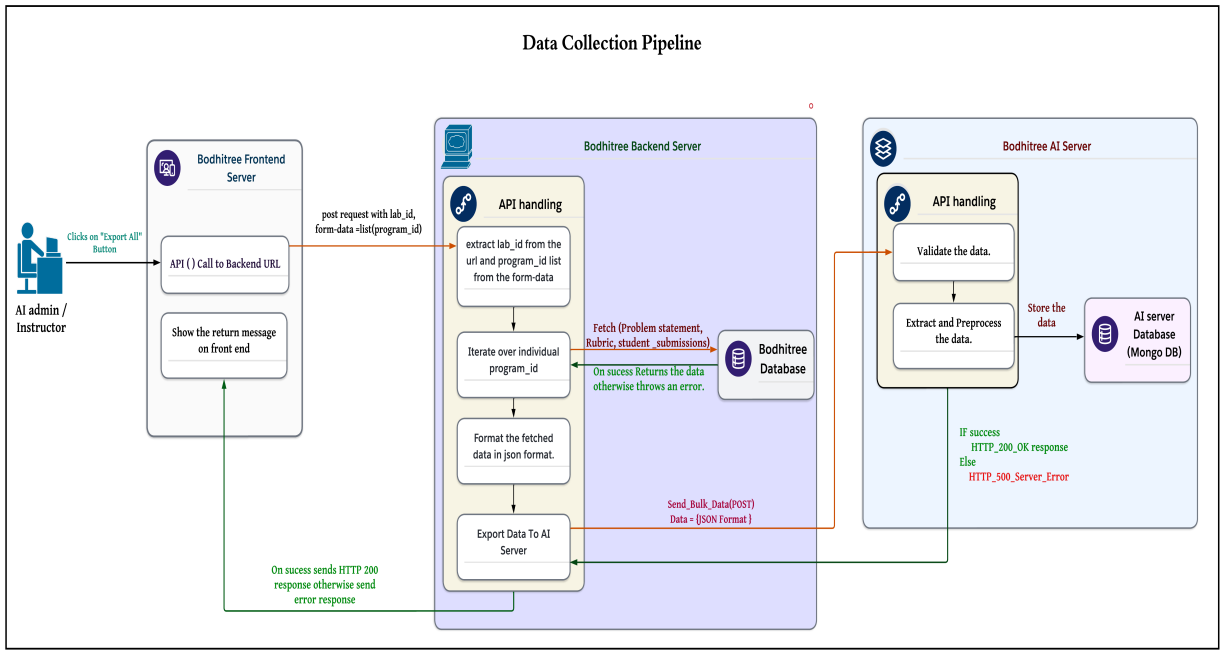

Figure 5.3: Architecture Of Data Collector

5.2 Pipeline Descriptions

As discussed in the pior section the complete system relies on three pipelines: Inference, Data Collection, and Retraining. In the subsequent subsection we will discuss these in details.

5.2.1 Inference Pipeline

The Inference Pipeline is the core functionality that allows TAs/Instructors to perform real-time code grading using the AI model. When a TA/Instructor submits a code sample through the web interface, the AI model processes the input and returns a grading suggestion based on predefined criteria. This pipeline is designed to provide quick, consistent, and accurate grading assistance to TAs/Instructors.

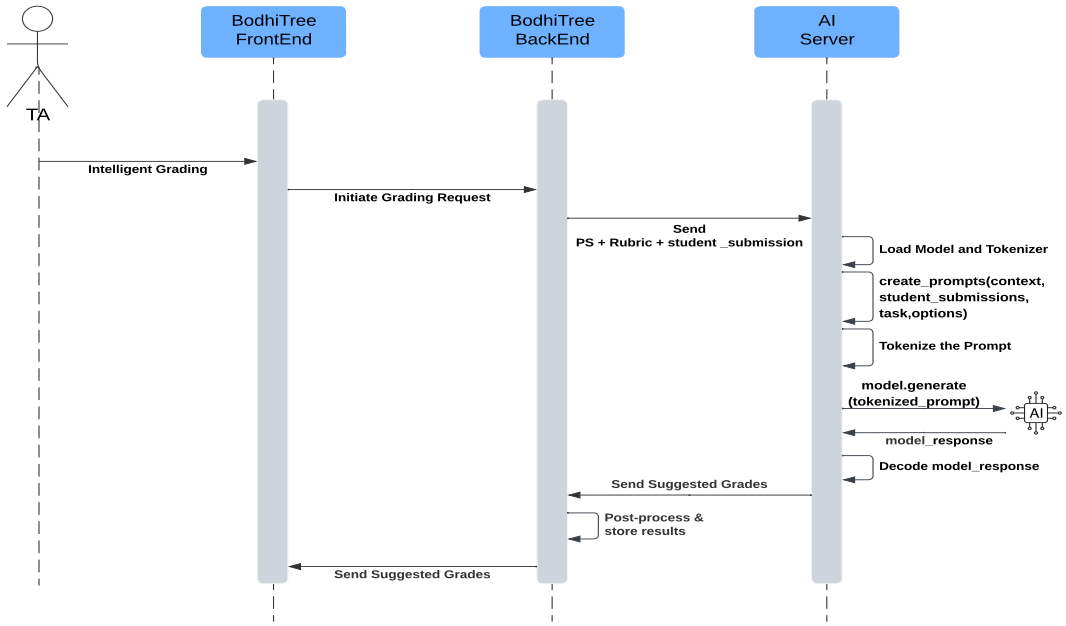

Figure 5.4: WorkFlow Of Inference

5.2.2 Inference Pipeline Process Flow

1. **BodhiTree Frontend (Initiating Grading Request):**
    - **Action:** A TA selects a lab question and submits a request for grading assistance from TA Buddy.

- **Data Involved:**
  - **Lab Information:** Lab ID, Problem Statement (ps), Rubric, and Student Submissions.

2. **BodhiTree Backend (Structuring Data):**
   - **Action:** The backend receives the grading request from the frontend and structures the necessary data.
   - **Key Components:**
     - **Problem Statement (ps):** Simplified description of the programming task.
     - **Rubric:** A dictionary containing criteria and their associated grading options.
     - **Submission:** A dictionary of student submissions.
   - **Process:**
     - The backend prepares this data in a structured format.
     - The structured data is then sent to the AI Server for further processing.

3. **AI Server:**

   (a) **Loading the Model:**
   - Upon receiving the request, the AI Server loads the CodeLlama model and its tokenizer.

   (b) `grade_submission(tokenizer, model, device, problem_statement, submission, rubrics)`:
   - This function handles the grading process for each student submission.
   - **Key Components:**
     - **Inputs:**
       * **submission:** The student's code.
       * **ps (Problem Statement):** Context for understanding the submission.
       * **rubric:** Criteria for evaluating the submission.
   - **Process:**

- Iterates through each criterion in the rubric.
- Calls `create_prompt` to generate a prompt for each criterion.

(c) **Prompt Creation (`create_prompt`):**

- **Action:** Creates a detailed and structured prompt to query CodeLlama.
- **Parameters:**
  - **Context (Problem Statement)**
  - **Code (Student Submission):** The actual code written by the student.
  - **Task:** Specifies the exact task for the model, which is to evaluate the submission based on the criterion and provide a JSON response.
  - **Options:** The grading options for the specific criterion (e.g., "A. Good variable names", "B. Poor variable names").
- **Prompt Structure:**
  - The prompt is formatted as follows:

(d) **Query CodeLlama:**

- The AI Server sends the generated prompt to CodeLlama for processing.
- **Output:**
  - **JSON Response:**
    - **Answer:** The suggested grade (e.g., "A").
    - **Reasoning:** Explanation behind the selected grade.

(e) **Result Compilation:**

- **Action:** After querying CodeLlama for all criteria, the AI Server compiles the results.
- **Data Compiled:**
  - A list of JSON objects containing grades and reasoning for each criterion.

(f) **Sending Back Results:**

- **Action:** The AI Server sends the compiled results back to the BodhiTree Backend.

4. **BodhiTree Backend (Result Handling):**
   - **Action:** The backend processes the results received from the AI Server.
   - **Result Storage:** The backend populates the results.
5. **BodhiTree Frontend (Displaying Results):**
   - The TA can now see the suggested graded results via the BodhiTree platform.
   - **Output:**
     - The TA sees the suggested grades with an underline and reasoning for each criterion for the student submissions.

5.2.3 Data Collector Pipeline

The Data Collector Pipeline is responsible for gathering and storing data from the Bodhitree interface, which is essential for improving the AI model which is using for the grading. This pipeline operates in both manual and automated modes, allowing AI Researchers to collect data as needed. The data collected includes problem statement, rubric, code submissions, AI-generated grading suggestions, TA/Instructor feedback, and any modifications made to the initial model sugessted grades.

5.2.3.1 Requirements

What all to store ?

1. Problem Statement: Problem descriptions and related metadata.
2. Student Submissions: Details about code submissions by students.
3. Criterion Descriptions: Details and descriptions of evaluation criteria.
4. Ratings for Respective Criterion: Ratings and associated details for each criterion.

5. Grading History: Records of ratings and comments (both manual and AI-generated) for submissions based on criteria.

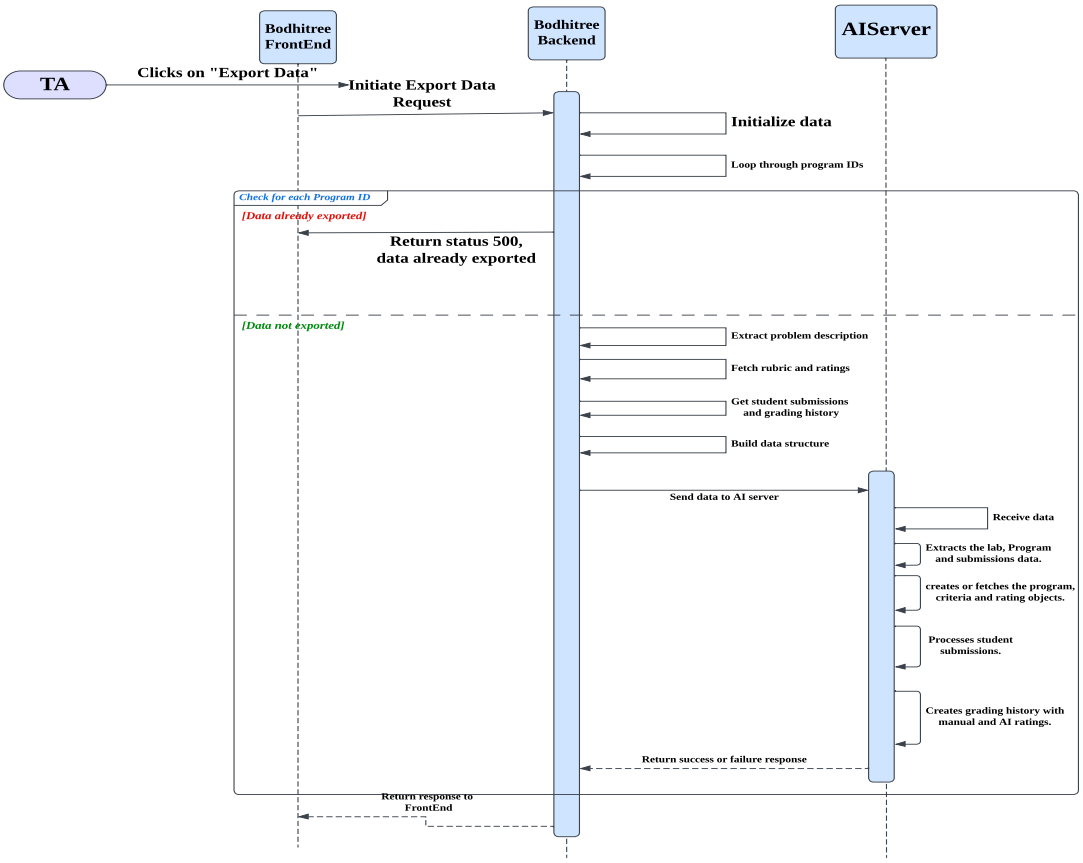

Figure 5.5: Workflow Of Data Collector

5.2.3.2 Data Collector Pipeline Process Flow

1. BodhiTree Server Front-End Workflow

   (a) **User Action:**

   Step 1: User clicks on the "Export All" button on the submission page.

   Step 2: The front-end sends a POST request to the backend URL `evalpro/rubric/exportSubmission/{labId}`.

   Step 3: The request contains form data with a list of `program_Id`.

2. BodhiTree Server Back-End Workflow

(a) **Request Handling:**

Step 1: The server listens for the POST request at `evalpro/rubric/exportSubmission/{labId}`.

Step 2: It extracts the `labId` from the URL and the `program_Id` list from the form data.

Step 3: For each `program_Id` in the list:

- Fetch the associated Problem Statement.
- Fetch the Rubrics.
- Fetch the Submission.

Step 4: Construct a well-formatted JSON structure containing the Problem Statement, Rubrics, and Submission for each `program_Id`.

Step 5: Send the JSON data to the AI Server via a POST request.

Step 6: After receiving a response from the AI Server, return a success message to the front-end.

3. AI Server Workflow

(a) **Data Processing:**

Step 1: The AI Server listens on port 27017 for incoming POST requests.

Step 2: Upon receiving the request, validate the JSON structure to ensure it conforms to the expected format.

Step 3: Extract and process the data from the JSON, molding it into a storage-efficient format.

Step 4: Send an HTTP 200 (OK) status code back to the BodhiTree Server as a response.

Step 5: Store the processed data.

5.2.4 Continuous Retraining Pipeline

The Continuous Retraining Pipeline ensures that the AI model remains accurate and effective over time. By periodically retraining the model with new data collected from the Data Collector Pipeline, this system adapts to changes in coding styles, grading criteria, and other variables. The diagram shows an automatic process for retraining a model, which is triggered by a scheduled task (cron job) that runs at regular intervals.

When the cron job starts, it checks if there is enough new data to retrain the model. It also makes sure the same process isn't running twice using a lockfile. If enough new data is available, the system looks for an idle GPU (a powerful computer processor) to handle the retraining. If no idle GPU is found, the process stops and waits for the next scheduled run. If a GPU is available, the retraining begins, and the system checks if it finishes successfully. If it does, the model is saved, and the system updates the status to show that the problem has been trained. If retraining fails, the process stops without making any changes. This process helps retrain the model efficiently, only when new data is available and the required resources are free.

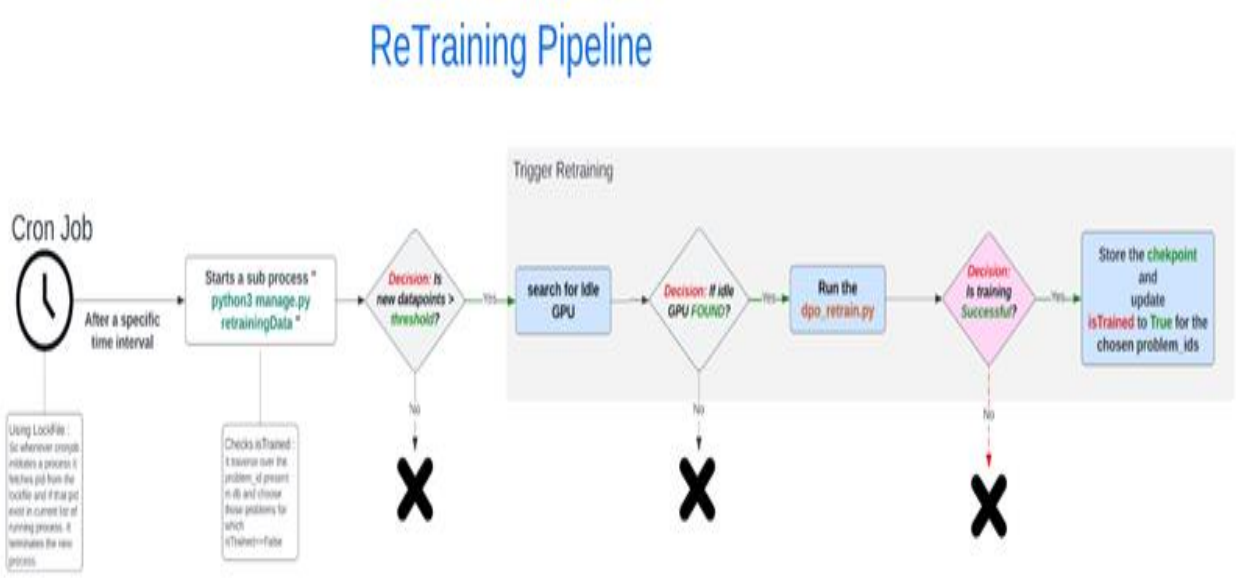

Figure 5.6: Working Of ReTraining Pipeline

5.2.4.1 Retraining Pipeline Process Flow

1. **Cron Job Initiation:**
   - A cron job starts at specific time intervals.
   - It triggers a subprocess running `python3 manage.py retrainingData`.
2. **Data Point Check:**
   - The system checks if new data points exceed a threshold.
     - If **no**, the process terminates.

  - If **yes**, it proceeds to the next step.

3. **GPU Availability Check:**
   - The system searches for an idle GPU.
     - If **no** idle GPU is found, the process terminates.
     - If an **idle GPU** is available, it continues.

4. **Model Retraining:**
   - The system runs `dpo_retrain.py` to retrain the model.
     - If **unsuccessful**, the process ends.
     - If **successful**, it moves to the final step.

5. **Update and Store:**
   - The system stores the new checkpoint.
   - It updates the `"isTrained"` status to `True` for the chosen problem IDs.

Chapter 6

Implementation Details

We discussed Design and Architecture in Chapter 5. In this section, we will focus on the database design and the implementation details of Ta Buddy in detail.

6.1 Database Design

Our AI autograder, TA-BUDDY, is powered by a pre-trained large language model and requires continuous retraining to prevent model drift over time. For retraining, high-quality data is crucial, storing and managing this data effectively on AIServer is also essential. A robust database on AIServer plays a critical role in this process, enabling the efficient storage, retrieval, and management of training data.

BodhiTree already has a robust and solid database design that allow us to store vast quantities of data while still being easy to alter as needed. Each field in the database is self explanatory and builtin checks for faulty entries. Motivated by the robust and solid database design of BodhiTree, we built a database on AIServer that is just as robust. Although the database is NoSQL for AIServer, we incorporated the models in such a way that it remains similar to an SQL database, ensuring queries are efficient and faster. As Rubric Grading is a component of the Bodhitree web application, we'll need to know few concepts related to Bodhitree, such as

- **User**: Either Instructors, Students, or TAs.
- **Course**: This is a subject that the instructor is teaching.
- **Program**: This is a programming activity.

- **Student Submission**: This represents the submissions of students in various activities.

We will explore some the models of BodhiTree which is going to be used by DataCollector to extract the data and then format it and send it to AIServer inorder to be stored in DB.

6.1.1 Program

This program model extends the Base program model. It stores all the program activity information such as its name,description, programming languages allowed etc.

Listing 6.1: Django Model for Program

```
1
2  class Program(BaseProgram):
3      '''
4      Program model.
5      '''
6      id = models.BigAutoField(primary_key=True)
7      lab = models.ForeignKey(BodhiLab,
           on_delete=models.CASCADE)
8      is_published = models.BooleanField(default=False)
9      created_on = models.DateTimeField(auto_now_add=True)
10     .
11     .
12     .
```

6.1.2 Rubric Criteria

This model stores the criterion information.

Listing 6.2: Django Model for Storing Criterion

```
1  class RubricCriteria(models.Model):
2      '''
3      Model for storing Criterion information
4      '''
5      id = models.BigAutoField(primary_key=True)
6      title = models.ForeignKey(Tag, on_delete=models.CASCADE)
7      description =
           models.CharField(max_length=MAX_CHARFIELD_LENGTH)
8      # To specify to which program does the rubric belongs.
```

```
    program = models.ForeignKey(Program,
        on_delete=models.CASCADE)
    # Variable which indicates that a ai grading model
        available
    is_ai_grading_available =
        models.BooleanField(default=False)
```

6.1.3 Rubric Rating

This model stores the rating information which corresponds to a particular criterion.

Listing 6.3: Django Model for Storing Rating

```
class RubricRating(models.Model):
    '''
    Model for storing rating information
    '''
    id = models.BigAutoField(primary_key=True)
    title = models.ForeignKey(Tag, on_delete=models.CASCADE)
    description =
        models.CharField(max_length=settings.MAX_CHARFIELD_LENGTH)
    # To specify to which criterion does the rating belongs.
    criteria = models.ForeignKey(RubricCriteria,
        on_delete=models.CASCADE)
    # Marks awarded to student if TA assign this rating for
        the criterion.
    marks = models.FloatField(default=0)
```

6.1.4 Manual Grading History

This model contains the grades and the comments given by the Instructors/TA after evaluation.

Listing 6.4: Django Model for Manual Grading History

```
class ManualGradingHistory(models.Model):
    ''' Contains The Rating Chosen & Comments For Respective
        Criterion Manually'''
    id = models.BigAutoField(primary_key=True)
    student_submission = models.ForeignKey(StudentSubmission,
        on_delete=models.CASCADE)
    criteria = models.ForeignKey(RubricCriteria,
        on_delete=models.CASCADE)
    rating = models.ForeignKey(RubricRating,
        on_delete=models.CASCADE, null=True)
```

```
    comments = models.TextField(blank=True)
    evaluated_on = models.DateTimeField(auto_now=True)
    is_graded_by_ai = models.BooleanField(default=False)
```

6.1.5 AI Grading History

This model contains the grades and the comments suggested by the TA Buddy.

Listing 6.5: Django Model for AI Grading History

```
class AIGradingHistory(models.Model):
    ''' AI Suggested Ratings For Individual Criterion'''
    id = models.BigAutoField(primary_key=True)
    student_submission = models.ForeignKey(StudentSubmission,
       on_delete=models.CASCADE)
    criteria = models.ForeignKey(RubricCriteria,
       on_delete=models.CASCADE)
    rating = models.ForeignKey(RubricRating,
       on_delete=models.CASCADE)
    evaluated_on = models.DateTimeField(auto_now=True)
    comments = models.TextField(default="Not Present",
       blank=True)
```

We introduced a new model in Evalpro 2023, AI History. AI Grading History stores all the intelligent grading details such as AI Suggested Ratings & Comments. AI History stores the track of data exported to AI Server.

This Fig 6.3 represents the entity-relationship (ER) diagram for AI Server. This schema helps to manage and organize data needed for training machine learning models with minimal effort. The model includes five main entities: `GradingHistory`, `Rating`, `Submission`, `Criteria`, and `Problem`. These entities are linked through foreign key relationships, ensuring efficient and organized data management.

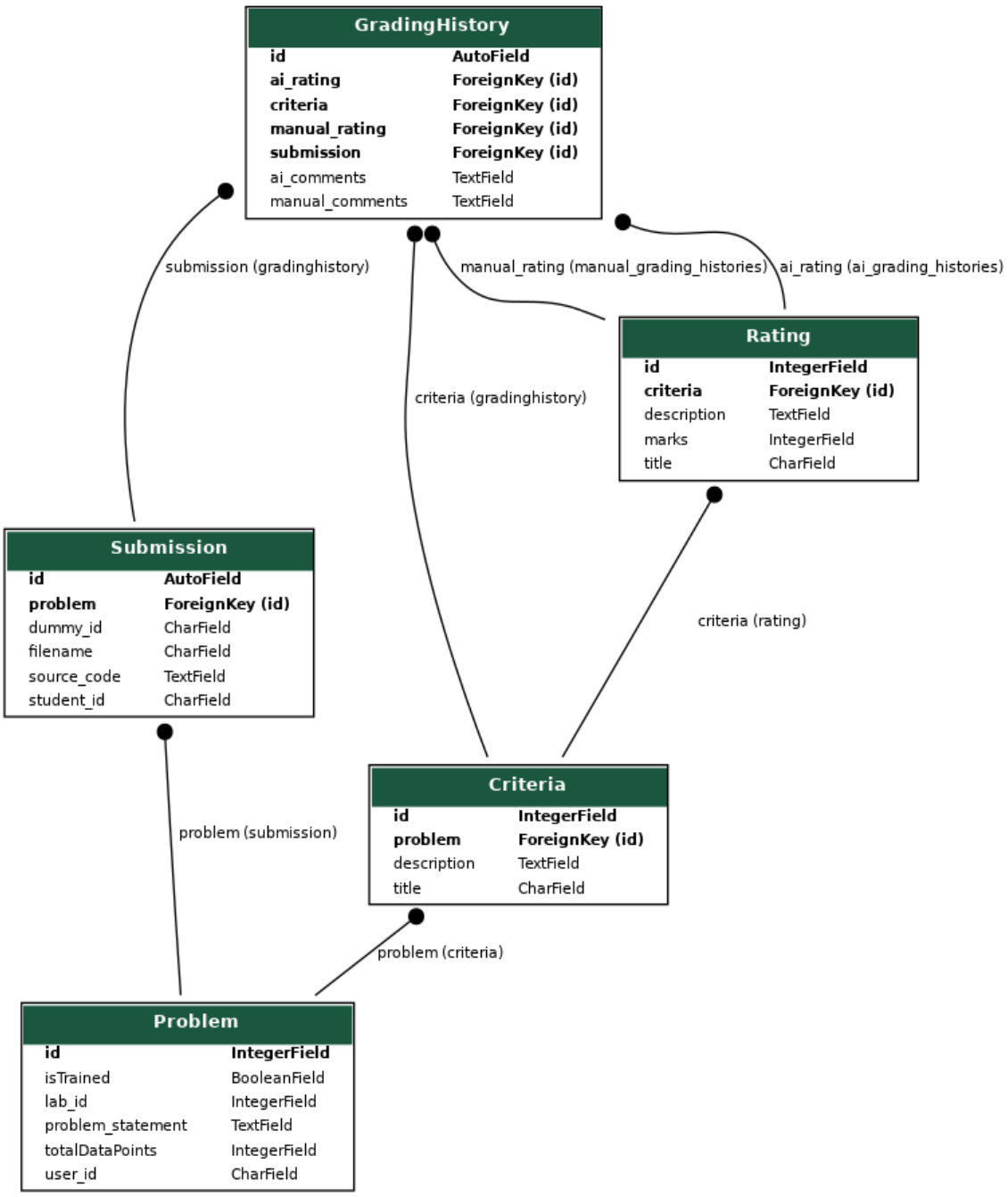

Figure 6.1: ER Diagram Of AI Server

6.2 Communication and Service Layer at BodhiTree

We outlined the database design we would need to construct in earlier section. Now that is done, let us look at how we designed the communication and service layer on bodhitree side.

6.2.1 URLs

In this section, we will look at how these URLs are linked to the API calls that will execute a service for us. If we look at listing 6.6, we can see that we have entered two URLs, and if we look at the right side of the URL, We can see that it's linked to an API call named Rubric. Normally, we'd link it to a direct function call, but because the Rubric is built with APIVIEWS, which are class-based API calls, we need to append asview() to it to tell Django that a function call needs to be made inside the Lab based on the request type. When these 5 URLs are multiplied by the request type, 20 new types of services can be identified, we will see how these API calls are implemented in the next section.

Listing 6.6: URLs implementation

```
urlpatterns = [
        path('rubric/predict/<str:id>/<str:criteria>',
           Rubric.as_view()),
    path('rubric/exportData/<str:id>', Rubric.as_view()),
]
```

6.2.2 Views

The views contains whatever arbitrary logic is necessary to return a response for a request. We'll write all the request handlers of API calls in `views.py`. Using the parent class APIVIEW, we've imported all the functionalities from Django including GET, POST, PUT, and DELETE. But we will just be looking at PUT and POST, for AI-Assisted grading there will a put request with form-data as the list of student submission and url appended with program_id and criterions_id. For export of data we will perform a post request with all the relevant data required in form-data. The figure below illustrates the Class Based API call for export data and ai suggested grades.

Listing 6.7: Classed Based API Call for Export Data

```
    def post(self, request, id, format=None):
        rubric_obj = RubricService(lab_id =
           id,user_id=request.user,data = request.data)
        return rubric_obj.exportData()
```

Listing 6.8: Classed Based API Call for AI Grading

```
    def put(self, request, id, criteria):
        # For ai based grading of rubric
        if 'predict' in (request.get_full_path()).lower():
            all_sub_obj = RubricService(program_id = id, data
               = request.data, prediction_criteria = criteria)
            all_sub_obj.predict_selected_submissions()
        return Response({"msg" : "200 OK"}, status=200)
```

6.2.3 Service

In this section we define the actual logic, which seperates the concern of data manipulation from the view and model layers. Here we retreive data from the database(through models) based on the parameters passed in URL `<str::id>`. This function in listing 6.9 is written in the service layer and is responsible for assigning an asynchronous task to predict the ratings for selected submissions based on given criteria. The exportData function present in the `service.py` is responsible for exporting data,rubrics, problem statements and student submission data, to an AI server. The process starts by retrieving relevant program activity data using program_ids and checks if the data has already been exported through an entry in the AIHistory model. If data has already been sent for a given program, the function returns a failure message to avoid redundant exports.

The function extracts the problem description from the program's description using `beautifulSoup` and retrieves rubric data, processing titles and ratings for each criterion. It then gathers student submissions, reads their source code, and retrieves both manual and AI grading history, consolidating all the data into a result dictionary for sending it to the AI server. The actual business logic for the exportData function is quite complex and extensive. Therefore, I've provided a simplified implementation of the Data Collector in Listing 6.10 below.

Listing 6.9: Rubric Service- Predict Grades Implementation

```
def predict_selected_submissions(self):
    ''' This function is to assign async task for predicting
       ratings for submissions for given criterion'''

    submissions_id = StudentSubmission.objects.
    filter(program_id=self.program_id,
       selected_for_evaluation=True).values_list('id',
       flat=True)
    criterias =
       RubricCriteria.objects.filter(program_id=self.program_id)
    criteria_data = RubricCriteriaSerializer(criterias, many
       = True).data

    for i, criteria in enumerate(criteria_data):
        criteria['title'] = Tag.objects.get(pk=
           criteria['title']).name
        ratings_list = RubricRating.objects.filter(criteria =
           list(criterias)[i])
        rating_data = RubricRatingSerializer(ratings_list,
           many = True).data

        for rating in rating_data:
            rating['title'] = Tag.objects.get(pk=
               rating['title']).name
        criteria['Ratings'] = rating_data

    ai_job_running = False

    try:
        codellama_auto_evaluation_prototype.
        delay(submissions_id, self.program_id, criteria_data)
    except Exception as e:
        StudentSubmission.objects.
        filter(id__in=batch).update(is_ai_grading_running=False)
    if(ai_job_running):
        return Response({"msg" : "Some submissions skiped"},
           status=200)
    return Response({"msg" : "Success"}, status=200)
```

Listing 6.10: Rubric Service- Export Data Implementation

```
def exportData(self):
    ''' This function exports the rubric, problem statement,
       and student submission data to the AI Server.'''
    result_dict = {}
    course_id = 0
    program_ids = self.data.getlist('program_id')
    user_id = self.user_id
    for program_id in program_ids:
        ai_history, created = AIHistory.objects.get_or_create(
            program_activity_id=program_id,
            defaults={'is_data_exported': False,
               'is_retraining_done': False}
        )

        if ai_history.is_data_exported:
            print(f"Data already exported for program_id
               {program_id}")
            return Response({"error": "Failed to send data",
               "details": "Data Already Exported"},
               status=500)
        extracted_text =
           self.extract_problem_description(program_id)
        rubric_list = self.extract_rubric_data(program_id)
        student_submissions =
           self.extract_student_submissions(program_id)

        try:
            if str(course_id) not in result_dict:
                result_dict[str(course_id)] = {}

            if str(lab_id) not in result_dict[str(course_id)]:
                result_dict[str(course_id)][str(lab_id)] = {}
            result_dict["user_id"] = str(user_id)
            result_dict[str(course_id)]
            [str(lab_id)][str(program_id)] = {
                "problem_statement": extracted_text,
                "file_name": "a.cpp",  # Replace with actual
                   file name if necessary
                "rubric": rubric_list,
                "student_submissions": student_submissions
            }
        except Exception as e:
            print(f"Unexpected error: {e}")
```

6.3 Communication and Service Layer at AIServer

The TaBuddy 1.0 provides functionality as a Django App. And doing this makes our app robust and provides functionality that can be used with other Django App. Having different app help us isolate their functionality and bring us closer to goal of making robust and efficient AI-Assisted autograder.

6.3.1 Database Layer

We will look at how a database table is built inside models.py and how we connect these models.To create a model, we must use OOPS in order to inherit the code that is already there in Django `from django.db import models`. *NOTE: Django inserts an id field for your tables, which is an auto incrementing number. It can be overridden by describing your own id field.*

Table 6.1: Problem Model

Field	**Description**
id	IntegerField (Primary Key) - Program Activity ID
problem_statement	TextField (Default: "No problem statement provided.")
user_id	CharField (Max Length: 255, Default: "Unknown User ID")
lab_id	IntegerField
totalDataPoints	IntegerField (Default: 0)
isTrained	BooleanField (Default: False)

Table 6.2: Submission Model

Field	**Description**
id	AutoField (Primary Key)
problem	ForeignKey to Problem (on delete: CASCADE)
student_id	CharField (Max Length: 255, Default: "Unknown Student ID")
dummy_id	CharField (Max Length: 255, Default: "Unknown Dummy ID")
filename	CharField (Max Length: 255, Default: "default_filename.cpp")
source_code	TextField (Default: "No source code provided.")

Table 6.3: Criteria Model

Field	**Description**
id	IntegerField (Primary Key)

Field	Description
problem	ForeignKey to Problem (on delete: CASCADE)
title	CharField (Max Length: 255, Default: "Default Criterion Title")
description	TextField (Default: "No description provided.")

Table 6.4: Rating Model

Field	Description
id	IntegerField (Primary Key)
title	CharField (Max Length: 255, Default: "Default Title")
description	TextField (Default: "No description provided.")
marks	IntegerField (Default: 0)
criteria	ForeignKey to Criteria (on delete: CASCADE)

Table 6.5: Grading History Model

Field	Description
id	AutoField (Primary Key)
submission	ForeignKey to Submission (on delete: CASCADE)
criteria	ForeignKey to Criteria (on delete: CASCADE)
manual_rating	ForeignKey to Rating (related name: 'manual_grading_histories')
ai_rating	ForeignKey to Rating (related name: 'ai_grading_histories')
manual_comments	TextField (Default: "No comments.")
ai_comments	TextField (Default: "No AI reasonings.")

6.3.2 URLs

In this section, we will look at how these URLs are linked to the API calls that will execute a service for us. If we look at listing 6.11, we can see that we have entered two URLs, and if we look at the right side of the URL, We can see that it's linked to an API call named DataPoint.

Listing 6.11: URLs implementation

```
urlpatterns = [
    path('llmpredict/',Predictor.as_view()),
    path('data-point/',DataPoint.as_view()),
]
```

6.3.3 Views

Django views are Python functions that handle HTTP requests and return HTTP responses, such as HTML documents. A web page built with Django consists of various views, each designed to perform specific tasks or functions. Typically, these views are organized in a file named `views.py`, which is located in the folder of your Django application. The DataPoint API view processes incoming POST requests by executing the post method, and this method contains the entire implementation of export data. The Predictor API Views has logics for handling POST and GET requests. When incoming data is received, the POST handler accepts the data and query codellama then return a taskID. As this task takes some minutes to process individual submission for a single criterion. So for an entire class it will take times in hours so make this task as a celery. Using the same taskID, the bodhitree backend performs get request after specific interval to fetch the ratings.

6.4 Celery Tasks

Bodhitree, A learning management system of IIT Bombay. Thousand of student of IIT bombay use it to learn programming by completing various lab, assignments and quizzes. Inorder to perform testcase-based evaluation, evalpro use Celery to execute student submssion code asynchronously on the backend server. This approach helps eliminate the need for the frontend to wait for the execution to complete, thereby reducing wait times for users. Similarly, to help the TAs/Instructor to manually grade the submission we introduced TA Buddy. TA Buddy uses Code Llama to analyze the submission and provide grades but this isn't done instantly. So to avoid TAs/Instructor to keep waiting after sending the request to perfom AI-Grading we used celery for it.

6.4.1 Celery : The Background Task Master

Celery is a distributed task queue that unlocks worker processes for Django, enabling it to collect, record, schedule, and perform tasks outside of the main program. Redis is the datastore and message broker between Celery and Django. In other words, Django and Celery use Redis to communicate with each other (instead of a SQL database). Redis can also be used as a cache as well.

6.4.1.1 Configure Celery in Django

We create a file named `celery.py` in the same directory as settings.py file. This file holds the celery configuration.

Listing 6.12: Celery Setup for AI BodhiTree

```
from __future__ import absolute_import, unicode_literals
import os
import torch.multiprocessing as mp
import atexit
import signal
import torch
import sys

mp.set_start_method('spawn', force=True)

from celery import Celery
from django.conf import settings

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
   'ai_bodhitree.settings')

app = Celery('ai_bodhitree')
app.conf.enable_utc = False
app.conf.update(timezone='Asia/Kolkata')

app.config_from_object('django.conf:settings',
   namespace='CELERY')
app.autodiscover_tasks()

@app.task(bind=True)
def debug_task(self):
    print(f'Request: {self.request!r}')

def cleanup(signum=None, frame=None):
    torch.cuda.empty_cache()
    print("Cleaned up resources.")
    sys.exit(0)

atexit.register(cleanup)
signal.signal(signal.SIGTERM, cleanup)
signal.signal(signal.SIGINT, cleanup)
```

6.4.1.2 Configure Celery Broker - Redis

We need to register REDIS with our django, so we add some lines in `settings.py`.

```
CELERY_BROKER_URL = 'redis://localhost:6379/0'
```

6.4.2 Create Celery Tasks: BodhiTree

On Bodhitree, we use Celery to handle the background tasks related to the grading of student submission via the AI Server. We use celery to execute the task asynchronously which means the task will be processed in the background without blocking the main program. It collects all the file_submitted, problem statement and rubrics which is sent to AI Server. After the grading request, it receives a task_id to track grading progress. This is another Celery task used to check the status of the grading request by sending a GET request to the AI server.

6.4.3 Create Celery Tasks: AI Server

On AI Server, Celery task is designed to grade student submissions using a pre-trained model called CodeLlama. It starts by ensuring the model and necessary tools are loaded. Then, it processes the submissions based on specific grading criteria, generating ratings and reasoning for each submission. After grading, it combines all the results into a single output while cleaning up temporary files created during the process. Finally, it returns a dictionary of graded results along with reasoning.

6.5 API Design

6.5.1 API Call For Inference

This API call is the actual functionality that will benefit the TA/Instructor with AI Suggested Ratings & Comments for each criterion.

6.5.2 API Call For Data Collector

This API call is for exporting the data to the AI Server through API-datadump. Whenever the instructor request for exporting data to AI Server, first of all fetch of data happens at BodhiTree Server side. Once it fetches everything, then there is a validation at Bodhitree server side. Then, it initiates a post request to AI Server

endpoint. The data is sent in a structured way so that at the another end-point the data can be easily restructured and stored in the DB. The format of data which is expected to be generated from the bodhitree backend which is going to be sent to AI Server within an API Call is shown in 6.5.2.1.

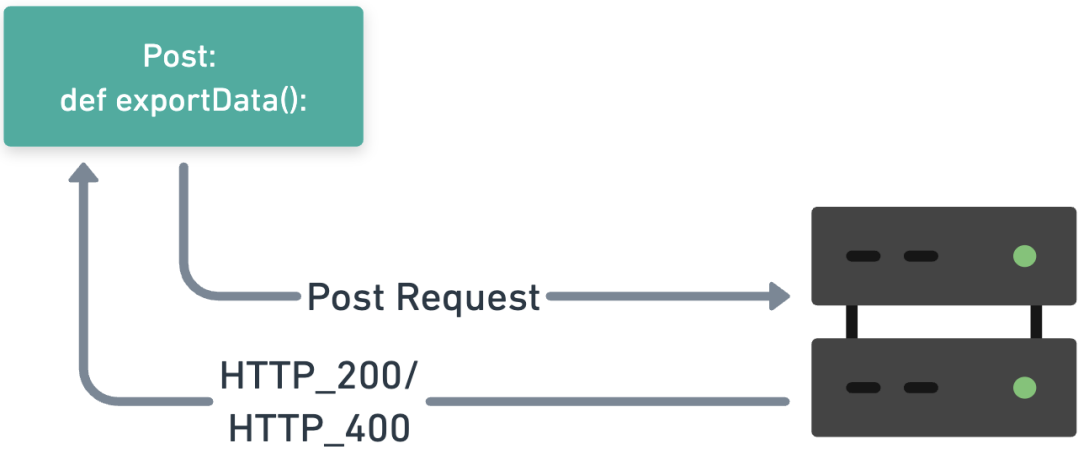

Figure 6.3: API Call For Data Collector

6.5.2.1 Structure of Data to Send To AI-Server:

```
    {
  "course_id": {
    "lab_id": {
      "program_id": {
        "problem_statement": "Problem Statement",
        "file_name": "a.cpp",
        "rubric": [
          {
            "criterion_id": {
              "title": "criterion_title",
              "description": "",
              "ratings": {
                "rating_id": {
                  "title": "rating_title",
                  "description": "rating_description",
                  "marks": 10
                }
              }
            }
          }
        ],
        "student_submissions": {
          "23m0838": {
            "source_code": "source_code",
            "manual_rating": {
              "criterion_id": [
                "rating_id",
                "Code is clean and follows conventions. Well-commented."
              ]
            },
            "ai_rating": {
              "criterion_id": [
                "rating_id",
                "High-quality code with excellent readability."
              ]
            }
          }
        }
      }
    }
  }
}
```

Figure 6.4: Structure Of Data Sent To AI Server

```
def post(self, request):
    if request.method == 'POST':
        print("the request got received\n")
        # Generate a unique task ID
        task_id = str(int(time.time() * 1000))
        # Create a directory for this task
        task_dir = os.path.join(settings.MEDIA_ROOT, task_id)
        os.makedirs(task_dir, exist_ok=True)
        # Create submissions folder
        submissions_dir = os.path.join(task_dir,
           'submissions')
        os.makedirs(submissions_dir, exist_ok=True)
        data = request.data
        # Save files
        [self.save_student_files(submissions_dir,
           request.FILES[id]) for id in request.FILES]
        # Save problem statement
        self.save_problem_statement(task_dir,
           request.data['problem_statement'])
        # Save criteria
        self.save_criteria(task_dir,
           request.data.get('criteria'))
        # Process the extracted files
        criterion_info = os.path.join(task_dir,
           'rubrics.json')
        context_path = os.path.join(task_dir, 'ps.txt')
        system_prompt_path = settings.SYSTEM_PROMPT_PATH
        output_path = settings.OUTPUT_PATH
        output_path = os.path.join(output_path, task_id)
        os.makedirs(output_path, exist_ok=True)

        try:
            task = query_codellama.apply_async(
                args=[system_prompt_path, context_path,
                   task_dir, output_path, criterion_info],
                kwargs={
                    'criterion_name': "",'max_length': 4096,
                    'few_shot': False,'few_shot_examples': 0,
                    'train_split': 0.7
                }
            )
            return Response({'status code': 202, 'task':
               task.id}, status=202)
        except Exception as e:
            print(e)
            return Response(status=404, data={'status': 404,
               'message': 'Something went wrong'})
```


Figure 6.2: Class Based API Call for Inferencing Task

Chapter 7

Features Added in Evalpro 2024 and AI4Code Project

7.1 Downloading Entire Lab along with Ratings and Comments

BodhiTree is an online learning platform developed at IIT Bombay, with the objective of making quality technical education widely accessible. The platform emphasizes personalized, flexible, and hands-on learning to enhance the educational experience for students and instructors alike. EvalPro (Evaluation of Programming Assignment) is a key module within BodhiTree. It supports the automated evaluation of programming assignments in languages such as C, C++, and Python. EvalPro utilizes a test-case-based grading system, where submitted code is evaluated against predefined test cases provided by instructors. Additionally, it includes a rubric-based grading system that incorporates an algorithmic identification model for subjective evaluation of assignments.

Training data is the foundation of Machine Learning Model. The data one uses to train their model must be of good quality inorder for their model to perform well. If the dataset is not good, your model may not be able to serve its intended purpose when you deploy it to a production environment. For our case, fine tuning of LLM, whenever we finetune we mend the LLM to align to our task so for this case also the dataset is very crucial.

In an educational context, it is essential for instructors to have access to all lab materials, including student submissions along with their ratings and comments. For AI researchers, such datasets are invaluable. These dataset can be treated as **golden**

dataset that can be used for training of AI models. This feature to download entire labs, complete with ratings and comments, was developed specifically to facilitate the needs of researchers.

This feature is integrated into the EvalPro module of BodhiTree, ensuring that lab data is accessible for instructional purposes and AI research.

7.1.1 Implementation Details

1. **Initialize Folder Paths:** Check if `self.archive_path` is `None`. If so, create an `archive_path` based on `user_id` and a default title `'root_folder'`. Remove the existing directory at `archive_path` if it exists. Create a `program_path` by appending `self.program_name` to `archive_path`. Define `zip_path` for future use.

2. **Create Program Directory:** Attempt to create the directory at `program_path`. If an error occurs, exit the method.

3. **Write Program Configuration:** If `output_format` is `'archive_program'` or `'sample_program'`: Open a file named `-program config.json` in write mode. Serialize `self.program` into JSON format and write it to the file.

4. **Export Rubric Data:** If `output_format` is `'archive_program'`: Create a `RubricService` instance and read rubric data. Write the rubric data to a `rubrics.json` file.

5. **Handle Submissions (if asked):** If `self.include_submissions` is `True`: Initialize a CSV file for rubric ratings and write the header rows (criteria titles, descriptions, rating titles, etc.). Create a dictionary to map criterion IDs to their descriptions. Fetch program and submission data from the database. Create a `submissions` directory within `program_path`.

6. **Process Each Submission:** For each submission: Create a directory for the student based on their email. Copy the submitted file to the student's directory. Initialize a list to track ratings. Read grades for the submission using another `RubricService` instance. Gather comments and ratings for each criterion: Update the ratings list based on fetched grades. Store comments in a dictionary. Write the gathered data to the CSV file(rubrics_rating.csv).

7. **Comments and Test Cases Mark Data:** Write overall comments to a `comments.json` file and testcases marks to a `grades.csv` file.

7.1.2 Folder Structure

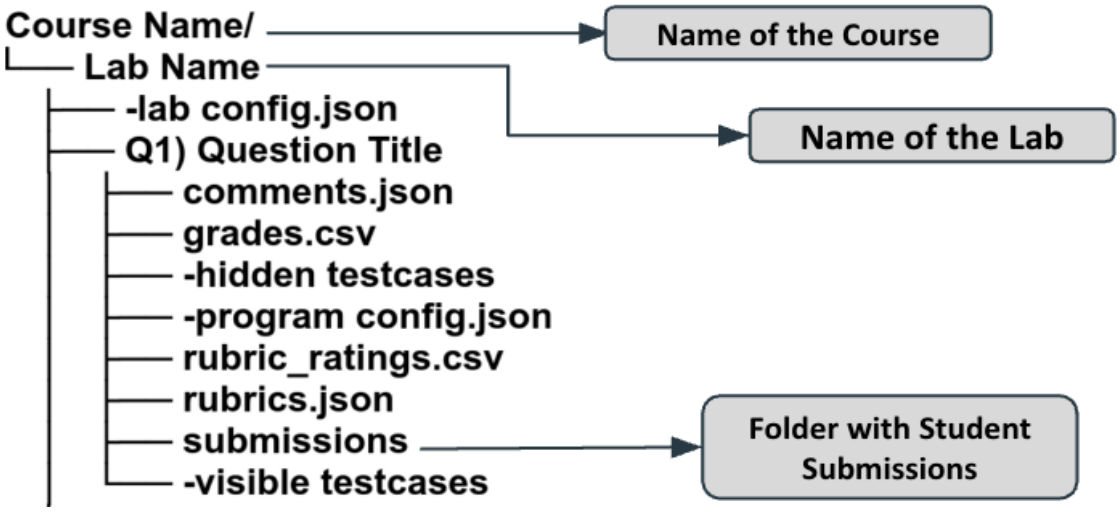

Figure 7.1: Folder Structure Call Of The Final Zip

7.2 AI Reasoning

Prior to this feature, one could only have AI Suggested rating for individual criterion. This limitation made difficult for the Instructor's/TA to unnderstand the reasoning behind the rating for that criterion, leaving them in doubt why LLM chose it. As a result this feature was required. So we had done some changes in the AI Server return value, instead of returning just the the rating we return a pair of rating along with reasonings. And in AI History Model, a new field comments was added where we store this reasoning.

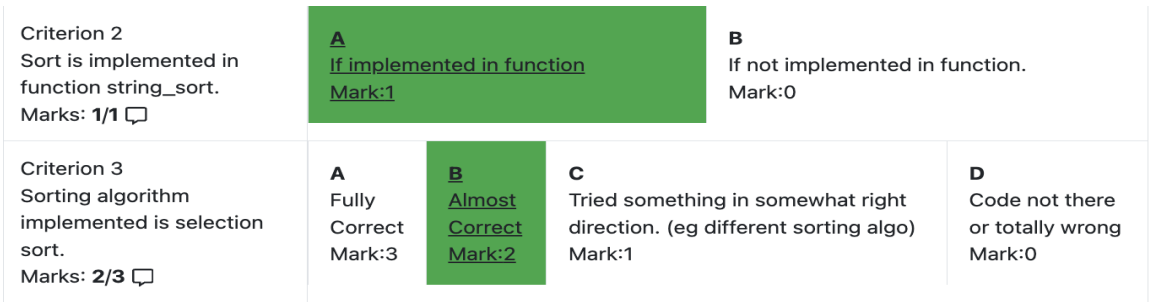

Figure 7.2: Before AI Reasoning Feature

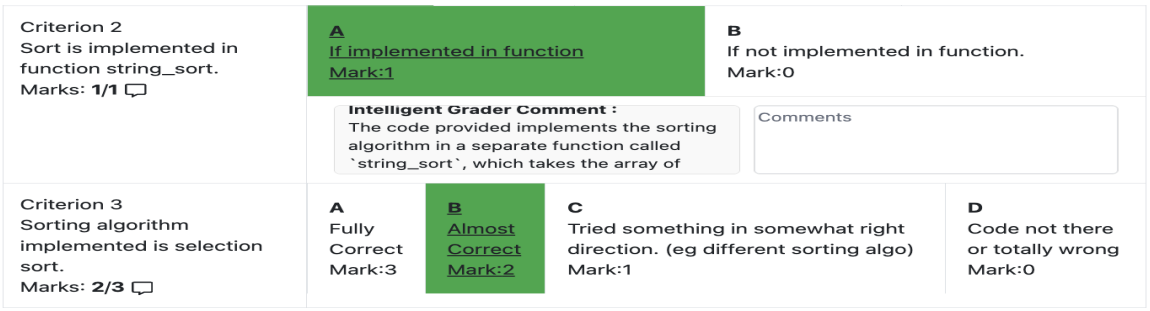

Figure 7.3: After AI Reasoning Feature

7.2.1 Implementation Details

Listing 7.1: AIGradingHistory Django Model

```
class AIGradingHistory(models.Model):
    id = models.BigAutoField(primary_key=True)
    student_submission = models.ForeignKey(StudentSubmission,
       on_delete=models.CASCADE)
    criteria = models.ForeignKey(RubricCriteria,
       on_delete=models.CASCADE)
    rating = models.ForeignKey(RubricRating,
       on_delete=models.CASCADE)
    evaluated_on = models.DateTimeField(auto_now=True)
    comments = models.TextField(default="Not Present",
       blank=True)
```

Listing 7.2: BodhiTree FrontEnd

```
{/* AI Comments Block */}
<div style={{
    padding: '10px',
    height: '80px', // Adjust the height as needed
    width: '50%',
    overflowY: 'scroll',
    backgroundColor: '#f9f9f9',
    border: '1px solid #ddd',
    borderRadius: '5px',
    fontSize: '14px',
    justifyContent: 'space-evenly',
}}>
    <span style={{
        display: 'block',
        fontWeight: 'bold',
    }}>
```

```
        Intelligent Grader Comment :
    </span>
    {aiGradingHistory[row.criteria_id]?.comments || "No
       reasoning provided."}
</div>
```

7.3 Bulk Submission Upload

Prior to this task, when we tried uploading submission 7.4 in Labs where two Labs had same programming activity, there was an issue uploading it. So we updated the script in populate_db_data module that takes an archive containing all the submissions organized lab wise i.e., each folder in the archive named with each lab having all its submissions.

```
Choose the option:
1. Create Student accounts
2. Create Instructor accounts
3. Create Courses
4. Modify number of students in a course using course id
5. Create labs using course id
6. Create or delete programs using lab id
7. Create Testcases using program id
8. Create submissions using program id
9. Add lab submissions to a course
10. Clear all AI evaluation
11. Delete database except for root user
12. Exit
Enter the option: 9
-------------------------------------------------------
Please enter course id to continue: 5
Please enter the path of the zip file: submissions.zip
Importing submissions for CS101F22_LE03_C_Q2-all-submissions.zip
Importing submissions for CS101F22_LE2_E_Q2-all-submissions.zip
Importing submissions for CS101F22_LE03_E_Q2-all-submissions.zip
Importing submissions for CS101A23_LQ01_D_Q4-all-submissions.zip
-------------------------------------------------------
```

Figure 7.4: UI For Bulk Submission Upload

The issue was identified that when we are uploading the submissions from the backend because the current code is just searching with the file name and hence it's unable to upload submissions in the next lab because there are two labs with same lab name and the code is simply choosing the first lab and pushing the submissions. `bodhi_demo/wsgi/bodhitree/populate_db_data/views.py`

```
for lab_sub in labs_subs:
    is_lab_present = False
    for lab_name in labs_names:
        if lab_sub.startswith(lab_name):
            is_lab_present = True
            lab_id = Lab.objects.filter(title=lab_name)[0].id
            break
    if not is_lab_present:
        print("Lab not present for "+lab_sub)
        continue
```

Figure 7.5: Issue

Solution : Apply the filter for course_id along with lab_name.

```
Choose the option:
1. Create Student accounts
2. Create Instructor accounts
3. Create Courses
4. Modify number of students in a course using course id
5. Create labs using course id
6. Create or delete programs using lab id
7. Create Testcases using program id
8. Create submissions using program id
9. Add lab submissions to a course
10. Clear all AI evaluation
11. Delete database except for root user
12. Exit
Enter the option: 9
--------------------------------------------------
Please enter course id to continue: 5
Please enter the path of the zip file: submissions.zip
Lab id is : 41  Course id is: 5
Completed Submission
Importing submissions for CS101F22_LE03_C_Q2-all-submissions.zip
Lab id is : 39  Course id is: 5
Completed Submission
Importing submissions for CS101F22_LE2_E_Q2-all-submissions.zip
Lab id is : 40  Course id is: 5
Completed Submission
Importing submissions for CS101F22_LE03_E_Q2-all-submissions.zip
Lab id is : 42  Course id is: 5
Completed Submission
Importing submissions for CS101A23_LQ01_D_Q4-all-submissions.zip
```

Figure 7.6: Fixed Bulk Lab Submission Upload

7.4 Weights & Biases Integration

Weights & Biases (W&B) is a powerful platform designed to enhance the productivity and efficiency of machine learning teams by enabling faster model development and optimization. By integrating a few lines of code, W&B allows practitioners to seamlessly debug, compare, and reproduce their models. It offers comprehensive tracking of key components such as model architecture, hyper-parameters, git commits, model weights, GPU usage, datasets, and predictions. Additionally, W&B fosters team collaboration by providing tools that allow for real-time sharing and comparison of

experiments, making it easier for teams to work together effectively and accelerate their model-building process.

7.4.1 Initialize W&B in FineTuning Code

To integrate Weights & Biases (W&B) into the DPO code, start by installing the W&B library using `pip install wandb`. Then, initialize W&B at the beginning of your script by calling `wandb.init()`, specifying the project name and relevant entity. You can track hyperparameters and configurations by logging them using `wandb.config.update()`. During the conversion process, you can log important metrics such as the conversion status and execution time using `wandb.log()`.

Listing 7.3: Experiment manager for logging to wandb

```
exp_manager:
  explicit_log_dir:
     /raid/ganesh/nagakalyani/autograding/taBuddy/logs
  exp_dir: taBuddy
  name: saurav_wandb_logs
  create_wandb_logger: True
  wandb_logger_kwargs:
    project: AI4Code
    name: code-llama-7b
  resume_if_exists: True
  resume_ignore_no_checkpoint: True
  create_checkpoint_callback: True
  checkpoint_callback_params:
    monitor: val_loss
    save_top_k: 10
    mode: min
```

7.5 Data Fetch Feature For R&D Researcher

The success of any AI model depends on the quality of the data used to train it. Good-quality data helps the AI make accurate predictions and ensures that the model works well in real-life situations. Data is the petrol of AI. A good quality data enhances model ability, accuracy and accelerates development. And by good quality data means , the data should be properly validated and accurate.

Just like any other AI project, our AI4Code project relies on high-quality data for effective model performance. This feature is designed to assist R&D researchers

by streamlining the entire process of transforming raw data into clean, high-quality data that can be directly fed into the model for fine-tuning.

The feature automates essential steps, including data pre-processing and data formation in the required format, significantly reducing the time researchers spend on scripting. By simplifying this workflow, researchers can focus more on refining the model and enhancing its performance, ultimately leading to better outcomes.

7.5.1 Options

Currently this Data Fetch Feature is available on DGX server with four options. :

7.5.1.1 Grading History

Grading history consists of:

- **Problem Statement**: The problem statement corresponding to the selected problem.
- **Student Submissions**: A dictionary mapping `student_id` to `source_code` (i.e., `studentSubmissions`).
- **Rubrics**: A dictionary containing the criteria descriptions and their associated ratings for the problem.
- **Manual Grades**: A dictionary mapping each `criterion_title` to a nested dictionary of `student_id` and their corresponding manual marks (i.e., `grades`).

7.5.1.2 Problem Statement

The problem statement corresponding to the given `problem_id`.

7.5.1.3 Criteria Along With Ratings

A complete set of rubrics, including criteria descriptions and their corresponding ratings, for an individual `problem_id`.

7.5.1.4 Student Submission Along With Ratings

This function provides detailed information for each submission related to a specific problem, including the submission's grading history with manual and AI ratings and comments (if available). If any ratings or comments are missing, it indicates that as well.

Chapter 8

Experiments

8.1 Dataset

In this experiment, we utilized the dataset from multiple offerings of CS101, where every student had to submit their lab assignments. Submissions were rated based on a variety of criteria such as code quality, use of proper variable names, and any additional features included. Each problem had specific criteria, and the submissions were generally rated across three levels of performance on each criterion, ensuring a comprehensive evaluation of both functional and qualitative aspects of the code. The table **??** represents the entire dataset.

Table 8.1: **Dataset Details**

No.	Problem Id	Problem Name	Concept	Submissions	Criteria
1	CP_00101_loop_cs101f22_LE01_A_Q1	Taylor Series	Loop	112	7
2	CP_00104_loop_cs101f22_LE02_B_Q1	Square root of a number	Loop	181	5
3	CP_00106_loop_cs101f22_LE2_A_Q2	Prime Number	Loop	155	7
4	CP_00107_sort_cs101f22_LE03_B_Q1	Sorting strings (insertion)	Sort, Func, Arr, Str	157	7
5	CP_00110_loop_cs101f22_LE01_C_Q2	Prime factors of a number	Loop	134	6
6	CP_00111_array_cs101f22_LE02_B_Q2	Majority element of array	Arrays	166	5
7	CP_00112_fun_cs101f22_LE02_C_Q1	Increasing/decreasing Array	Arr, Func	159	6
8	CP_00115_sort_cs101f22_LE03_C_Q1	Sorting strings (Bubble)	Sort, Func, Arr, Str	105	6
9	CP_00116_sort_cs101f22_LE03_E_Q1	Sorting strings (selection)	Sort, Func, Arr, Str	93	6
10	CP_00117_mat_cs101f22_LE03_B_Q2	Matrix transpose times matrix	Arr, Func	178	5
11	CP_00206_var_cs101s23_lq01_b_q2	Addition of (a/b) with (c/d)	Variables	157	5
12	CP_00208_var_LQ01_B_Q4	Calculate Combinations	Variables	146	4
13	CP_00211_condition_cs101s23_LQ01_C_Q3	Perform operation for given number	Conditional Statements	149	10
14	CP_00214_var_cs101s23_lq01_d_q2	Expression evaluation	Variables	158	6
15	CP_00216_var_cs101s23_LQ01_D_Q4	Leap year or Not?	Variables	129	6
16	CP_00315_var_cs101a23_lq01_c_q5	sin(x) calculation	Variables	135	6
17	CP_00105_sort_cs101f22_LE2_E_Q2	Selection sort implementation	Sort, Func, Arr, Str	140	11
18	CP_00108_loop_cs101f22_LE01_B_Q2	Print reverse of a number	Loop	178	11
19	CP_00113_sort_cs101f22_LE03_A_Q1	Sorting strings (merge sort)	Sort, Func, Arr, Str	131	10
20	cs101a23_LQ_01_D_Q4	Implementation of fmod function	Func, Operators	110	9
21	cs101a23_LQ01_b_Q4	Babylonian Algorithm	Loops	160	8
22	cs101a23_lq01_a_q2	Binary to decimal conversion	Loops, Variables	71	12
23	cs101s23_lq01_c_q4	Find Pythagorean triplets	Variables	136	7
24	CP_00102_loop_cs101f22_LE01_A_Q2	Check palindrome of Number	Loops, Variables	140	11
25	cs101a23_LQ01_D_Q3	Complex numbers operations	Var, Conditions	152	14
26	cs101a23_LQ01_A_Q5	LCM Computation	Variables	118	10
27	cs101a23_lq01_a_q3	Bisection method for root	Loops, Variables	75	10

We had access to student submissions from multiple offerings of CS101 (Computer Programming and Utilisation), the introductory programming course at IITB. Each offering had around 600-700 students per semester. We chose problems from lab exams conducted in this course. These lab exams had a wide variety of problems ranging from arrays to implementation of sorting algorithms. However, there was an issue with this dataset. For almost all the problems, the submissions were automatically evaluated using test cases considering the size of the class, thereby neglecting aspects of the code which can be graded only subjectively.

In order to make this dataset useful to our research, multiple grading exercise was conducted where TAs were recruited to grade these submissions according to some custom subjective criteria designed separately for each problem. These criteria were designed keeping in mind the requirements of the code specified in the problem statement. Common criteria like "Is the code well commented?", "Are the variable names good?" as well as problem specific unique criteria like "Are arrays being used?", "Is the correct function signature used for string compare?" were included. Instead of assigning real number of grades, the TAs had to pick a *rating* for each criterion from some pool of ratings ranging from good to bad. For example, for the criterion "Is the code well commented?", the set of ratings can be A. Well commented, B. Ok commented and C. Poorly commented. The grading was done in one coordinated session, to ensure consistency.

8.2 DPO FineTuning

DPO as mentioned in section 2.6.2 is a technique which directly optimizes the model to improve the likelihood of the preferred responses to the likelihood of the rejected responses. We need a performance dataset of the form $x^i, y_w^i, y_{l\,i=1}^{iN}$ where y_w and y_l denote chosen and rejected responses for the input x respectively. For a given criterion with N ratings, we can create $N-1$ preference pairs for a given student submission. In each of these pairs, the "chosen" field will be the response assigning the ground truth grade to the submission and the "rejected" field will be the response assigning any grade other then the ground truth grade. Every data point in the fine-tuning dataset consists of three fields, "prompt"(the input prompt for performing inference with the LLM), "chosen"(response in JSON format assigning the ground truth grade)

and "rejected"(response in JSON format assigning any other grade). Figure shows an example demonstrating the data format for DPO.

For fine-tuning **Code Llama-7B**, we use a portion of the dataset as the fine-tuning dataset. In our experiments, we use two approaches. In the first approach we chose 21 problems for fine-tuning the models. Even within the 21 problems, only 70% of the submissions were used for fine-tuning. 30% of the submissions from the above mentioned 21 problems were reserved for testing. This approach is unseen problems. In the second approach we fine-tune the model with entire 70% of dataset. And evaluate on rest of the 30% of the data. In this case the entire data is seen, so its seen case. The intention behind doing this was to get an idea about the behaviour of the two approaches.

```
training_args = DPOConfig(
    per_device_train_batch_size=4,
    per_device_eval_batch_size=1,
    num_train_epochs=1,
    logging_steps=1,
    save_steps=0.2,
    gradient_accumulation_steps=4,
    gradient_checkpointing=True,
    learning_rate=1e-5,
    evaluation_strategy="steps",
    eval_steps=0.5,
    output_dir=args.output_dir,
    report_to="tensorboard",
    lr_scheduler_type="cosine",
    warmup_steps=100,
    optim="paged_adamw_32bit",
    bf16=True,
    remove_unused_columns=False,
    gradient_checkpointing_kwargs=dict(use_reentrant=False),
    seed=0,
)
```

Table 8.2: DPOTraining Args

```
peft_config = LoraConfig(
    r=args.lora_r,
    lora_alpha=args.lora_alpha,
    lora_dropout=args.lora_dropout,
    target_modules=[
        "q_proj",
        "v_proj",
        "k_proj",
        "out_proj",
        "fc_in",
        "fc_out",
        "wte",
    ],
    bias="none",
    task_type="CAUSAL_LM",
)

dpo_trainer = DPOTrainer(
    model,
    ref_model=None,
    args=training_args,
    beta=0.1,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    peft_config=peft_config,
    max_prompt_length=2048,
    max_length=1536
)
```

Table 8.3: Combined LoraConfig and DPOTrainer Settings

Data format for DPO

{"prompt": "<s>[INST] <<SYS>> Your task is to choose the MOST suitable option among a set of options I provide, about a code which will also be provided. Give your output as a json with a single field ‘answer’. Do not output anything else. Strictly follow this output format at any cost.
<</SYS>>
Context : Write a C++ program that determines the approximate square root of a given number S up to a desired precision epsilon. The program should take S and epsilon as input and print the value of the square root up to 5 decimal places. The Babylonian method is used to find the square root. The method starts with an initial guess and iteratively improves the guess until convergence is achieved. The program should terminate when the absolute difference between the current guess and the previous guess falls below epsilon.
Code :
#include<iostream>#include<cmath>using namespace std;double sq_root(double S, double epsilon){double x=2;while(fabs(x/2-S/(2*x))>epsilon){x = x/2+S/(2*x);}return x;}int main(){double S, epsilon;cin>>S>>epsilon;cout<<fixed; cout.precision(5);cout<<sq_root(S, epsilon);}
Task :
Choose the option which is most suitable for the above code for the criterion ‘Whether babylonian method is implemented in a function’. Give your output as a json with a single field ‘answer’. Do not output anything else. Strictly follow this output format.
Options :
A. Separate function is used and function signature is correct
B. Separate function is used but function signature is not correct
C. totally wrong
Response : The required ouptut in json format is :
[/INST] {"answer" : ‘A. Separate function is used and function signature is correct’} </s>", "chosen" : {"answer" : ‘A. Separate function is used and function signature is correct’}, "rejected" : {"answer" : ‘C. totally wrong’} }

Figure 8.1: An example showing the data format for DPO

8.3 Dora FineTuning

In recent years, Low-Rank Adaptation (LoRA) has become a popular method for fine-tuning large language models. It works by breaking down the weight updates into smaller, more manageable pieces, allowing models with billions of parameters to be fine-tuned efficiently. However, a new technique called DoRA (Dual-Rank Adaptation) has been developed to improve upon LoRA, especially in terms of accuracy, while keeping the same efficiency. DoRA is designed to get closer to the accuracy of full fine-tuning (FFT) without adding extra complexity to the model's inference process.

DoRA uses a dual approach, separating the fine-tuning process into two parts: one focuses on directional updates, and the other on controlling the strength (or magnitude) of those updates. By handling these separately, DoRA can make more precise adjustments to the model, leading to better performance across different tasks while still being computationally efficient. After fine-tuning both parts, DoRA brings them back together to form the final weight matrix.

This method has several advantages:

1. **More Focused LoRA Adjustments:** By letting LoRA only handle directional updates, DoRA makes the tuning process more efficient and targeted.

2. **Increased Stability:** Separating the directional updates helps reduce training instability, making the process smoother.

3. **Finer Control:** By independently tuning the magnitude, DoRA allows for more precise adjustments to the model, improving its performance across tasks.

DoRA has been shown to consistently outperform LoRA in many tasks involving large language models and vision-language models. For example, it has led to significant improvements in common-sense reasoning, multi-turn dialogue benchmarks, and image/video-text understanding. In tasks like compression-aware language models and text-to-image generation, DoRA has also demonstrated its effectiveness. This work was accepted as an oral presentation at ICML 2024, which had a highly competitive 1.5% acceptance rate.

In short, DoRA combines the accuracy of full fine-tuning with the efficiency of LoRA, making it a strong choice for fine-tuning large models across various applications.

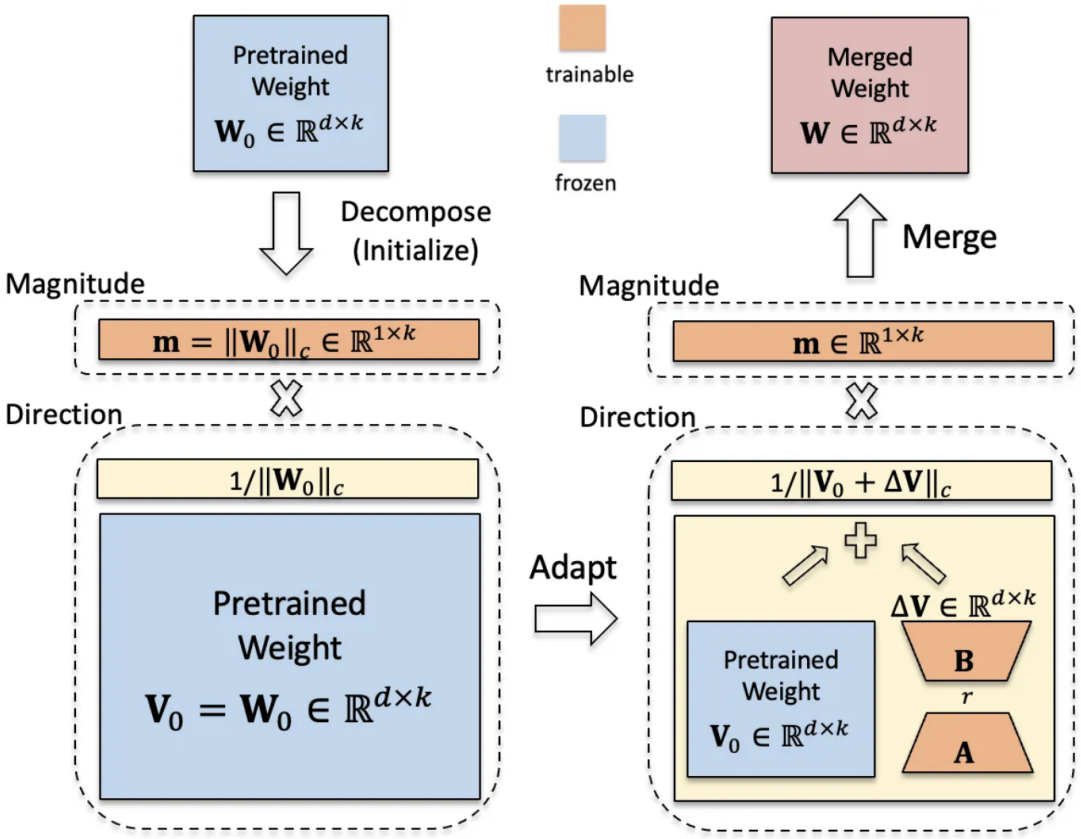

Figure 8.2: Working of DORA

	DPO With Dora	**DPO Without Dora**
Overall Accuracy	0.7047	0.6800
Precision	0.5684	0.5950
Recall	0.5768	0.5800
F1 Score	0.5725	0.5900
Macro Accuracy	0.7094	0.6900

Table 8.4: DPO Comparison for Seen Data

	DPO With Dora	**DPO Without Dora**
Overall Accuracy	0.6537	0.6300
Precision	0.5773	0.5450
Recall	0.5517	0.5350
F1 Score	0.5642	0.5400
Macro Accuracy	0.6494	0.6300

Table 8.5: DPO Comparison for Unseen Data

Chapter 9

Field Trial

9.1 DataSet Collection

As discussed in earlier sections, the dataset has been an integral part of this tool. However, a dataset of this type isn't easily available online. Even though we had data from multiple offerings of CS101, the data were raw and couldn't be used directly. Therefore, we had to conduct multiple grading melas to obtain a high-quality dataset.

A Grading Mela focuses on cleaning the dataset and establishing ground truth solutions for it. We conducted three grading melas, collecting 3,725 submissions across 27 problems. Over 30 TAs, either they were previously TA in the CS101 course or has attained a AA grade, were hired to grade these submissions.

9.1.1 Grading Mela 1.0 & 2.0

9.1.1.1 PRE-Grading Mela

1. Problem statements are selected.
2. Well-defined criteria and ratings, covering all functional and qualitative aspects of the code, are created.
3. The selected problem statements and rubrics are configured on the BodhiTree interface.
4. Student submissions are uploaded through the backend, ensuring they remain anonymized.

9.1.1.2 During Grading Mela

1. TAs are assigned an equal number of submissions to grade.
2. TAs are given a record sheet to track the time they spend on grading.
3. A Supervisor TA is assigned to each TA to assist with any issues.
4. The Supervisor TA maintains a master sheet to record the number of submissions graded by each TA during every hour interval.

9.1.1.3 Post Grading Mela

1. All graded submissions from the BodhiTree interface are exported.
2. Preprocessing is performed on the submissions and ratings to convert the data into the desired format.
3. The processed data is used to fine-tune the model.

9.1.2 Grading Mela 3.0

This grading mela was different from the previous two. It served as both a field trial for dataset collection and a tool testing exercise. To ensure an efficient evaluation of the tool, we aimed to fairly compare the grading times between AI-assisted grading and manual grading.

For each Programming Activity submission, the dataset was divided into two splits: X and Y. Suppose two TAs, TA1 and TA2, were assigned to this activity. In the first half, TA1 performed manual grading on the X-Split, while TA2 conducted AI-assisted grading on the same X-Split. In the second half, the roles were reversed for the Y-Split, with TA1 performing AI-assisted grading and TA2 handling manual grading. This setup allowed for a balanced comparison between both grading methods.

9.2 Evaluation of AI-Assisted Grader - TaBuddy

The purpose of this evaluation was to assess the usefulness of TA Buddy, our AI grader assistant, in comparison to traditional manual grading methods. By examining how TA Buddy performed in real grading scenarios, we aimed to determine its effectiveness and efficiency in assisting human teaching assistants (TAs) during the grading process.

We formulated three main questions that would help us understand the impact of TA Buddy on the grading experience:

1. **Acceptance Rate of AI-Suggested Grades:**
    - **Question:** What is the acceptance rate by the human TAs for the AI Assistant TA Buddy's suggested grades?
    - **Explanation:** This question aims to measure how often the human TAs agreed with the grades suggested by TA Buddy. For instance, if TA Buddy suggested a rating A for a Criterion X for a particular submission, we wanted to see how many times the human TAs accepted that suggestion without making any changes.
2. **Time Savings in Grading:**
    - **Question:** Overall, taking into account the time spent verifying and possibly overruling the AI-suggested grades, does TA Buddy save total person-minutes spent in manual grading? If so, how much time does it save?
    - **Explanation:** This question focuses on the efficiency aspect of using TA Buddy. We wanted to determine whether TA Buddy helps reduce the total time human TAs spend on grading assignments by comparing the total time spent with and without TA Buddy.
3. **Comparison of Grading Ratings:**
    - **Question:** How different are the ratings selected by the AI-assisted human grader compared to those selected by the purely manual human grader?
    - **Explanation:** This question aimed to investigate the differences in grading outcomes between the AI-assisted method and traditional manual grading by comparing the grades assigned by human TAs who used TA Buddy with those assigned by TAs who graded without AI assistance.

9.2.1 Experiment Design

The primary aim of this experiment was to evaluate the effectiveness of TA Buddy, an AI-assisted manual grading tool integrated into the BodhiTree LMS. This evaluation involved analyzing its impact on grading consistency, accuracy, and efficiency when assessing programming assignments in CS101. To conduct the evaluation, we selected a diverse set of six programming problems from past CS101 offerings, ensuring that these problems were different from those used during the training of TA Buddy. This approach was essential to minimize bias and ensure a fair assessment of the tool's performance. In total, 740 student submissions were analyzed across these problems (as detailed in (Table 9.1). For each selected programming problem, we developed detailed grading rubrics that captured the specific source code grading guidelines set by the instructor.

Table 9.1: Summary Statistics of the Evaluation Dataset

Problem	Problem Complexity	No. of Submissions	No. of Criteria	Avg No of Lines
#1: Evaluating Expressions	Easy	150	6	11
#2: Reversing Numbers	Easy	150	11	27
#3: Binary To Decimal	Moderate	70	12	31
#4: Complex Number Operations	Difficult	150	14	48
#5: Sorting Strings	Difficult	70	6	32
#6: Strictly Increasing/Decreasing	Moderate	150	6	17

This problem set was configured once each in two separate courses on our BodhiTree Platform. The 740 submissions were uploaded into each course, resulting in two copies of the entire evaluation dataset. One course (C1) was used for pure manual grading, while another course (C2) was designated for AI-assisted grading. In both C1 and C2, we first ran the conventional test case autograding. In C2, the Code LLAMA-powered TA Buddy was utilized to obtain suggested ratings for all the criteria across all submissions using the inferencing pipeline. We recruited fifteen TAs (graduate students, some of whom had been past CS101 TAs) to grade the submissions in both courses, C1 and C2. Each submission was graded twice: once in a 'pure' manual manner by a Grader TA in C1, and once by a different Grader TA in C2, who had access to the AI Assistant TA Buddy's suggested grades and could either confirm or overrule those grades. Each TA graded approximately the same number of submissions.

To ensure a fair comparison between the grading time of AI-Assisted Grading and Manual Grading, each Programming Activity submission was divided into two splits, X and Y. Two TAs, TA1 and TA2, were allocated for this activity. In the first half, TA1 performed manual grading for the X-Split, while TA2 conducted AI-Assisted grading for the same X-Split. In the second half, the roles were reversed: TA1 graded the Y-Split using AI assistance, and TA2 performed manual grading for the Y-Split. This approach allowed for a direct comparison of the grading times associated with both methods. The grading sessions were conducted under the supervision of our R&D team, which ensured that the timings for this work were meticulously maintained. After the grading process, the results from both courses were downloaded using the bulk download feature developed by us, which is accessible through the web interface. Thus, the data required to answer the first three evaluation questions was readily available from this setup.

9.2.2 Results

As shown in Figure 4, compared with pure manual grading time, which is normalized to 100%, AI-assisted grading was observed to require less time for all problems, with a reduction in grading time ranging from 7.40% to 45.23%. This indicates that AI-assisted grading can streamline the grading process effectively. It is important to note that the AI-assisted grading time includes the duration taken by a TA to inspect the student source code, verify whether the AI-generated grade is accurate, possibly select another grade if discrepancies are found, and write a different comment.

Table 9.2: AI-Suggested Grades (AISG) compared with AI-Assisted Grades (AIAG) and Pure Manual Grades (PMG)

	AIAG = AISG	**AIAG = PMG**
#1	79.81%	91.95%
#2	88.48%	90.31%
#3	85.85%	91.99%
#4	84.82%	94.64%
#5	60.55%	87.38%
#6	67.91%	87.12%
Average	**77.57%**	**90.55%**

Table 9.2 presents two key comparisons related to the grading process. The first comparison is between the AI-suggested grade provided by TA Buddy and the final AI-assisted grade selected by the teaching assistants (TAs) who graded submissions

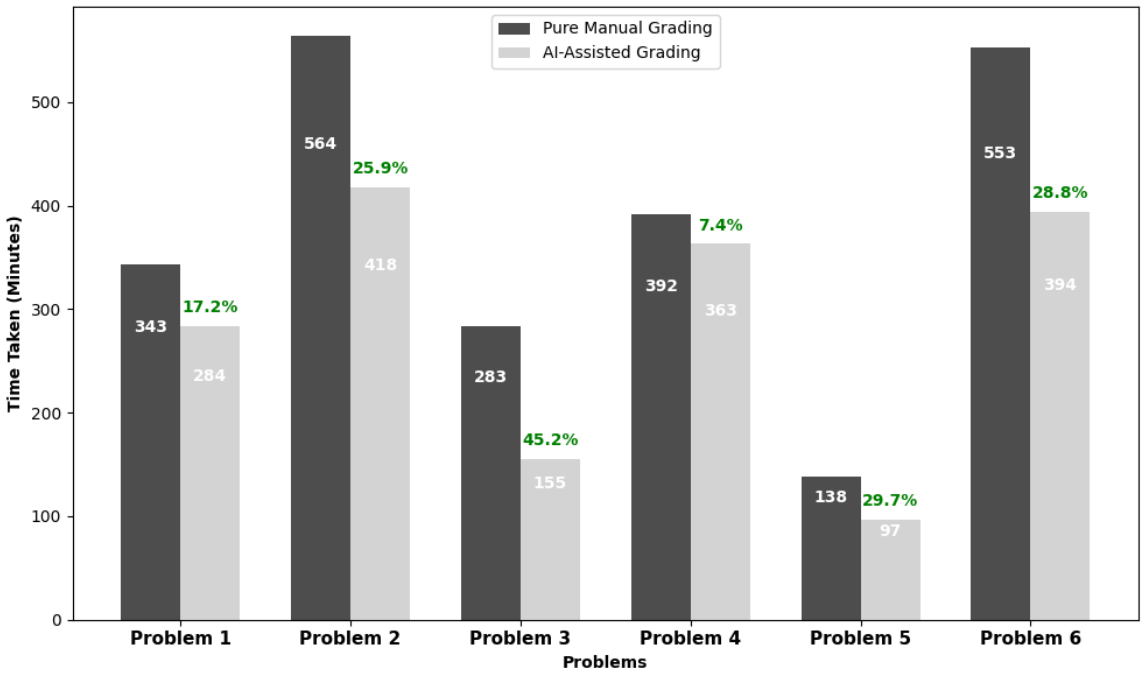

Figure 9.1: Comparison of Time required by Grader TAs. Pure Manual Grading Vs AI-Assisted Manual Grading

in course C2. This comparison aims to evaluate the extent to which TA Buddy is helpful to the TAs during the grading process.

The second comparison assesses the AI-assisted grade against the pure manual grade. This measures the influence that the visibility of the AI-suggested grades has on the TAs' grading decisions. The results indicate that the Grader TAs agreed with the suggested grade from TA Buddy only 60% of the time in the worst-case scenario, while in the best-case scenario, this agreement rose to 88%.

Despite this variability in agreement with the AI-suggested grades, it is noteworthy that the final grade selected by the AI-assisted grader TAs aligned with the pure manual grade an average of 90% of the time. This finding confirms that the suggested grades from TA Buddy do not unduly influence the Grader TAs' final choices, indicating that they maintain their independent judgment when determining grades.

9.2.3 Survey & Feedback

Apart from objective criteria of suggested grades accuracy and time, do the grader TAs *like* the overall flow of manual grading with AI-suggested grades?

Finally, the Grader TA survey results reported in Table 9.3 firstly measure perceptions of rubric-based grading. The first score indicates that TAs find rubric-based grading well-suited for evaluating a variety of submissions. It evaluate perceptions of rubric-based grading. The findings indicate that TAs believe rubric-based grading is well-suited for assessing a variety of submissions. In terms of AI assistance, nearly all graders expressed a preference for AI-assisted grading over manual grading, citing its ease of use and increased efficiency, with scores leaning towards "Agree" or "Strongly Agree." Overall, the satisfaction score for the AI Grader reflects that the tool was both effective and satisfying for its users.

For the survey, TAs were presented with a series of questions and asked to rate the statements on a Likert scale. Additionally, they provided a User Effort Score based on the ease of use and a Satisfaction Score reflecting their overall experience with the tool.

Table 9.3: Survey questions were presented to TAs and were asked to rate the statements on a Likert Scale, and then provide a User Effort Score based on its ease of use and Satisfaction score for their experience with the tool.

Likert Scale (Strongly Disagree = 1, Disagree = 2, Neutral = 3, Agree = 4, Strongly Agree = 5)	
Statement	***Avg. Score***
Rubric-based grading is flexible enough for grading a variety of submissions.	4.13
AI-assisted grading is better than Manual Grading for programming assignments.	3.93
AI-assisted grading is efficient and places less burden than Manual grading for programming assignments.	4.27
AI-assisted grading provides accurate evaluations of programming assignments.	3.60
User Effort Score (1 - 5 inclusive)	
How would you rate the Intelligent Grader Tool based on its ease of use?	4.2
Satisfaction Score (1 - 5 inclusive)	
What is your feedback regarding the Intelligent Grader tool based on your first-hand experience from the usage?	4.0

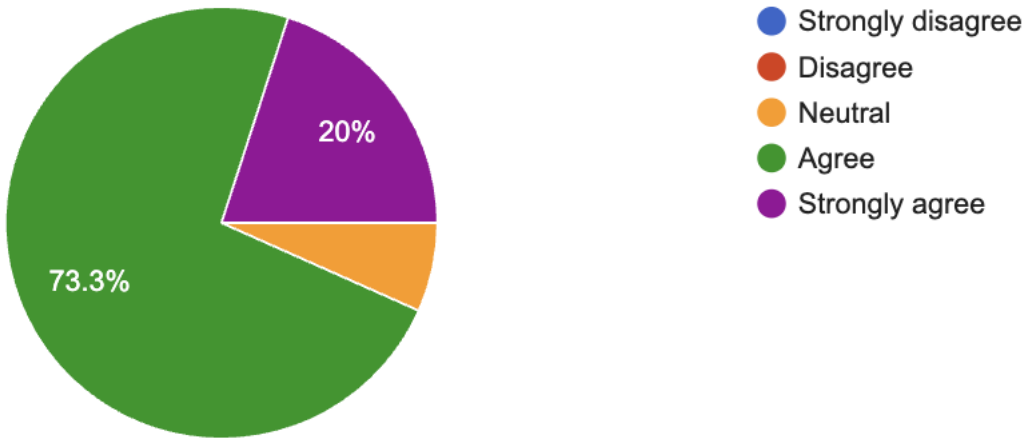

Figure 9.2: Rubric based grading is flexible enough for grading variety of submissions

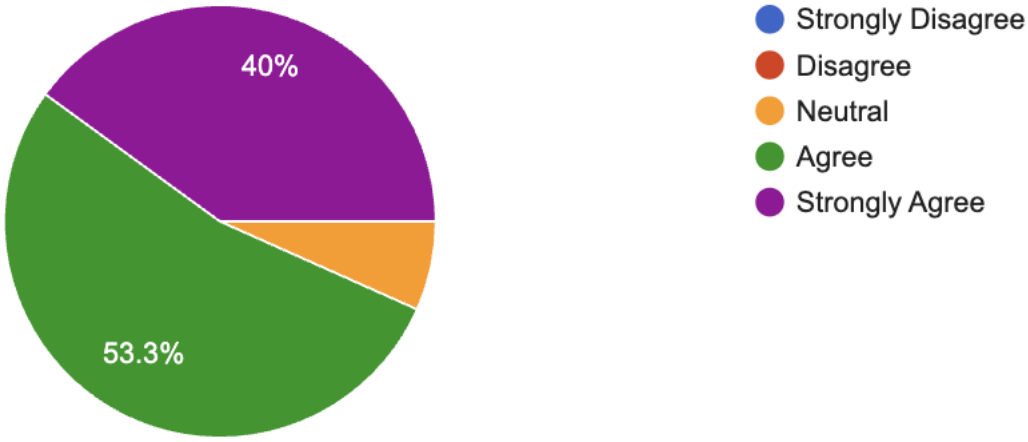

Figure 9.3: Rubric based grading facilitate efficient and fair assessment of student work

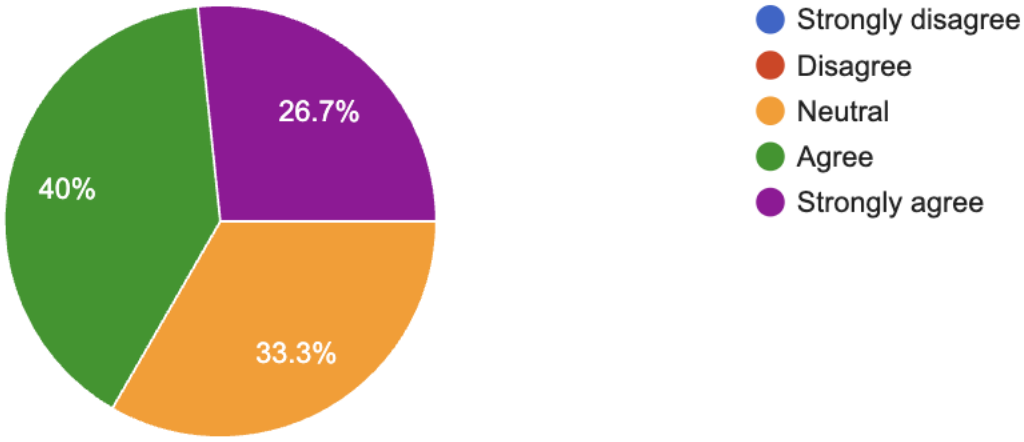

Figure 9.4: AI-assisted grading is better than Manual Grading for programming assignments.

9.3 Replica CS101 Lab Quiz Trial - TaBuddy

The purpose of this trail was to check all the new features such as Lab Replication, Entire Lab Download, Bulk Submission Upload, Inferencing and Data Export. We will discuss each in brief in the upcoming sections.

9.3.1 Entire Lab Download

At start, we need to download the labs which need to be replicated. The feature was present from before but due to some merge conflicts the feature wasnt working. So we worked on fixing those and introduced a new json file with it `comments.json` which provides the TAs/instructor and Intelligent Grader comments which was needed by AI4Code Project Researcher team.

9.3.2 Lab Replication

Currently, BodhiTree-Evalpro had option to upload entire lab at once but it didnt support those labs which were configured in exam mode. So we had to perform some changes in `lab-config.json` manually.

9.3.3 Bulk Submission Upload

After the lab is completely setup, and the problem statements are live. We need to have the student submission so we can do grading for it. Submitting one by one can be very time consuming, so with this feature. We can simply upload all the student submission from the backend and even they are anonymous.

9.3.4 Evaluation

We performed evaluation of the submission by inferencing with a Code LLama-7b Model trained on entire dataset of 27 labs which was dicussed above. We use DPO finetuning approach to fine tune the Code LLama-7b model.

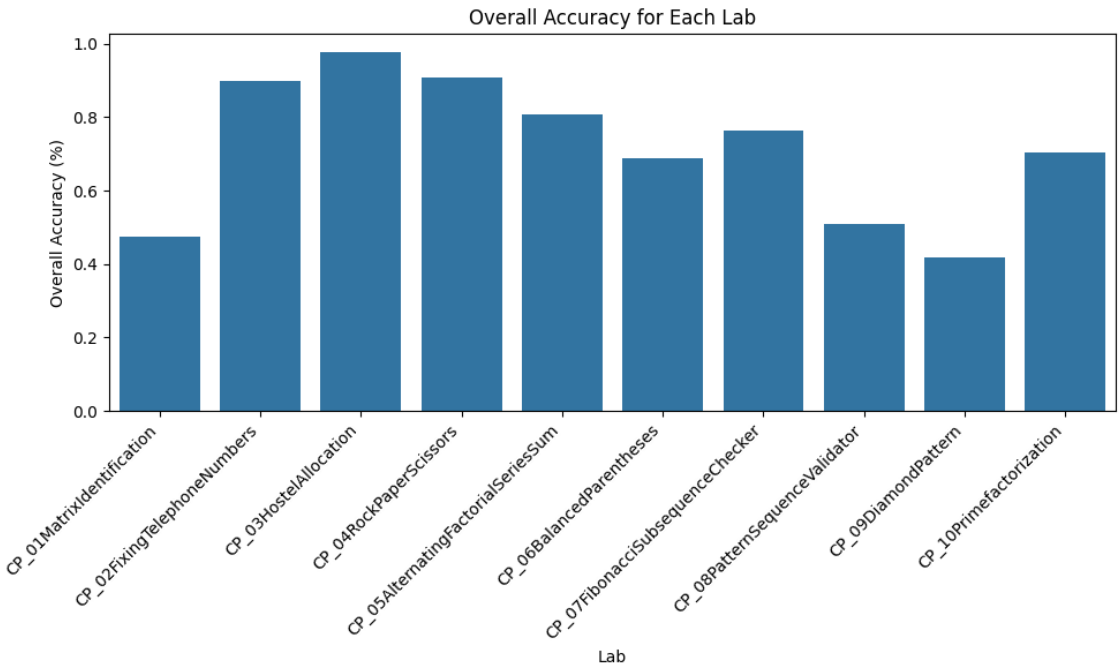

Figure 9.5: OverAll Accuracy

9.3.5 Analysis

From the above accuracy figure, we can conclude that its performing average. For some its worse such as Diamond Pattern, Matrix Identification but the reason is that question is something new and model hasnt so much idea about it. Because even in original dataset, the count of people achieving good marks are even low. And the other reason can be for the model to perform poor is the quality of rubrics.

Chapter 10

Conclusion and Future Work

In this work, we focused on the development and integration of TA-Buddy, an AI-assisted grading tool, into the BodhiTree-Evalpro. The core objective was to save Instructor's/TA grading time by providing AI suggested ratings for student submissions.

The integration of TA-Buddy into BodhiTree EvalPro streamlined the grading process by allowing Teaching Assistants (TAs) to leverage AI-based suggestions for rubric ratings based on student submissions. By utilizing pre-trained large language models (LLMs) fine-tuned with Direct Preference Optimization (DPO), TA-Buddy is able to provide reliable ratings even for unseen problem statements and rubrics.

In addition to the inference, a significant part of this integration involved the design and implementation of data collection and automated retraining pipelines. The data collection pipeline gathers student submissions, rubric feedback, and grading data, creating a rich dataset for continuous improvement of the AI model. The retraining pipeline triggers at regular intervals, retraining the model as more data accumulates, thereby ensuring the model's accuracy improves over time.

10.1 Future Work

There are several ways to improve TA Buddy, including making the system more automated, finding better models, using more diverse data and implementing robust authorization and authentication systems. These improvements would help make TA Buddy secure, faster, easier to use, and more accurate.

10.1.1 Benchmarking Automation

Without automation, the evaluation of checkpoints during fine-tuning presents several challenges that severely impact efficiency and productivity.One major issue is the need for manual monitoring and execution. Every time a checkpoint is saved,human intervention is required to choose and evaluate the best model.

10.1.2 Lighter & Faster Model

Try finding a model which take less storage, inference and training time than Code Llama-7B.

10.1.3 Enhancing Dataset Diversity

TA Buddy could be improved by focusing on gathering diverse and comprehensive datasets that represent a wide variety of programming assignments and student submissions. By incorporating data of various programming labs, the system can minimize potential biases that may arise from overfitting to a narrow dataset.

10.1.4 Integrating Visualization Tool for Real-Time Model Monitoring

In the future, we could improve how we track the performance of our machine learning models by using Grafana[19] or any other visualization tool, a tool that creates real-time dashboards. With Grafana, we can display important statistics like accuracy, precision, and error rates in easy-to-read graphs and charts. This would make it much simpler to monitor how well a model is performing without needing to dive into complex data.

10.1.5 Adding Authorization and Authentication

Currently, the AI Assistant autograder is set up on an AI server and connected to the BodhiTree LMS through a Django-based REST API.However, to make the system more robust and secure for broader use, it is essential to implement proper authorization and authentication mechanisms.

Bibliography

[1] Janet Rountree Anthony Robins and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[2] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[3] Youri Voet, Thomas Schaper, Olmo Kramer, Devin Hillenius. Codegrader, 2017. `https://www.codegrade.com/`.

[4] Pieter Abbeel, Arjun Singh, Ibrahim Awwal, Sergey Karayev. Gradescope, 2014. `https://www.gradescope.com/`.

[5] Veronica Cateté, Erin Snider, and Tiffany Barnes. Developing a rubric for a creative cs principles lab. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 290–295, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Brian Whitmer and Devlin Daley. Canvas, 2008. `https://www.instructure.com/canvas/login/`.

[7] Bodhi tree. In *BodhiTree documentation https://wiki.bodhi-tree.in/.*

[8] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

[10] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

[11] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. *arXiv preprint arXiv:2402.09353*, 2024.

[12] Tom Christie. Django rest framework, 2011. Version 3.x.

[13] Kennedy Ochilo Hadullo and Daniel Makini Getuno. Machine learning software architecture and model workflow: A case of django rest framework. *American Journal of Applied Sciences*, 2021. Open access article distributed under a Creative Commons license.

[14] Jan Koskinen. Designing a machine learning pipeline with continuous training for time series forecasting. 2024.

[15] P Liang, B Song, X Zhan, Z Chen, and J Yuan. Automating the training and deployment of models in mlops by integrating systems with machine learning. appl. comput. eng. 2024, 67, 1–7.

[16] XPNDAI. Traditional ml pipeline vs. generative ai pipeline, 2024. Available at: `https://xpndai.com/traditional-vs-generative-ai-pipeline`.

[17] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2023.

[18] Neptune AI. Retraining your model during deployment: Continuous training and continuous testing, 2023. Accessed: 2024-10-09.

[19] Grafana Labs. *Grafana: The open observability platform*, 2024. `https://grafana.com/`.