# Test Case-Informed Knowledge Tracing for Open-ended Coding Tasks

Zhangqi Duan
University of Massachusetts Amherst
Amherst, MA, USA
zduan@cs.umass.edu

Nigel Fernandez
University of Massachusetts Amherst
Amherst, MA, USA
nigel@cs.umass.edu

Alexander Hicks
Virginia Tech
Blacksburg, VA, USA
alexhicks@vt.edu

Andrew Lan
University of Massachusetts Amherst
Amherst, MA, USA
andrewlan@cs.umass.edu

## Abstract

Open-ended coding tasks, which ask students to construct programs according to certain specifications, are common in computer science education. Student modeling can be challenging since their open-ended nature means that student code can be diverse. Traditional knowledge tracing (KT) models that only analyze response correctness may not fully capture nuances in student knowledge from student code. In this paper, we introduce Test case-Informed Knowledge Tracing for Open-ended Coding (TIKTOC), a framework to simultaneously analyze and predict both open-ended student code and whether the code passes each test case. We augment the existing CodeWorkout dataset with the test cases used for a subset of the open-ended coding questions, and propose a multi-task learning KT method to simultaneously analyze and predict 1) whether a student's code submission passes each test case and 2) the student's open-ended code, using a large language model as the backbone. We quantitatively show that these methods outperform existing KT methods for coding that only use the overall score a code submission receives. We also qualitatively demonstrate how test case information, combined with open-ended code, helps us gain fine-grained insights into student knowledge.

## CCS Concepts

• **Applied computing** → **Education**; • **Computing methodologies** → **Natural language processing**.

## Keywords

Computer Science Education, Large Language Models, Open-ended Coding Questions, Test Cases

**ACM Reference Format:**
Zhangqi Duan, Nigel Fernandez, Alexander Hicks, and Andrew Lan. 2025. Test Case-Informed Knowledge Tracing for Open-ended Coding Tasks. In *LAK25: The 15th International Learning Analytics and Knowledge Conference (LAK 2025), March 03–07, 2025, Dublin, Ireland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3706468.3706500



*LAK 2025, Dublin, Ireland*

ACM ISBN 979-8-4007-0701-8/25/03
https://doi.org/10.1145/3706468.3706500

## 1 Introduction

Open-ended coding questions, or similarly program synthesis tasks, which require students to write open-ended code according to natural language instructions [9, 43], are common in computer science (CS) education. Since these questions require students to *construct* an entire solution, i.e., write code, they may enable education researchers to gain deeper insights into student knowledge compared to other question formats such as multiple-choice. In many domains such as math and/or essay writing, researchers have found evidence that students' open-ended responses contain useful information on their knowledge states, e.g., having misconceptions [5, 12, 44] or generally lacking sufficient knowledge [4]. A particular advantage of open-ended coding questions in CS education is that student code submissions can be automatically graded, by *compiling* them and passing them through a series of *test cases*; this automated evaluation is usually not applicable in other subjects. Compiling student code can identify syntax errors in student-written code [37], while analyzing whether they passed each test case may identify conceptual or logical errors [46]; both types of error information can be used to provide timely *feedback* to students to help them correct their errors.

However, existing work on *student modeling* for open-ended questions is limited, possibly due to the noisy nature of open-ended student responses and the scarcity of available data. Evaluation of such models can also be challenging, both quantitatively and qualitatively. The most common evaluation metric is binary *correctness* of student-written code, which corresponds to 1 if a code submission compiles with no syntax error and passes all test cases, and 0 otherwise [38, 55]. This evaluation metric is standard in student modeling literature, for both the item response theory (IRT) [47] and knowledge tracing (KT) [7] model families.

Along these lines, existing work has attempted to automatically identify errors and misconceptions among students [41] or discover knowledge components (KCs) [40]. Despite these functionalities, these works are fundamentally limited by the fact that overall code correctness does not reveal fine-grained insights about student knowledge. More detailed information on whether a student code passes each specific test case is likely necessary to pinpoint specific student errors and even misconceptions.

**Table 1: Example from our test case-augmented CodeWorkout dataset, which shows a student's code submission to an open-ended coding problem, along with whether it passes a subset of associated test cases.**

Problem: A sandwich is two pieces of bread with something in between. Write a Java method that takes in a string str and returns the string that is between the first and last appearance of 'bread' in str. Return empty string '' if there are not two pieces of bread.

| Student Code Submission | Test Cases | | Pass/Fail |
|---|---|---|---|
| | Input | Output | |
| `public String getSandwich(String str)`<br>`{`<br>`  int length = str.length();`<br>`  if ( str.startsWith("bread") && str.endsWith("bread") )`<br>`  {`<br>`  return str.substring(5, str.length() - 5);`<br>`  }`<br>`  else`<br>`  {`<br>`  return "";`<br>`  }`<br>`}` | 'breadjambread' | 'jam' | ✓ |
| | 'xxbreadjambreadyy' | 'jam' | ✗ |
| | 'xxbreadbreadjambreadyy' | 'breadjam' | ✗ |
| | 'breadbread' | '' | ✓ |
| | '' | '' | ✓ |
| | 'breaxbreadybread' | 'y' | ✗ |
| | 'breadbreadbread' | 'bread' | ✓ |

Recently, building on rapid advances in large language models (LLMs) and their ability to automatically generate code, researchers have started to develop LLM-powered student models that are capable of not only predicting student code correctness but also *generating* possible student code submissions. These models have the potential to provide fine-grained analysis of student code, extract insights on specific coding styles, errors, and misconceptions, and track how they evolve over time. However, evaluating predicted open-ended codes is much more challenging than evaluating predicted student-written code correctness. The open-ended KT (OKT) framework [23] replaces the binary-valued response prediction model in existing KT methods with an LLM-based code generator, in order to predict student code submission in a line-by-line, token-by-token way. Despite this new capability, evaluation of code prediction accuracy relies on the CodeBLEU metric [34], which is influenced by the token n-gram overlap between the predicted code and actual student-written code. As a result, this evaluation is better at capturing *surface code semantic similarity* rather than digging deep into student logic behind their code. The student attempt synthesis framework [43] uses human experts in-the-loop to evaluate code prediction performance, which is not scalable to real-world application scenarios. For both of these methods, passing the generated code and actual student-written code through test cases and comparing the results will result in a more comprehensive evaluation: test case pass/fail helps us to characterize student code *functionality* in addition to surface semantic similarity.

## 1.1 Contributions

In this work, we extend KT for open-ended coding questions to incorporate test case-level information, attempting to use them to define a more fine-grained KT task for CS education. Our contributions are summarized as follows:

- We define a new KT task that analyzes and predicts whether a student passes each test case in their code submission. This task is more challenging for KT methods since it requires them to make numerous predictions simultaneously for each student code submission.
- We augment the CodeWorkout dataset [8] with the test cases used for a subset of the questions. We have publicly released this augmented dataset to serve as a benchmark on the test case-level KT task for future KT methods.
- We introduce Test case-Informed Knowledge Tracing for Open-ended Coding (TIKTOC)[1], a novel method for the test case-level KT task. TIKTOC uses a multi-task learning setup to jointly maximize i) test case-level student code submission correctness prediction accuracy and ii) student code generation accuracy, effectively combining and balancing the two types of KT methods outlined above.
- We conduct experiments on the CodeWorkout dataset and show that TIKTOC's multi-task learning setup leads to significant improvement in both test case pass/fail prediction accuracy (15% gain in AUC) and code prediction accuracy (6% gain in CodeBLEU) over existing state-of-the-art KT methods adapted to this task.
- We qualitatively show that the new KT task and resulting methods may lead to deeper and more fine-grained insights into student knowledge for open-ended coding questions. We discuss potential use cases of TIKTOC in CS education, along with its limitations, and outline several directions for future work.

# 2 Related Work

## 2.1 Knowledge Tracing

KT [7] is a well-studied task in the student modeling literature. It studies the problem of breaking down student learning into a series of time steps practicing certain KCs and using the correctness of each response step to track student knowledge. Classic Bayesian knowledge tracing methods [30, 51] use latent binary-valued variables to represent whether a student masters a KC or not. With the increase in popularity of neural networks, multiple deep learning-based KT methods were developed. These models have limited interpretability since student knowledge is modeled as hidden states in complex, deep neural networks. Most of these methods use long short-term memory networks [15] or variants [31, 42], with other variants coupling them with memory augmentation [53], graph neural networks [50], attention networks [14, 28],

[1]Our code and supplementary test case data can be found at https://github.com/umass-ml4ed/tiktoc.

or pre-trained word embeddings or LLMs to leverage question text information [23, 24]. These methods vary in how they represent questions and student knowledge; see [1] for a survey. KT methods have been applied to many different educational domains, including CS [16, 39, 55].

## 2.2 Program Synthesis in Computer Science Education

There is a line of existing work on analyzing student-generated code, most noticeably using the Hour of Code dataset released by Code.org [32, 43, 48], for tasks such as error analysis and automated feedback generation that are meaningful in CS education settings. Program synthesis techniques have been applied for CS education to generate (possibly buggy) student code [13, 23], generate new problems [2] with code explanations [36], generate student-code guided test cases [21], provide real-time hints [35], guide students with Socratic questioning [20], and suggest bug fixes [19]. However, past work has mostly not incorporated test cases to model student learning in programming. TIKTOC is among the first attempts to analyze and predict whether a student's code submission passes each test case, potentially offering more fine-grained insights into student knowledge.

# 3 Knowledge Tracing at the Test Case Level

We now formulate the task of KT at the test case level for open-ended coding problems in CS education.

## 3.1 Problem Formulation

The goal of the classic KT task is to estimate a student's mastery of KCs/skills/concepts from their responses to past problems and use these estimates to predict their future performance. Formally, given the history of a student's past interactions (e.g., a response to a problem), $x_0, \ldots, x_t$, KT aims to predict aspects of their interaction $x_{t+1}$ (e.g., response correctness on the next problem). For open-ended coding problems, we define a student's interaction with a coding problem as $x_t := (p_t, \{s_t^i\}, c_t, a_t, \{y_t^i\})$, where $p_t$ is the textual statement of the coding task, $\{s_t^i\}$ are test cases developed to test correctness of code for this problem, $c_t$ is the code submitted by the student, $a_t$ is the score the submission receives, and $y_t^i \in \{0, 1\}$ is the test case level past/fail indicator. Here, we use $i$ to index the test cases for each problem. Following existing work [39], the binary correctness $a_t$ is equal to 1 if the code submitted by the student passes all test cases and 0 otherwise, which provides us a measure of the *overall* correctness of a student submission.

In existing KT methods for CS education [23, 27, 39], test cases have not been extensively involved in both modeling and evaluation, as we discussed above. However, well-designed test cases can be used to anticipate common logical, syntax, and runtime errors among students, thereby naturally providing fine-grained error insights into nuanced aspects of student knowledge. With this motivation, we investigate a new *test case-level KT task*, which we formalize as: Given a history of student interactions $x_0, \ldots, x_t$ on past programming problems, with each interaction containing information $x_t = (p_t, \{s_t^i\}, c_t, a_t, \{y_t^i\})$, predict which test cases they will pass/fail, i.e., $\{y_{t+1}^i\}$, on their next attempted problem $p_{t+1}$.

## 3.2 Dataset Creation

We augment the existing CodeWorkout dataset [8] with test cases for a subset of the questions. We provide an illustrative example from our test case-augmented CodeWorkout dataset in Table 1, which includes a sample programming problem, along with a subset of its associated test cases, and a sample code submission made by a student attempting the problem. We provide statistics of the augmented dataset in Table 2 and show the CodeWorkout [8] interface in Figure 1.

The CodeWorkout dataset is a large, real-world programming education dataset previously used in the Second CSEDM Data Challenge [6]. It is a unique, publicly available dataset with university-level Java problems; the dataset contains actual open-ended code submissions from real students, collected from an introductory Java programming course at a large US university. The problems cover various programming concepts including conditionals, and loops, among others. We augment 17 problems with test cases, with an average of 18 test cases per problem, leading to 305 total test cases. These test cases are authentic ones deployed when the dataset was collected: running compilable student code submissions $c_t$ on these test cases recovers the score $a_t$, in the original dataset on PSLC DataShop [8].

The test cases we extract are a subset of the original test cases used by CodeWorkout during the collection of the CSEDM dataset in 2019 [6]. Historically, CodeWorkout has supported three methods of writing instructor-provided tests: JUnit test cases written by exercise authors, output validation tests that provide input and match output from the execution of the exercise to an expected value, and JUnit tests that were interpolated into base JUnit test file server side; we extract test cases of the second type that are written by course staff and CodeWorkout developers. The test cases were designed to evaluate the correctness of the student submission by checking for common logical errors in the students' code. Test cases can be both public and hidden: public ones often seek to remind the students to consider specific edge cases, while hidden ones will reserve one logical fallacy or edge case as an exercise for the student to identify. For example, Table 1 includes a test case that inputs an empty string (''). This test is an example of an edge case check, verifying the behavior of the student's code at a boundary value that they might not consider as valid input. Another good test case ('breadbreadbread') checks whether students handle multiple instances of the keyword token correctly, by ensuring they properly grab the first and last instances in a string, instead of a common misconception, the first and second instances. There are often multiple variants of easy test cases ('breadjambread') to give students credit and encourage them, while they incrementally build and test their code [11].

We leverage these test cases associated with each problem to obtain *test case-level pass/fail labels*, i.e., $\{y_t^i\}$, for each student code submission. We build an automated evaluation pipeline that first compiles a student code submission for a problem. If a student code fails to compile, we mark all test cases associated with the problem as failed for this submission. To account for cases like infinite loops, we add a timeout condition in our pipeline: If a student code times out after 30 seconds when attempting compilation, we also mark all associated test cases as failed. If a student code compiles, our

**Table 2: Statistics of our test case-augmented CodeWorkout dataset with 3714 student code submissions.**

| | | | | |
|---|---|---|---|---|
| # unique problems | 17 | | | |
| # unique students | 246 | | | |
| # total no. of test cases | 305 | | | |
| # student code submissions | 3714 | | | |
| Statistic | $\mu$ | $\sigma$ | Min | Max |
| # test cases/problem | 17.9 | 4.6 | 8 | 26 |
| # lines/submission | 16.7 | 7.6 | 5 | 82 |
| # tokens/problem | 67.3 | 23.1 | 40 | 123 |
| # tokens/test case | 9.8 | 4.7 | 3 | 32 |
| # tokens/submission | 80.8 | 34.6 | 12 | 344 |
| # submissions/student | 15.1 | 3.1 | 4 | 17 |
| # submissions/problem | 218.5 | 11.1 | 198 | 233 |


**Figure 1: The CodeWorkout [8] interface with the problem on top, the coding area on the left, and the test case feedback on the right.**

pipeline automatically runs the code against all associated test cases. If the output of the code matches the expected output obtained by running the correct solution code, we mark the test case as passed and as failed otherwise.

## 4 Methodology

KT at the test case level is a novel task with *no existing methods*. Therefore, we build on two strong, state-of-the-art KT methods for programming, Code-based Deep KT (Code-DKT) [39] and Open-Ended KT (OKT) [23], to develop a multi-task learning approach and apply it to the test case level KT task. In this section, we first introduce these methods before detailing our novel method, TIKTOC, for KT at the test case level.

### 4.1 Code-DKT

Code-DKT [39] is a KT method that leverages the content of student code to improve on DKT [31] for open-ended coding tasks. Instead of using only the binary-valued correctness of the last student response as input, Code-DKT uses code2vec [3], a neural code representation model, to obtain a meaningful representation of past student code submissions. To match the original DKT's data format, Code-DKT characterizes a student code submission as binary-valued, aggregated across all test cases: if all test cases pass, the submission is correct, otherwise, it is incorrect. Therefore, an interaction is simplified to $x_t := (p_t, c_t, a_t \in \{0, 1\})$. For each student, at each timestep $t$ (i.e., an interaction or student submission), Code-DKT updates the student's estimated knowledge state by

$$h_t = \text{LSTM}(h_{t-1}, c_t, p_t, a_t),$$

through a long short-term memory (LSTM) network [15]; the input to the LSTM update module is a combination of the code2vec embedding of the student code, $c_t$, and a one-hot encoded representation of the last problem-response correctness, $(p_t, a_t)$, like DKT [31]. To predict the correctness of the next student submission to the problem $p_{t+1}$, Code-DKT uses a linear prediction head and a sigmoid function:

$$\hat{a}_{t+1} = \sigma(W \cdot h_t), \tag{1}$$

where $W$ denotes the parameters of a learnable linear layer. Code-DKT then minimizes the binary cross entropy (BCE) loss for submission correctness prediction from one student code submission:

$$\mathcal{L}_{\text{Code-DKT}} = a_t \cdot \log \hat{a}_t + (1 - a_t) \cdot \log(1 - \hat{a}_t)$$

with the final objective being the mean of this loss over code attempts by all students to all attempted problems.

Code-DKT and other KT methods predicting overall correctness are fundamentally limited by their lack of fine-grained insights about student knowledge. On the other hand, leveraging test cases and predicting which test cases a student code would pass could provide specific student errors and misconceptions.

### 4.2 OKT

OKT [23] is the first KT method that is generative in nature; it leverages LLMs to predict the open-ended code submitted by a student on their next programming problem in a line-by-line, token-by-token way, instead of just predicting the binary-valued correctness of their code.

In OKT, the input to the KT model relies on a different embedding method to encode past student code submissions than Code-DKT. For student code, OKT first transforms it into an Abstract Syntax Tree (AST), followed by using the ASTNN [54] model to obtain an embedding preserving the syntactic and semantic features of the code. Moreover, as an improvement over Code-DKT, OKT also encodes the textual problem statement using text embedding methods to include as part of the input to the KT model. The KT model in OKT is flexible and can be adapted from any existing binary-valued KT method. For example, if we choose DKT as the KT model, then we simply use the encoded problem statement and student code from the previous response as input to the LSTM model, just like Code-DKT.

The main difference between OKT and all other existing KT methods is that it uses an LLM-based response generation component to predict the next student code submission, $c_{t+1}$, given the current knowledge state, $h_t$, and the statement of the next problem, $p_{t+1}$. Specifically, the next problem text $p_{t+1}$ is tokenized by an open-source LLM into a sequence of $M$ tokens, where each token has a $D$-dimensional embedding, i.e., $\bar{p}_m \in \mathcal{R}^D$ for $m = 1, \ldots, M$ (we drop the timestep $t + 1$ in denoting problem tokens for simplicity). Then, OKT injects the knowledge state of the student $h_t$

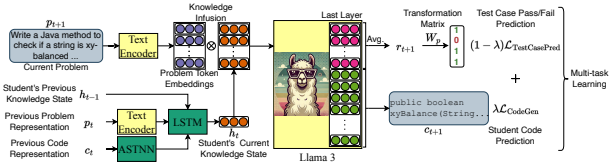



**Figure 2: Overview of TIKTOC's model architecture with the Llama 3 LLM as the backbone. TIKTOC embeds the student's previous open-ended code to update the student's knowledge estimate, which is then combined with the current problem, as input to Llama 3. TIKTOC simultaneously learns to predict both 1) whether a student's code submission passes each test, and 2) the student's open-ended code, with a multi-task learning setup.**

into the LLM, creating *knowledge-guided* token embeddings using a linear alignment function $f$, i.e., $p_m = f(\bar{p}_m, h_t)$. These knowledge-guided problem embeddings are fed into Llama 3 to generate the predicted student code, token-by-token. The loss to be minimized for one student code submission is given by:

$$\mathcal{L}_{\text{OKT}} = \frac{1}{N}\sum_{n=1}^{N} -\log P_\theta\left(c^n \,\middle|\, p_m, \{c^{n'}\}_{n'=1}^{n-1}\right),$$

where $N$ is the number of tokens in the student code submission. $\theta$ denotes the set of learnable parameters, which includes the underlying KT model, the alignment function, and the LLM's parameters (if fine-tuned). The final training objective is the mean of this loss over code submissions by all students to all attempted problems.

OKT analyzes richer information than Code-DKT in the form of open-ended student code. However, comparing predicted code and actual student-written code only captures surface code semantic similarity. We need to look deeper to gain fine-grained insights into student knowledge and identify specific logical errors or misconceptions.

## 4.3 TIKTOC

We now detail our TIKTOC model, illustrated in Figure 2. TIKTOC has two key innovations: First, we utilize test cases and model whether a student attempt passes each single test case, potentially offering more fine-grained insights into student knowledge. Second, we use a multi-task learning setup to combine the objectives of both Code-DKT and OKT: TIKTOC simultaneously analyzes and predicts 1) whether a student's code submission passes each test case, and 2) the student's open-ended code. Intuitively, these two tasks can be mutually beneficial to each other, which motivates the multi-task learning approach. For example, failing a test case evaluating a loop termination conditional significantly reduces the space of possible code, which enables better student code prediction reflecting this error, and vice versa.

*Setup.* TIKTOC combines and builds upon elements from both Code-DKT and OKT into a unified model that uses a single LLM as the backbone. For each student, at each timestep $t$, TIKTOC first converts their sequence of past interactions $x_0, \ldots, x_t$ attempting programming problems up to timestep $t$ into continuous representations, where each interaction $x_t := (p_t, c_t, \{y_t^i\})$ contains the attempted problem, their code submission, and pass/fail outcomes of the test cases associated with the problem. Similar to OKT [23], for the problem representation, TIKTOC uses the mean problem token embeddings from Llama 3, and for the code representation, uses ASTNN [54] to embed student code.

*Code generation.* The code generation setup for TIKTOC is similar to OKT [23], with DKT as the underlying KT model, except that we use Llama 3 [25], a more recent, more powerful open-source LLM as the code generation backbone. We use the same way to inject student knowledge states into the LLM embedding space, with one modification: we use hidden knowledge states with $D = 4096$ dimensions to align with Llama 3's text embedding space. The loss for one student code submission is given by

$$\mathcal{L}_{\text{CodeGen}} = \sum_{n=1}^{N} -\log P_\theta\left(c^n \,\middle|\, p, j, \{c^{n'}\}_{n'=1}^{n-1}\right), \tag{2}$$

with the final $\mathcal{L}_{\text{CodeGen}}$ loss being the mean over code attempts by all students to all attempted problems.

*Test case prediction.* For the task of predicting individual test case pass/fail outcomes for a student code submission, we modify the setup of Code-DKT's prediction head. We average the hidden states of the last layer of Llama 3 that correspond to only the input, i.e., the knowledge-guided problem statement embeddings, to obtain a representation $r$ that we use for predictions. We choose to not include the output code prediction hidden states in our average, since including them means that the predictions are conditioned on generated code, which can simply be done by the code compiler in a non-probabilistic way. Building on Code-DKT [39], we transform $r$ into a prediction vector $\hat{y}$ through a transformation matrix $W_p$, i.e.,

$$\hat{y} = \sigma(W_p r).$$

The prediction vector has $|S_p|$ dimensions, where $|S_p|$ is the number of test cases associated with problem $p$. Here, element $i$ of the vector $\hat{y}$ represents the probability of the student submission passing test case $s_p^i$. The BCE loss for test case pass/fail outcome prediction for one student code attempt is given by

$$\mathcal{L}_{\text{TestCasePred}} = \frac{1}{|S_p|}\sum_{i=1}^{|S_p|} y^i \cdot \log(\hat{y}^i) + (1-y^i)\cdot\log(1-\hat{y}^i). \tag{3}$$

with the final $\mathcal{L}_{\text{TestCasePred}}$ loss being the mean over code attempts by all students to all attempted problems.

We learn a separate transformation matrix $W_p$ for each problem since the number and nature of test cases differ across problems. We experiment with different variants of parameterizing $W_p$ which we detail in Section 6.2. We find that treating test cases as one-hot encodings performs best. We randomly initialize $W_p$ and set the number of columns to be equal to the number of test cases associated with problem $p$, with each column thereby learning a representation of its corresponding test case.

*Multi-task learning.* Our final multi-task objective minimizes a combination of both losses together, i.e., the code generation loss $\mathcal{L}_{\text{CodeGen}}$ in Eq. 2 and the test case outcome prediction loss

$\mathcal{L}_{\text{TestCasePred}}$ in Eq. 3, with a balancing parameter $\lambda \in [0, 1]$ controlling the importance of the two losses, i.e.,

$$\mathcal{L}_{\text{TIKTOC}} = \lambda \mathcal{L}_{\text{CodeGen}} + (1 - \lambda)\mathcal{L}_{\text{TestCasePred}}. \quad (4)$$

The optimal value of $\lambda$ can be learned [18] as a parameter of the model or through a grid search. In our experiments, we find TIKTOC significantly outperforms baseline KT methods with an initial guess of $\lambda = 0.5$ making a grid search redundant. We hypothesize that both objectives offer valuable insights into the underlying KT problem and can benefit each other, improving over standalone versions of both Code-DKT and OKT.

# 5 Experiments

## 5.1 Metrics

For the test case-level KT task, since the pass/fail status of each student code submission on each test case is binary-valued, we evaluate the performance of KT methods using standard metrics such as AUC, F1 Score, and accuracy [31, 39]. Since the test case pass/fail outcome labels can be imbalanced, AUC and F1 Score are important. For the *student code prediction* task, following OKT [23], we evaluate model-generated student codes against ground-truth student codes using CodeBLEU [34], a variant of the classic text similarity metric BLEU [29], adapted for code. This metric measures both the syntactic and semantic similarity between two pieces of code. To test whether models simply memorize frequent student code in the training data, we also evaluate predicted *student code diversity*. We measure code diversity by using the popular dist-N metric [22] which computes the ratio of unique N-grams to the total number of N-grams.

## 5.2 Baselines

KT at the test case level is a novel task with *no existing methods*. Therefore, we adapt both Code-DKT [39] and OKT [23] to this task, to serve as strong baselines. We adapt the standard Code-DKT method, detailed above in Sec. 4.1, to our test case-level KT task, which we refer to as **Code-DKT-TC**. Specifically, we modify Code-DKT's prediction head to predict individual test case pass/fail outcomes for a student code submission in a similar fashion as TIKTOC. We concatenate the embedding of the next problem $p_{t+1}$ from Llama 3 [25] with the student's estimated knowledge state, $h_t$, from the LSTM [15]. We then transform it into a prediction vector, $\hat{y}$, through a non-linear transformation, $\hat{y} = \sigma(W_p(p_{t+1} \oplus h_t))$, where $\oplus$ denotes vector concatenation and $W_p$ denotes a problem-specific learable transformation matrix. The training objective for Code-DKT-TC is the same as the test case prediction objective of TIKTOC in Eq. 3. To ensure a fair comparison, we estimate a student's knowledge state in the same way as in OKT, using Llama 3's mean token embeddings of the problem statement and the ASTNN embedding of student code as input to the LSTM. We switch from the original code2vec [3] representation to ASTNN [54] to embed student codes and use ASTNN across methods to align with the settings in OKT. This switch is also motivated by the observation that ASTNN outperforms code2vec in embedding student codes when modeling student learning in programming [27].

We also adapt OKT, detailed in Sec. 4.2, to our test case-level KT task, which we refer to as **OKT-TC**. This baseline simply takes the predicted student code submission generated by OKT, compiles it, and evaluates it on the test cases. This evaluation pipeline follows recent work [37] on detecting errors/bugs present in predicted code by OKT. Same as before, if the predicted code fails to compile or times out during compilation, we mark all associated test cases as failed. We change the base LLM of OKT [23] from GPT-2 [33] to the more recent and powerful Llama 3 [25]. We use Llama 3 as the base LLM across all KT methods we experiment with in this work to ensure a fair comparison.

As a sanity check, to estimate the difficulty of our test case-level KT task and to estimate a lower bound of performance, we also incorporate two simple baselines: **Random**, which simply predicts the test case pass/fail outcome of a student code randomly with equal probability, and **Majority**, which simply predicts the test case pass/fail outcome of a student code as pass since pass occurs more frequently than fail.

## 5.3 Experimental Setup

Following prior work [39], we experiment on the first code submission for each student on each problem in the CodeWorkout dataset. To ensure a fair comparison across KT methods, we use the instruction-tuned version of Llama 3 [25] with 8B parameters as the base LLM, and a frozen ASTNN [54] code embedding model to embed student codes following OKT [23]. We use the Parameter Efficient Fine-Tuning (PEFT) library from HuggingFace [49] to load Llama 3 8B Instruct, and train via low-rank adaptation (LoRA) [17] (LoRA $\alpha = 256$, LoRA $r = 128$, LoRA dropout $= 0.05$) using 8-bit quantization [10]. We use the AdamW [26] optimizer with a batch size of 32. We perform a grid search to find the optimum learning rate (LR). For TIKTOC and OKT, we use an LR of $1e-5$ for Llama 3 8B Instruct, $5e-5$ for the LSTM, and $1e-4$ for the transformation matrix $W_p$ as well as the linear alignment function $f$. We use a linear LR scheduler with warmup and perform gradient clipping. For Code-DKT-TC, we use an LR of $1.5e-3$ for the LSTM and $1e-3$ for the transformation matrix $W_p$, with the ReduceLROnPlateau LR scheduler.

For Code-DKT-TC and TIKTOC, we learn a separate transformation matrix $W_p$ for each problem, with each column representing an associated test case using $D = 4096$ dimensions. For OKT, we use the Java compiler from OpenJDK 11.0.23 to compile and run predicted student codes against test cases. For TIKTOC, we find that simply setting the multi-task balancing parameter in Eq. 4 to $\lambda = 0.5$ works well across all settings. A likely cause is that once normalized over all test cases and all code tokens, both objectives are BCE losses and are already on the same scale, which alleviates the need for a grid search over this parameter. For OKT and TIKTOC, we use greedy decoding to generate student code. We fine-tune for 20 epochs with early stopping on the validation set on a single NVIDIA L40S 48GB GPU, with each epoch taking up to 1, 35, and 45 minutes, for Code-DKT, OKT, and TIKTOC, respectively.

# 6 Results, Analysis, and Discussion

In this section, we quantitatively evaluate model performance on test case-level KT and student code prediction, perform an ablation study on TIKTOC, and compare TIKTOC with other KT methods through qualitative case studies.

## 6.1 Quantitative Results

Table 3 shows the average performance (and standard deviation) on test case pass/fail prediction and code generation, across 5 random folds, for all methods. We see that the simple Random and Majority baselines perform poorly, which suggests that test case-level student response prediction is inherently difficult. This difficulty comes from the need to simultaneously predict numerous test cases for each student code submission. Code-DKT-TC outperforms OKT-TC on test case level pass/fail prediction, although it cannot perform the code generation task. This result can be explained by Code-DKT-TC being explicitly trained on the test case-level prediction objective; since OKT-TC evaluates test case pass/fail after the entire predicted student code is generated, the discrete nature of code makes this process non-smooth, lowering the robustness of the prediction task.

We see that our proposed approach, TIKTOC, outperforms the next best baseline on test case pass/fail prediction, Code-DKT-TC, by a wide margin of 15.6% on the AUC metric. TIKTOC also outperforms the best baseline on student code prediction, OKT-TC, by 6.1% on the CodeBLEU metric. TIKTOC achieves a statistically significant performance improvement over Code-DKT-TC/OKT-TC, the next best baseline(s) on test case-level KT/student code prediction, with p-values of 0.005/0.02, respectively, obtained via a paired t-test. These results validate our hypothesis that test case-level pass/fail prediction and student code prediction are two different tasks that are beneficial to each other; TIKTOC effectively leverages this symbiosis with a multi-task learning approach.

## 6.2 Ablation Study

Table 4 shows the results of an ablation study for TIKTOC. We see that combining student knowledge states with the LLM's input text embeddings is crucial and removing the student model results in a large drop in performance, as seen in row **No Knowledge Estimation**. This observation highlights the importance of the underlying student model.

There can be many ways to represent different test cases in TIKTOC. We find that treating test cases as one-hot encodings performs best, as seen in row **One-hot Test Case**, where we initialize the embeddings of test cases independently at the start of the model training process. Alternatively, instead of random initialization, we experiment with initializing each test case representation with their mean token embedding from Llama 3 [25]. This method, shown in row **Embed Test Case**, leverages the textual information of the test cases. We also experiment with embedding both the problem content and that of test cases, shown in row **Embed Test Case with Problem**. However, we see that neither of these approaches improves performance over one-hot encoding test cases, which is counterintuitive; one possible reason is that the test cases are usually functions and not purely textual in nature, which can be difficult for an LLM to capture. Future work needs to develop ways to effectively leverage the content of test cases to improve performance on the test case pass/fail prediction task.

## 6.3 Qualitative Case Studies

We now qualitatively show that the test case pass/fail prediction and code generation objectives in our multi-task learning setup are mutually beneficial to each other with three qualitative case studies.

*Code generation helps test case pass/fail predictions.* Table 5 shows a sample interaction from the test set, where a student attempts a problem to check whether a string is "xy-balanced". TIKTOC accurately predicts a semantically close student code submission to the ground-truth student code submission, albeit with different variable names. Code-DKT incorrectly predicts that the two harder test cases, the empty string '', and the string 'bbb', without either 'x' or 'y', which test edge cases in a student's logic, will fail. TIKTOC, on the other hand, can leverage its knowledge of the anticipated student code, to implicitly run this code against its test case representations, thereby imitating a compiler, to accurately predict the test cases would pass.

*Test case pass/fail predictions help code generation.* Table 6 shows a sample interaction where a student attempts a problem to check whether three integers are evenly spaced. TIKTOC correctly predicts all test case pass/fail labels matching the ground-truth labels. In particular, test cases containing three unsorted evenly spaced integers, for example, $(4, 6, 2)$ and $(6, 2, 4)$, are correctly predicted as failed. OKT, which does not have access to these valuable test case pass/fail predictions, incorrectly predicts a student code that will pass all test cases, using constructs such as "Math.abs" not present in the ground-truth student code. TIKTOC, on the other hand, can leverage its knowledge of which test cases are predicted to fail, to predict a semantically closer student code to the ground-truth student code. It successfully predicts a similar bug that reflects the student's logical error in assuming the three input integers are sorted.

*Heatmap of test case pass/fail predictions.* Table 7 shows a heatmap of test case pass/fail probabilities predicted by TIKTOC, for a student attempting four problems, $P_{20}$, $P_{17}$, $P_{22}$, and $P_{21}$ in a row. These problems mainly cover the KC of if/else conditionals, requiring students to use multiple such conditionals correctly in their code. Problems $P_{20}$ and $P_{17}$ are similar, with $P_{17}$ being the easiest among the four problems; $P_{22}$ involves the additional KC of writing a helper method, and $P_{21}$ is the most difficult, requiring students to consider repetition and ordering of the input. Each cell in the heatmap represents a test case and is labeled with the actual pass/-fail status. Cell colors depict the TIKTOC predicted probability of the student passing the test case, with darker colors corresponding to higher probabilities. Each column represents a subset of test cases associated with a problem since there are too many to visualize together. We also note that test cases differ across problems, which means cells in the same row may not perfectly align. Instead, we align them manually by grouping test cases across problems with similar difficulty or underlying KC.

For problem $P_{20}$, TIKTOC correctly predicts the student will pass all easy test cases that check whether there is an else block that returns the sum of the three input integers. TIKTOC correctly predicts the student will fail a hard edge test case, $(121, 121, 121)$, with repeated input integers. We see that the predicted code (shown in the middle of Table 7) is semantically close to the ground-truth

**Table 3: Performance on 1) test case-level KT and 2) student code prediction for all approaches across all metrics. TIKTOC, with a multi-task learning setup with the Llama 3 LLM as the backbone, outperforms existing KT approaches by a wide margin. Standard deviation across 5 random seeds are shown in parentheses. Best performance is in bold and second best is underlined.**

| | Test Case-level KT Performance | | | Student Code Prediction | |
|---|---|---|---|---|---|
| Model | AUC ↑ | F1 Score ↑ | Accuracy ↑ | CodeBLEU ↑ | Dist-1 ↑ |
| Random | 0.501 (0.8%) | 0.545 (2.5%) | 0.502 (0.6%) | – | – |
| Majority | 0.501 (0.3%) | 0.719 (8.9%) | 0.620 (5.5%) | – | – |
| Code-DKT-TC [39] | 0.661 (4.3%) | 0.771 (2.4%) | 0.677 (2.5%) | – | – |
| OKT-TC [23] | – | 0.763 (3.3%) | 0.647 (3.2%) | 0.522 (2.1%) | **0.383** (1.3%) |
| TIKTOC (ours) | **0.764** (4.3%) | **0.794** (4.5%) | **0.723** (3.0%) | **0.554** (1.0%) | 0.369 (5.9%) |

**Table 4: Ablation study of TIKTOC comparing different model variants. Best performance is in bold and second best is underlined.**

| | Test Case-level KT Performance | | | Student Code Prediction | |
|---|---|---|---|---|---|
| TIKTOC Model Ablations | AUC ↑ | F1 Score ↑ | Accuracy ↑ | CodeBLEU ↑ | Dist-1 ↑ |
| No Knowledge Estimation | 0.699 | 0.799 | 0.648 | 0.517 | 0.357 |
| Embed Test Case | 0.757 | 0.834 | 0.792 | 0.532 | 0.360 |
| Embed Test Case with Problem | 0.744 | 0.828 | 0.786 | 0.55 | 0.366 |
| One-hot Test Case (Main Model) | **0.764** | **0.837** | **0.802** | **0.566** | **0.369** |

**Table 5: Qualitative case study showing code generation helps test case pass/fail predictions. TIKTOC can leverage its knowledge of anticipated student code to implicitly run this code against its test case representations. As a result, it can accurately predict test case pass/fail outcomes. A subset of test cases is shown. The code indentation is changed for brevity.**

Problem: A string is xy-balanced if for all the 'x' characterss in the string, there exists a 'y' character somewhere later in the string. So 'xxy' is balanced, but 'xyx' is not. One 'y' can balance multiple 'x'. Return true if the given string is xy-balanced.

| Ground-truth Student Code | TIKTOC Predicted Student Code | Test Cases | | ✓/✗ Preds | | |
|---|---|---|---|---|---|---|
| | | Input | Output | Ground-truth | Code-DKT-TC | TIK-TOC |
| `public boolean xyBalance(String str)`<br>`{`<br>`  int length = str.length() - 1;`<br>`  char character;`<br>`  for(int i = length; i >= 0; i--){`<br>`  character = str.charAt(i);`<br>`  if(character == 'x'){`<br>`  return false;`<br>`  }`<br>`  else if(character == 'y'){`<br>`  return true;`<br>`  }`<br>`  }`<br>`  return true;`<br>`}` | `public boolean xyBalance(String str)`<br>`{`<br>`  int len = str.length() - 1;`<br>`  char ch;`<br>`  for(int i = len; i >= 0; i--){`<br>`  ch = str.charAt(i);`<br>`  if(ch == 'x')`<br>`  return false;`<br>`  else if(ch == 'y')`<br>`  return true;`<br>`  }`<br>`  return true;`<br>`}` | 'aaxbby' | 'true' | ✓ | ✓ | ✓ |
| | | '' | 'true' | ✓ | ✗ | ✓ |
| | | 'aaxbb' | 'false' | ✓ | ✓ | ✓ |
| | | 'yaaxbb' | 'false' | ✓ | ✓ | ✓ |
| | | 'yaaxbby' | 'true' | ✓ | ✓ | ✓ |
| | | 'xaxxbby' | 'true' | ✓ | ✓ | ✓ |
| | | 'xaxxbbyx' | 'false' | ✓ | ✓ | ✓ |
| | | 'xxbxy' | 'true' | ✓ | ✓ | ✓ |
| | | 'bbb' | 'true' | ✓ | ✗ | ✓ |

student code, both containing a similar logical error. TIKTOC correctly predicts the student will pass all test cases for $P_{17}$, the easiest problem, which tests KCs practiced before, in $P_{20}$.

The student's past knowledge from $P_{17}$, $P_{20}$, and other past problems, does not transfer to the new KC, writing a helper method, introduced in problem $P_{22}$. We see that their code fails all test cases, an outcome correctly predicted by TIKTOC. On problem $P_{21}$, the hardest problem, TIKTOC correctly predicts that the student will pass easy test cases covered before, but fail a hard edge case, (13, 13, 13), of a similar KC from $P_{20}$, where the inputs are all equal. This logical error is also reflected in the predicted student code (shown in the right of Table 7) in a semantically similar manner to the ground-truth student code. Across these problems, we see the student's estimated knowledge levels on similar KCs generally increase monotonically.

## 6.4 Possible Use Cases in CS Education

We now discuss how TIKTOC's test case pass/fail predictions of student code can be useful in real-world CS educational scenarios. For example, if a student is predicted to consistently fail test cases assessing operator precedence, this observation might suggest that they exhibit misconceptions of arithmetic expressions. Instructors can use TIKTOC's predictions to anticipate student errors on open-ended coding tasks through predicted test case pass/fail outcomes, even *before* actually assigning these tasks to students. They can then prepare corresponding feedback, adjust the difficulty of the problems, and even design a suitable curriculum in advance.

Writing good test cases for open-ended coding tasks can be challenging; the predictions from TIKTOC can perhaps help instructors on this task. For example, if a majority of students are predicted to pass all test cases for a problem, it may mean that existing test

**Table 6: Qualitative case study showing test case pass/fail predictions help code generation. TIKTOC leverages its knowledge of which test cases are predicted to fail to predict code that is similar to the ground-truth. As a result, the predicted code shows the student's error in assuming the three input integers are sorted. Subset of test cases shown. Code indentation changed for brevity.**

Problem: Given three ints, a, b, and c, one of them is small, one is medium and one is large. Return true if the three values are evenly spaced, so the difference between small and medium is the same as the difference between medium and large.

| Test Case Input | ✓/✗ Preds | | Ground-truth Student Code | OKT Predicted Student Code | TIKTOC Student Code |
|---|---|---|---|---|---|
| | Ground-truth | TIKTOC | | | |
| 2, 4, 6 | ✓ | ✓ | `public boolean evenlySpaced (int a, int b, int c) { if((a-b)==(b-c)) return true; else return false; }` | `public boolean evenlySpaced (int a, int b, int c) { if(Math.abs(a-b)==Math.abs(b-c) ||Math.abs(b-a)==Math.abs(a-c) ||Math.abs(a-c)==Math.abs(c-b)){ return true; } return false; }` | `public boolean evenlySpaced (int a, int b, int c) { int dif; dif = c-b; if( dif == b-a){ return true; } else{ return false; } }` |
| 4, 6, 2 | ✗ | ✗ | | | |
| 4, 6, 3 | ✓ | ✓ | | | |
| 6, 2, 4 | ✗ | ✗ | | | |
| 6, 2, 8 | ✓ | ✓ | | | |
| 2, 2, 2 | ✓ | ✓ | | | |
| 2, 2, 3 | ✓ | ✓ | | | |

cases do not cover enough diversity among student codes. One can verify this postulate by checking clusters among the predicted code, following the visualization approach used in [23]. In this case, instructors can explore creating new and harder test cases, possibly targeting edge cases, such as using an empty string as input, before verifying the predictions with TIKTOC again. For student support, if a student struggles to solve a problem, teachers can use TIKTOC to find students who also failed similar test cases but ultimately succeeded, and use their code trajectories to provide targeted, personalized hints.

## 7 Conclusions and Future Work

In this paper, we proposed a challenging new KT task for open-ended coding tasks, which analyzes and predicts whether a student passes each test case in their code submission. To benchmark existing and new KT methods on this task, we have publicly released an augmented version of the real-world CodeWorkout dataset with test cases. We detailed TIKTOC, a novel KT method for this task, which uses a multi-task learning objective to jointly optimize 1) test case pass/fail prediction accuracy and 2) student code prediction accuracy. Through extensive experiments, we showed that TIKTOC outperforms existing state-of-the-art KT methods adapted to this task. Through qualitative case studies, we showed that our multi-task learning setup is effective and discussed potential use cases in real-world CS educational scenarios. To the best of our knowledge, TIKTOC is among the first attempts to leverage test cases to model student learning open-ended coding tasks. We identify several limitations and avenues for future work. First, we can explore ways to effectively leverage the exact content of test cases to improve pass/fail prediction accuracy. Second, we can explore providing the ground-truth test case pass/fail labels of previous attempts by a student to the LSTM to estimate their knowledge state like DKT [31]. Third, test cases can be labeled with knowledge concepts (KCs), with a training objective encouraging test case/KC learning curves (student error rates) to follow the power law of practice [40, 45]. Fourth, TIKTOC does not explicitly control for fairness across students from different demographic groups and could incorporate fairness regularization into the training objective [52].

## Acknowledgments

This work is partially supported by the NSF under grants 2215193, 2237676, and 2418657.

## References

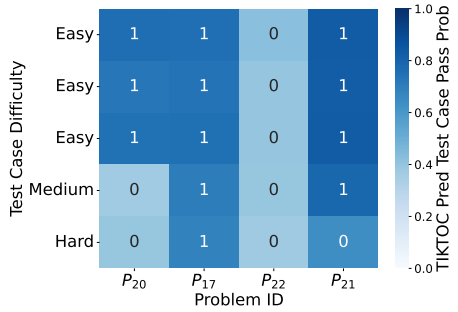
[1] Ghodai Abdelrahman, Qing Wang, and Bernardo Nunes. 2023. Knowledge tracing: A survey. *Comput. Surveys* 55, 11 (2023), 1–37.
[2] Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. 2020. Synthesizing Tasks for Block-Based Programming. In *Advances in Neural Information Processing Systems (NeurIPS)* (Vancouver, BC, Canada). Article 1874, 12 pages.
[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[4] John R Anderson and Robin Jeffries. 1985. Novice LISP errors: Undetected losses of information from working memory. *Human–Computer Interact.* 1, 2 (1985), 107–131.
[5] John Seely Brown and Richard R Burton. 1978. Diagnostic models for procedural bugs in basic mathematical skills. *Cogn. sci.* 2, 2 (1978), 155–192.
[6] Challenge Organizers. 2021. The 2nd CSEDM Data Challenge. Online: https://sites.google.com/ncsu.edu/csedm-dc-2021/.
[7] Albert Corbett and John Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model. User-adapted Interact.* 4, 4 (Dec. 1994), 253–278.
[8] DataShop@CMU. 2021. Dataset: CodeWorkout data Spring 2019. Online: https://pslcdatashop.web.cmu.edu/Files?datasetId=3458.
[9] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 345–356. https://doi.org/10.1145/2884781.2884786
[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).
[11] Stephen Edwards and Zhiyi Li. 2016. Towards Progress Indicators for Measuring Student Programming Effort during Solution Development. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. ACM, Koli Finland, 31–40.
[12] Molly Q Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. 2018. Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics. In *Proc. CHI Conf. Human Factors Comput. Syst.* 1–12.
[13] Nigel Fernandez and Andrew Lan. 2024. Interpreting Latent Student Knowledge Representations in Programming Assignments. In *Proceedings of the 17th*

**Table 7: Heatmap of test case pass/fail predictions by TIKTOC for consecutive student attempts on four problems involving multiple if/else conditionals. Ground-truth pass/fail labels are shown as 1/0. TIKTOC accurately predicts test case pass/fail outcomes.**

Problem 20: Write a function in Java that implements the following logic: Given 3 int values, a, b, and c, return their sum. However, if one of the values is the same as another of the values, it does not count towards the sum.

Problem 21: Write a function in Java that implements the following logic: Given 3 int values, a, b, and c, return their sum. However, if one of the values is 13 then it does not count towards the sum and values to its right do not count. So for example, if b is 13, then both b and c do not count.

TIKTOC Test Case ✓/✗ Prediction Heatmap




TIKTOC Pred Student Code for $P_{20}$

```
public int loneSum
  (int a, int b, int c) {
  if (a == b) {
    return c;
  }
  else if (a == c) {
    return b;
  }
  else if (b == c) {
    return a;
  }
  else {
    return a + b + c;
  }
}
```

TIKTOC Pred Student Code for $P_{21}$

```
public int luckySum
  (int a, int b, int c) {
  if (a == 13) {
    return c;
  }
  else if (b == 13) {
    return a;
  }
  else if (c == 13) {
    return a + b;
  }
  else {
    return a + b + c;
  }
}
```

*International Conference on Educational Data Mining*, Benjamin PaaÄŸen and Carrie Demmans Epp (Eds.). International Educational Data Mining Society, Atlanta, Georgia, USA, 933–940. https://doi.org/10.5281/zenodo.12730003

[14] Aritra Ghosh, Neil Heffernan, and Andrew S Lan. 2020. Context-Aware Attentive Knowledge Tracing. In *Proc. ACM SIGKDD*. 2330–2339.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

[16] Muntasir Hoq, Sushanth Reddy Chilla, Melika Ahmadi Ranjbar, Peter Brusilovsky, and Bita Akram. 2023. SANN: programming code representation using attention neural network with optimized subtree extraction. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 783–792.

[17] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=nZeVKeeFYf9

[18] Alex Kendall, Yarin Gal, and Roberto Cipolla. 2018. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7482–7491.

[19] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023. Automated Program Repair Using Generative Models for Code Infilling. In *International Conference on Artificial Intelligence in Education (AIED)*.

[20] Nischal Ashok Kumar and Andrew Lan. 2024. Improving Socratic Question Generation using Data Augmentation and Preference Optimization. *Proceedings of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA)* (2024).

[21] Nischal Ashok Kumar and Andrew Lan. 2024. Using Large Language Models for Student-Code Guided Test Case Generation in Computer Science Education. *AI4ED workshop at AAAI Conference on Artificial Intelligence* (2024).

[22] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2016. A Diversity-Promoting Objective Function for Neural Conversation Models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kevin Knight, Ani Nenkova, and Owen Rambow (Eds.). Association for Computational Linguistics, San Diego, California, 110–119.

[23] Naiming Liu, Zichao Wang, Richard Baraniuk, and Andrew Lan. 2022. Open-ended Knowledge Tracing for Computer Science Education. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 3849–3862. https://doi.org/10.18653/v1/2022.emnlp-main.254

[24] Qi Liu, Zhenya Huang, Yu Yin, Enhong Chen, Hui Xiong, Yu Su, and Guoping Hu. 2019. Ekt: Exercise-aware knowledge tracing for student performance prediction. *IEEE Trans. Knowl. Data Eng.* 33, 1 (2019), 100–115.

[25] AI @ Meta Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[26] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[27] Ye Mao, Yang Shi, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2021. Knowing" When" and" Where": Temporal-ASTNN for Student Learning Progression in Novice Programming Tasks. *International Educational Data Mining Society*. (2021).

[28] Shalini Pandey and George Karypis. 2019. A self attentive model for knowledge tracing. In *Proc. Int. Conf. Educ. Data Mining*. 384–389.

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[30] Zach A Pardos and Neil T Heffernan. 2010. Modeling individualization in a Bayesian networks implementation of knowledge tracing. In *Proc. Int. Conf. User Model. Adaptation Personalization*. 255–266.

[31] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. *Advances in neural information processing systems* 28 (2015).

[32] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proc. ACM conf. learn. Scale*. 195–204.

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[34] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[35] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education (IJAIED)* 27, 1 (2017).

[36] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *ACM Conference on International Computing Education Research (ICER)*.

[37] Martha Shaka, Diego Carraro, and Kenneth Brown. 2024. Error Tracing in Programming: A Path to Personalised Feedback. In *Proceedings of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2024)*, Ekaterina Kochmar, Marie Bexte, Jill Burstein, Andrea Horbach, Ronja Laarmann-Quante, Anaïs Tack, Victoria Yaneva, and Zheng Yuan (Eds.). Association for Computational Linguistics, Mexico City, Mexico.

[38] Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. 2022. Code-DKT: A code-based knowledge tracing model for programming tasks. *International Conference on Educational Data Mining (EDM)* (2022).

[39] Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. 2022. Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks. In *Proceedings of the 15th International Conference on Educational Data Mining*, Antonija Mitrovic and Nigel Bosch (Eds.). International Educational Data Mining Society, Durham, United Kingdom, 50–61. https://doi.org/10.5281/zenodo.6853105

[40] Yang Shi, Robin Schmucker, Min Chi, Tiffany Barnes, and Thomas Price. 2023. KC-Finder: Automated knowledge component discovery for programming problems.

In *International Conference on Educational Data Mining (EDM)*.

[41] Yang Shi, Krupal Shah, Wengran Wang, Samiha Marwan, Poorvaja Penmetsa, and Thomas Price. 2021. Toward semi-automatic misconception discovery using code embeddings. In *International Learning Analytics and Knowledge Conference (LAK)*. 606–612.

[42] Dongmin Shin, Yugeun Shim, Hangyeol Yu, Seewoo Lee, Byungsoo Kim, and Youngduck Choi. 2021. Saint+: Integrating temporal features for ednet correctness prediction. In *11th Int. Learn. Analytics Knowl. Conf.* 490–496.

[43] Adish Singla and Nikitas Theodoropoulos. 2022. From {Solution Synthesis} to {Student Attempt Synthesis} for Block-Based Visual Programming Tasks. *arXiv preprint arXiv:2205.01265* (2022).

[44] John P Smith III, Andrea A DiSessa, and Jeremy Roschelle. 1994. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *j. learn. sci.* 3, 2 (1994), 115–163.

[45] George S Snoddy. 1926. Learning and stability: a psychophysiological analysis of a case of motor learning with clinical applications. *Journal of Applied Psychology* 10, 1 (1926), 1.

[46] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and scalable detection of logical errors in functional programming assignments. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[47] W. J. van der Linden and R. K. Hambleton. 2013. *Handbook of Modern Item Response Theory*. Springer Science & Business Media.

[48] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning. *Int. Educ. Data Mining Soc.* (2017).

[49] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[50] Yang Yang, Jian Shen, Yanru Qu, Yunfei Liu, Kerong Wang, Yaoming Zhu, Weinan Zhang, and Yong Yu. 2020. GIKT: A Graph-based Interaction Model for Knowledge Tracing. In *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*.

[51] Michael V Yudelson, Kenneth R Koedinger, and Geoffrey J Gordon. 2013. Individualized bayesian knowledge tracing models. In *Int. Conf. artif. intell. educ.* Springer, 171–180.

[52] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2017. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *26th International Conference on World Wide Web (WWW)*. 1171–1180.

[53] Jiani Zhang, Xingjian Shi, Irwin King, and Dit-Yan Yeung. 2017. Dynamic key-value memory networks for knowledge tracing. In *Proc. Int. Conf. World Wide Web*. 765–774.

[54] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. (2019), 783–794.

[55] R. Zhu, D. Zhang, C. Han, M. Gaol, X. Lu, W. Qian, and A. Zhou. 2022. Programming Knowledge Tracing: A Comprehensive Dataset and A New Model. In *2022 IEEE International Conference on Data Mining Workshops (ICDMW)*. 298–307.