Automated Knowledge Component Generation and Knowledge Tracing for Coding Problems

Anonymous ACL submission

Abstract

Knowledge components (KCs) mapped to problems help model student learning, tracking their mastery levels on fine-grained skills thereby facilitating personalized learning and feedback in online learning platforms. However, crafting and tagging KCs to problems, traditionally performed by human domain experts, is highly labor-intensive. We present a fully automated, LLM-based pipeline for KC generation and tagging for open-ended programming problems. We also develop an LLM-based knowledge tracing (KT) framework to leverage these LLM-generated KCs, which we refer to as KCGen-KT. We conduct extensive quantitative and qualitative evaluations on a real-world student code submission dataset. We find that KCGen-KT outperforms existing KT methods and human-written KCs on future student response prediction. We investigate the learning curves of generated KCs and show that LLM-generated KCs result in a better fit than human-written KCs under a cognitive model. We also conduct a human evaluation with course instructors to show that our pipeline generates reasonably accurate problem-KC mappings.

1 Introduction

In student modeling, an important task is to map problems (or items or questions) to specific skills or concepts, referred to as knowledge components (KCs). KCs provide an invaluable resource to model student learning (Bier et al., 2014), estimating their mastery levels (Corbett and Anderson, 1994) on fine-grained units of knowledge. Accurately estimating student mastery levels on KCs helps enable both 1) teacher feedback, by showing this information in teacher dashboards, and 2) adaptive and personalized learning in online learning platforms or intelligent tutoring systems (Huang et al., 2020), by tailoring instructions and content sequencing according to student knowledge levels. Identifying fine-grained KCs students struggle (Rivers et al., 2016) with also enables content designers to develop targeted instructional content and practice problems for students.

KCs are typically crafted by human domain experts, who also tag problems with KCs that students need to master to solve the problem correctly. This process can be highly labor-intensive, prone to bias and errors, and may not be scalable. There exist solutions to automate parts of this process using Natural Language Processing (NLP) tools, usually employing classification algorithms (Pardos and Dadu, 2017), to tag KCs to problems, which relies on having a predefined set of KCs. Recent advances in Large Language Models (LLMs) have shown potential in developing automated approaches for KC *identification* in addition to tagging, in domains such as math (Ozyurt et al., 2024) and science (Moore et al., 2024). Automatically generating KCs is challenging since KCs need to satisfy various criteria including being relevant to problems, being specific enough to provide teacher and student support, and being generalizable across settings. Another important aspect is that they need to satisfy cognitive science principles, i.e., student error rates on a KC should decrease as they attempt it more times, according to the power law of practice (Snoddy, 1926).

Unlike other domains, generating KCs for open-ended programming problems that are common in the domain of computer science education has unique challenges. Writing code is inherently nonlinear, with complex interactions between programming concepts and skills, and requires students to construct functioning code from scratch. Moreover, a programming problem can often have multiple valid solutions using different strategies, which may cover different sets of KCs. Prior work (Hosseini and Brusilovsky, 2013) uses a Java parser to convert a solution program into an Abstract Syntax Tree (AST) and reports ontological concepts at the lowest level as KCs. LLMs, with their advanced

Problem: The number 6 is a truly great number. Given two int values, a and b, return true if either one is 6. Or if their sum or difference is 6. Note: the function Math.abs(num) computes the absolute value of a number.		
Representative Solution Code	Generated KCs	Human-written KCs
`public boolean love6(int a, int b){ if (a == 6 || b == 6){ return true; } else if ((a + b) == 6 || Math.abs(a - b) == 6) { return true; } else{ return false; } }`	If and else if statement Basic arithmetic operations Logical operators Numerical comparisons Absolute value computation	If/Else Math $(+ - * /)$ LogicAndNotOr LogicCompare

Table 1: Example programming problem from the CodeWorkout dataset with a sample student solution code, comparing KCs generated by our KCGen-KT framework to human-written KCs.

programming and reasoning abilities, are yet to be tested for automated KC generation and tagging for programming problems. Other recent works (de Alencar et al., 2025) use AST root nodes in student code submissions as KCs and show that these KCs lead to good learning curve fit under student models (Pavlik et al., 2009). Due to spatial constraints, see Section A in the Appendix for a more detailed review of related work.

1.1 Contributions

In this paper, we explore using LLMs to automatically generate KCs for open-ended programming problems. We also develop an LLM-based knowledge tracing (KT) framework to leverage these LLM-generated KCs, which we refer to as **KCGen-KT**[1]. Our contributions are summarized as follows:

1. We develop a fully automated, LLM-based pipeline for KC generation and tagging. We first select a diverse set of representative student code submissions to each problem and then prompt GPT-4o (OpenAI, 2024), an advanced, proprietary LLM, to identify KCs that are required to solve the problem. Then, to aggregate KCs across problems and de-duplicate similar ones, we cluster KCs on semantic similarity, followed by summarizing each cluster into a KC description. Finally, we automatically tag problems with KCs according to the clustering results. Table 1 shows an example problem with the set of LLM-generated KCs.

2. We develop an LLM-based KT method to leverage the textual descriptions of the generated KCs for the KT task. Our method explicitly captures student mastery levels on each KC and is thus interpretable, while also predicting both the actual student code submission and its correctness.

3. We conduct an extensive quantitative and qualitative evaluation on the CodeWorkout dataset (DataShop, 2021) that contains real-world student code submissions to open-ended programming problems. Results show that KCGen-KT outperforms existing KT methods and human-written KCs on predicting future student performance. We also investigate the learning curves for these KCs and show that LLM-generated KCs have a comparable level of fit to human-written KCs under the performance factor analysis model. We also conduct a human evaluation to show that the KC tagging accuracy of our pipeline is reasonably accurate to human instructors.

[1]We will make our code publicly available.

2 Methodology

We now detail our automated LLM-based approach to generate KCs for programming problems, and then introduce KCGen-KT, a strong KT method leveraging the semantics of the generated KCs to improve student performance prediction.

2.1 Automated KC Generation

For KC generation, we use GPT-4o (OpenAI, 2024), an advanced proprietary LLM with strong reasoning and programming abilities. Illustrated in Figure 1, we generate KCs for a programming problem following 3 key steps: 1) generating KCs associated with each problem and their descriptions separately through few-shot prompting, 2) cluster KCs across all problems, and 3) summarizing each cluster to obtain a finalized description of each KC. We detail these steps below.

Initial KC Generation For each programming problem, we prompt GPT-4o in a chain-of-thought manner, to generate a list of KCs that capture the underlying skills or concepts necessary to solve the problem. We also include a few carefully constructed in-context examples in our prompt as few-

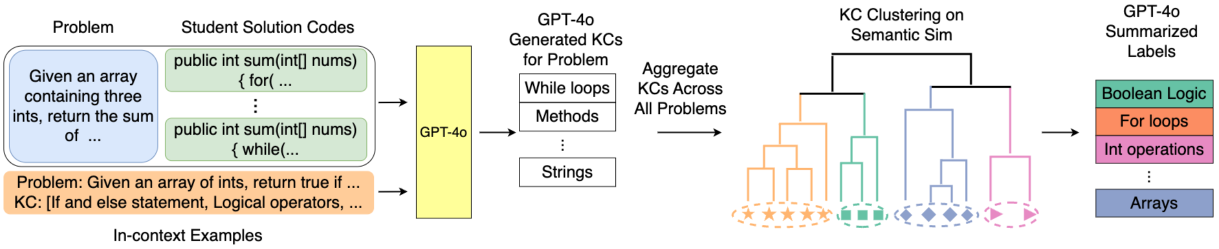

Figure 1: Illustration of our three-step automated KC generation and tagging pipeline.

shot demonstrations. We prompt GPT-4o to convert human-written topic tags, such as "If-else", from the CodeWorkout dataset into more fine-granular natural language descriptions, and use them as the in-context examples. We instruct the model to reason step-by-step: to first identify why a particular KC is relevant to the problem, and then generate a clear textual description of the KC. To help the LLM better understand what is required to solve a problem, we include correct student submissions as examples in the prompt. Since programming problems can often be solved in multiple valid ways, we include diverse examples to ensure comprehensive coverage of relevant KCs. Therefore, we apply a clustering algorithm to the CodeBERT (Feng et al., 2020) embeddings of all correct student submissions and sample one per cluster, with the number of clusters controlling the diversity of examples. Empirically, we find that this approach yields fine-grained, function-level KC descriptions. See Appendix E for the exact prompt used for all steps in our KC generation pipeline.

Clustering KCs and Controlling Abstraction Level The KCs generated for each problem are initially fine-grained, often describing specific function-level skills or concepts. To control the level of abstraction and obtain more generalizable KC descriptions, we first compute the Sentence-BERT (Reimers and Gurevych, 2019) embedding of the textual description of each KC, then apply Hierarchical Agglomerative Clustering (HAC) using cosine similarity as the distance function. By adjusting the number of clusters, we can flexibly merge semantically similar KCs into broader categories, effectively controlling the abstraction level of our KC descriptions. This clustering process enables us to move from detailed skill-level descriptions to higher-level conceptual groupings, aligning the KCs with different pedagogical or analytical goals depending on the downstream application.

Labeling KC Clusters Finally, we label each KC cluster by prompting GPT-4o to generate a single, informative name that represents the cluster. We use a chain-of-thought prompt that guides the model to first reason whether any KC in the cluster can represent the entire group. If such a KC exists, it is selected as the cluster label; otherwise, the model is instructed to synthesize a concise description that captures the shared meaning of KCs in the cluster. This process yields a final set of generated KCs across problems at the desired level of abstraction. As a final step, we obtain problem-KC mappings, i.e., a Q-matrix (Barnes, 2005), by mapping each initially generated KC for each problem to its corresponding summarized cluster label.

2.2 Improving Knowledge Tracing via LLM-generated KCs

We now detail KCGen-KT, a novel LLM-based KT method that exploits KC semantics and explicitly models student mastery levels on each KC.

KT Problem Formulation For open-ended programming problems, we define each student response to a problem as $x_t := (p_t, \{w_t^i\}, c_t, a_t)$, where p_t is the textual statement of the problem, $\{w_t^i\}$ are the KCs associated with the problem, c_t is the student code submission, and a_t is the correctness of the submission; in most existing KT methods, a_t is treated as binary-valued (correct/incorrect). Therefore, our goal is to estimate a student's mastery level of each KC given their past responses, $x_0, \ldots, x_t$, and use this estimate to predict both 1) the overall binary-valued correctness $a_{t+1} \in \{0, 1\}$ and 2) the open-ended code c_{t+1} submitted by the student on their next attempted problem p_{t+1}. Following previous work (Shi et al., 2022), $a_t = 1$ if the student-submitted code passes all test cases associated with the problem, and $a_t = 0$ otherwise.

KCGen-KT KCGen-KT leverages the KCs associated with a problem in two ways: 1) by improving the problem representation using the semantic information of KCs, and 2) by improving the student representation by building an interpretable student profile modeling student mastery levels on KCs.

Following TIKTOC (Duan et al., 2025), we use an open-source LLM, Llama 3 (Llama Team, 2024), as the backbone to predict both the overall correctness and actual open-ended student code in a token-

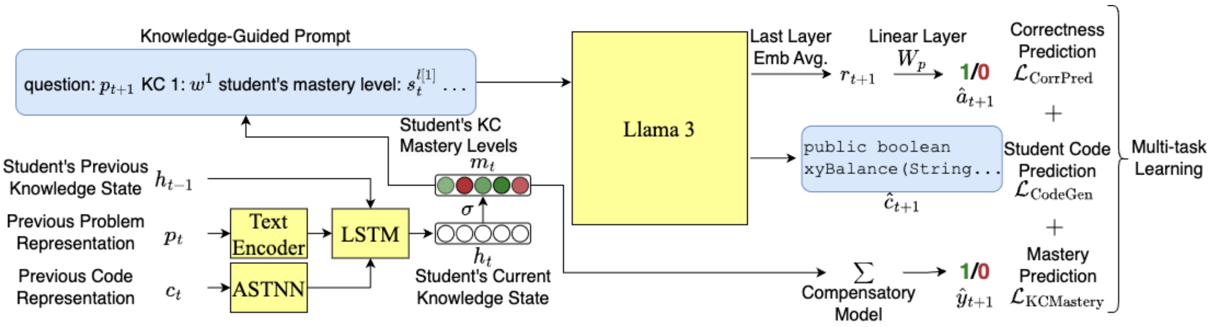

Figure 2: Overview of our KCGen-KT's model with the Llama 3 LLM as the backbone. KCGen-KT leverages KC semantics, tracking student mastery levels on each KC, to predict both correctness and the student code submission.

by-token manner, in a multi-task learning approach. KCGen-KT differs from OKT (Liu et al., 2022) by leveraging the content of the KCs, and from Code-DKT (Shi et al., 2022) by using text embedding methods to embed the textual problem statement.

Student Knowledge on KCs For each student, at each timestep t, KCGen-KT updates the student's 512-dimensional knowledge state vector $h \in \mathcal{R}^{512}$, through a long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) network as in DKT (Piech et al., 2015), given by $h_t = \text{LSTM}(h_{t-1}, p_t, c_t)$. This knowledge state h_t is compressed into a k-dimensional mastery vector $m_t \in [0, 1]^k$, where k is the total number of KCs, through a linear layer with weights W_m and bias b_m, followed by a sigmoid function to map the values of m_t to be in the range of $[0, 1]$, given by $m_t = \sigma(W_m h_t + b_m)$. Each dimension j of m_t denotes a student's mastery level on the jth unique KC, with larger values denoting higher mastery.

Predictions To use LLMs to predict the student response to the next problem, we need to connect student KC knowledge with the textual input space of LLMs. Therefore, following previous work (Fernandez et al., 2024; Liu et al., 2023), we transform KC mastery levels into *soft* text tokens, i.e.,

$$s_t^j = m_t^j \cdot \text{emb}^{\text{true}} + (1 - m_t^j) \cdot \text{emb}^{\text{false}},$$

where emb^{true} and $\text{emb}^{\text{false}}$ are the embeddings of the text tokens "true", and "false", respectively. In other words, we use student KC mastery levels m_t^j to combine two hard, discrete text tokens ("true" and "false") into a differentiable soft token s_t^j, to enable the flow of gradients during training. We pass this student knowledge information using the input format of `KC 1: <`w^1`>. The student's mastery level on <`w^1`> is:` s_t^1. ... into the LLM for prediction tasks.

Knowledge-Guided Response Prediction We construct our LLM prompt for the next response prediction by including both 1) the textual statement of the next problem and 2) student mastery levels on the KCs associated with the problem, as `question:` p_t`. <KCs with student mastery levels>`.

To predict the binary-valued correctness of the next student response, we average the hidden states of the last layer of Llama 3 that correspond to only the input (knowledge-guided prompt) to obtain a representation r, transformed for correctness prediction using a linear transformation matrix W_p and a sigmoid function, given by $\hat{a}_{t+1} = \sigma(W_p \cdot r)$. We minimize the binary cross entropy (BCE) loss (for one response):

$$\mathcal{L}_{\text{CorrPred}} = a_{t+1} \cdot \log \hat{a}_{t+1} + (1 - a_{t+1}) \cdot \log(1 - \hat{a}_{t+1}).$$

To predict student code, we feed the knowledge-guided prompt into Llama 3 to generate the predicted code $\hat{c}$ token-by-token. We minimize

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

where N is the number of tokens in the student code. θ denotes the set of learnable parameters, which includes the KT model, the linear layer with weights W_m and bias b_m for student mastery levels, and the parameters of the finetuned Llama 3.

Promoting Interpretability To promote interpretability of the student KC knowledge parameters, we use a compensatory model (Maier et al., 2021) and take the average of individual student KC mastery levels to obtain an overall mastery level $\hat{y}_{t+1} = \frac{1}{\sum_{k=1}^{K} \mathbb{I}(w_k)} \sum_{k=1}^{K} m_t^k \cdot \mathbb{I}(w_k)$, where the indicator function $\mathbb{I}(w_k)$ is 1 if the KC w_k is associated with the problem, and 0 otherwise. Empirically, we found that averaging over KC masteries performed better than taking a product over them, consistent with findings in prior work (Maier et al., 2021). We then minimize the BCE loss between this overall KC mastery level for this problem and its binary-valued correctness,

$$\mathcal{L}_{\text{KC}} = a_{t+1} \cdot \log \hat{y}_{t+1} + (1 - a_{t+1}) \cdot \log(1 - \hat{y}_{t+1}).$$

This loss regularizes the model to be monotonic, i.e., high knowledge on KCs corresponds to a high probability of a correct response, thus promoting the interpretability of m_t^j.

Multi-task Learning Objective Following previous work (Duan et al., 2025) showing multiple objectives in KT are mutually beneficial to each other, our final multi-task training objective minimizes a combination of all three losses together, with a balancing parameter $\lambda \in [0, 1]$ controlling the importance of the losses, as

$$\mathcal{L}_{\text{KCGen-KT}} = \lambda(\mathcal{L}_{\text{CodeGen}} + \mathcal{L}_{\text{CorrPred}}) + (1-\lambda)\mathcal{L}_{\text{KC}},$$

where losses are averaged over code submissions by all students to all problems.

3 Experimental Evaluation

We now detail our quantitative experimental settings to evaluate KCGen-KT on future student code submission correctness prediction.

Dataset Details The CodeWorkout (DataShop, 2021) dataset was first used in the Second CSEDM Data Challenge (Challenge Organizers, 2021) and contains actual open-ended code submissions from real students, collected from an introductory Java programming course, together with problem textual statements and human-written KC tags (estimated programming concepts) on each problem. In total, there are 246 students attempting 50 problems covering various programming concepts including conditionals, and loops, among others. Following prior work (Shi et al., 2022), we only analyze students' first submissions to each problem, leading to a total of $10,834$ code submissions.

Metrics For the binary-valued correctness prediction task, following (Shi et al., 2022), we use standard metrics such as **AUC**, **accuracy**, and **F1** score. For the student code prediction task, following (Liu et al., 2022), we measure the similarity between generated student code and ground-truth student code using **CodeBLEU** (Ren et al., 2020), a variant of the classic text similarity metric BLEU (Papineni et al., 2002). This metric is customized for code and measures both syntactic and semantic similarity between two pieces of code.

Baselines In terms of KCs, we compare our generated KCs against human-written KCs that are available in the CodeWorkout dataset. We test a version of KCGen-KT by replacing our LLM-generated KCs with human-written KCs and keeping the KT method unchanged, which we refer to as **KCGen-KT (Human-written KCs)**. In terms of KT methods, we adapt Test case-Informed Knowledge Tracing for Open-ended Coding (TIKTOC) (Duan et al., 2025), a recent, strong KT method for programming, as the main baseline. TIKTOC also uses Llama 3 as the backbone and a multi-task learning setup to jointly predict the exact code token-by-token and whether it passes each test case. We slightly modify it for our KT task, replacing test case prediction with overall code correctness prediction, by reducing the dimension of the prediction head from the number of test cases to one, for overall correctness prediction only. We refer to the resulting method as **TIKTOC***. We also use **Code-DKT**, a popular KT method for programming that leverages the content of student code, to predict the overall correctness of student code submissions. As a sanity check, to estimate a lower bound of performance on our KT task thereby providing a sense of task difficulty, we include two simple baselines: **Random**, which simply predicts the overall binary-valued correctness of a student code randomly with equal probability, and **Majority**, which simply predicts the majority correctness label (incorrect) among students for each problem.

Experimental Setup For the KT method component of KCGen-KT as well as for all KT baselines, to ensure a fair comparison, we use the instruction-tuned version of Llama 3 (Llama Team, 2024) with 8B parameters as the base LLM and a frozen ASTNN (Zhang et al., 2019) as the code embedding model. See Appendix B for detailed parameter settings. We repeat our experiments across 5 random train-validation-test data splits.

4 Results, Analysis, and Discussion

We now discuss our quantitative evaluation results and qualitatively analyze the estimated student KC mastery levels and predicted code. We also analyze the learning curves, conduct an ablation study, and investigate the characteristics of KCs across different levels of abstraction.

4.1 Quantitative Evaluation

KCGen-KT outperforms baselines Table 2 shows the average performance (and standard deviation) on our two KT tasks: binary correctness prediction and student code generation for all methods. For KCGen-KT, we report results using the best-performing configuration, which generates 60 KCs

Model	KT Correctness Pred.			Code Pred.
	AUC ↑	F1 Score ↑	Accuracy ↑	CodeBLEU ↑
Random	0.499	0.368	0.506	–
Majority	0.500	0.644	0.526	–
Code-DKT (Shi et al., 2022)	$0.766_{\pm 1.8\%}$	$0.672_{\pm 3.3\%}$	$0.724_{\pm 1.0\%}$	–
TIKTOC* (Duan et al., 2025)	$0.788_{\pm 1.3\%}$	$0.666_{\pm 3.0\%}$	$0.726_{\pm 1.3\%}$	$0.507_{\pm 1.5\%}$
KCGen-KT(Human-written KCs)	$0.797_{\pm 1.6\%}$	$0.706_{\pm 2.6\%}$	$0.727_{\pm 2.0\%}$	$0.557_{\pm 2.8\%}$
KCGen-KT(Generated KCs)	$\mathbf{0.816}_{\pm 1.2\%}$	$\mathbf{0.727}_{\pm 2.7\%}$	$\mathbf{0.746}_{\pm 1.2\%}$	$\mathbf{0.580}_{\pm 1.8\%}$

Table 2: Comparing KCGen-KT against baselines on KT performance across all metrics. KCGen-KT, especially with LLM-generated KCs, outperforms other KT methods. Best performance is in **bold** and second best is underlined.

after clustering, based on 5 student submissions per problem. We see that the Random and Majority baselines perform poorly, which suggests that the correctness prediction KT task is inherently difficult. Our proposed framework, KCGen-KT with either human-written or generated KCs, outperforms other strong KT methods that do not use KCs, including TITKOC* and Code-DKT. This observation suggests that for KT methods that use LLMs as the backbone, leveraging the semantic information in KC descriptions improves KT performance. More importantly, KCGen-KT with our generated KCs outperforms human-written KCs, by a consistent margin on both tasks, with statistical significance ($p < 0.05$). This observation shows that high-quality KC descriptions and accurate tagging are key to improving downstream KT performance. The performance gap is more evident in code prediction, which shows that semantically informative KCs, as evident from Table 1, are especially important to LLMs in generative tasks.

Less Fine-grained KCs hurt performance To investigate the impact of KC granularity on model performance, we experiment with three levels of abstraction. We consider the 103 unique KCs generated by the first step of our pipeline, before clustering, as the most fine-grained (low-level) representation. We then apply the clustering algorithm with the number of clusters equal to 60 and 10 to get two other KC sets with medium and high abstraction levels. As the number of clusters decreases, the resulting KC sets become increasingly abstract, forming a hierarchy of representations. We evaluate KCGen-KT's performance using these three KC sets. Table 3 shows that the highest abstraction level yields the lowest performance across all metrics on both tasks. In contrast, medium and low abstraction levels achieve comparable performance, which justifies our choice of using 60 KCs with medium-level abstraction. These results also suggest that overly abstract KCs may not pinpoint the necessary skills in a problem, underscoring the importance of having sufficient granularity in KCs for downstream student modeling tasks.

KC Abstraction Level	AUC	F1	Acc	CodeBLEU
Low	0.815	0.726	0.737	0.572
Medium	**0.816**	**0.727**	**0.746**	**0.580**
High	0.794	0.683	0.708	0.557

Table 3: Comparing different KC abstraction levels. SD omitted due to spatial constraints. Best performance is in **bold** and second best is underlined.

Ablation Study We conduct an ablation study among all components of KCGen-KT. We find that including correct student submissions is crucial; removing it results in a noticeable performance decrease. We also explore the impact of using LLM-generated solutions and switching submissions to AST representation. See Appendix C for details.

4.2 Qualitative Evaluation

Case Study Table 4 shows the estimated KC mastery levels and predicted code submission for a student on a problem in the test set. The low student mastery level on KCs "For loop iteration" and "Array indexing and assignment" results in a run time error in the predicted code by indexing the array outside of its bounds. In contrast, the higher predicted mastery level of other KCs results in the correct implementation of the if and else if statements, proper use of boolean expressions, and accurate application of the logical AND operator. This example shows that informative KC descriptions generated by the LLM can help KCGen-KT make more accurate student code predictions. In practice, the predicted mastery level may offer instructors interpretable insights into the student's understanding of specific programming concepts.

Learning Curve Analysis A common method to assess the quality of KCs is examining how well they match cognitive theory; the expected pattern on the KCs should follow the *power law of practice*, which states that the number of errors should

Problem: Given an array of ints, return true if the array contains two 7s next to each other, or there are two 7s separated by one element, such as with $\{7, 1, 7\}$. Otherwise, return false.		
Predicted Student Code Submission	LLM-generated KC	Mastery
`public boolean has77(int[] nums){ for (int i = 0; i < nums.length - 1; i++){ if (nums[i] == 7 && nums[i + 1] == 7){ return true; } else if (nums[i] == 7 && nums[i + 2] == 7){ return true; } } return false; }`	For loop iteration Array indexing and assignment Boolean logic Logical operators Numerical comparisons If and else if statement	26.1% 28.1% 51.9% 57.6% 59.9% 74.4%

Table 4: Example showing low student knowledge on relevant KCs map to specific errors in predicted student code.

decrease as the amount of practice on certain KCs increases (Newell and Rosenbloom, 2013; Snoddy, 1926). Hence, we compare the error rate across different attempts at KCs and the estimated student KC mastery levels from KCGen-KT. For this experiment, we prompt GPT-4o to label whether each student submission contains an error on each KC. For all incorrect student submissions, we provide the problem statement, the associated KCs, and the student code, and prompt GPT-4o to (1) reason about the errors, (2) generate a corrected version of the code, and (3) assign a binary correctness label to each KC, indicating whether the student made an error on this KC in their submission (See Appendix E for the exact prompt). We acknowledge that although we find this process to be empirically accurate, a formal validation is necessary to verify the accuracy of the KC-level correctness labels.

To plot the curves, at attempt t, we average the binary correctness label (where we use 1 as incorrect) over all students on the problem that represents their t-th attempt at the KC. We also calculate the predicted error rate similarly, using KCGen-KT to estimate the mastery level of each student on each KC at each time step and taking the complement.

Figure 3 shows three representative learning curves among all LLM-generated KCs. In all cases, both the ground truth and predicted error rate curves exhibit a general decreasing trend as the number of attempts increases, consistent with the power law of practice. The first predicted learning curve closely aligns with the ground truth, demonstrating KCGen-KT's ability to accurately capture student learning progressions. The second predicted learning curve matches the ground truth error rates in trend, but has higher overall values. The third curve further exacerbates this discrepancy, for a KC that appears more frequently in the dataset. The ground truth error rates decrease overall but have significant fluctuation, making it difficult to fit the predicted learning curve. The likely reason is that students attempt problems in different orders in the CodeWorkout dataset, with some students skipping certain questions; this variation means that the same attempt may correspond to questions with different difficulties across students, making the average error rate noisy. For a more quantitative evaluation, we follow prior work (Pavlik et al., 2009) and fit PFA models on each KC. Results show that the weighted R^2 metric using the LLM-generated KCs is 0.21, and using the human-written KCs is 0.18. Therefore, the LLM-generated KCs fit the power law of practice slightly better compared to human-written KCs.

KC Ontologies We show a portion of the KC ontology subtree obtained from KCGen-KT, for string-related concepts, in Figure 4. The root node shows the KC at the highest abstraction level; going down the tree from there, we see how the KCs identified increase in granularity at each level. To build the ontology tree, we start from the top-level KC and identify all KC labels from the next level that are semantically included in it, and do it iteratively for all KCs. This mapping defines the parent-child relationships between KCs across different abstraction levels. This example demonstrates the controllable abstraction property of our KC generation pipeline, where adjusting the number of clusters directly controls the granularity of the generated KCs.

5 Human Evaluation

We perform a human evaluation to assess the quality of the generated KCs and the accuracy of problem-KC mappings on a sample of 30 questions. We recruit 5 annotators with experience in teaching college-level programming. To evaluate KC quality, we show annotators both LLM-generated KCs and baseline human-written KCs for each problem, and ask them to indicate their preference between the two sets. On average, the LLM-generated KC sets are preferred in 54.5% of the cases, suggesting

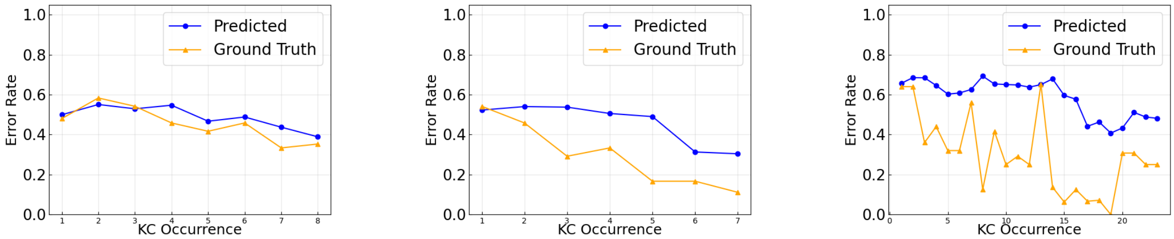

Figure 3: Representative learning curves for three generated KCs (Equality Comparison, String Length Determination, and For Loop Iteration), showing a generally decreasing error rate over attempts. Our KCs result in better model fit (0.21 vs. 0.18 in R^2) than human-written KCs under cognitive models (Pavlik et al., 2009).

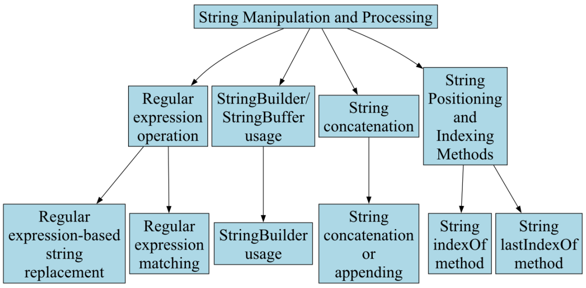

Figure 4: A section of the generated KC ontology (related to `Strings`, at different abstraction levels.

that the generated KC is generally more informative to human instructors than human-written KCs.

To evaluate problem-KC mappings, we ask annotators to label every KC mapped to each problem and perform a two-stage annotation. First, they determine whether the KC is relevant to the problem. For KCs labeled as relevant, they then rate how essential it is to the problem, in three categories: essential, marginal, or non-essential. Based on these annotations, we compute the average proportion of relevant KCs per problem, as well as the average proportions of essential and non-essential KCs per problem across all annotators. Results show that the average percentage of relevant KCs per problem is 92.0% for the LLM-generated set and 91.6% for the baseline. The average percentage of essential KCs per problem is 50.1% for the generated set and 49.5% for the baseline, and the average percentage of non-essential KCs is 31.9% and 33.7%, respectively. These findings suggest that the LLM-based KC tagging is reasonably accurate and comparable to human-labeled baselines, although there remains significant room for improvement. See Appendix D for the detailed annotation rubrics for these tasks and inter-rater agreement results.

Qualitative annotator feedback further reveals several important directions for future work. The LLM-generated KCs are generally easier to interpret and process due to their use of natural language. However, annotators note that some of these KCs could be consolidated into more concise representations, since they occasionally express overlapping concepts. Additionally, because KCs are generated independently for each problem, the LLM may overlook commonly relevant KCs present across multiple problems. Almost all feedback suggests that a human-AI collaboration approach for KC identification is a necessity: we can use KCGen-KT to provide quantitative feedback on downstream KT performance and learning curve fit, while humans merge, split, or edit LLM-generated KC descriptions and tags.

6 Conclusions and Future Work

In this paper, we presented a fully automated, LLM-based pipeline for KC generation and tagging for open-ended programming problems. We also developed an LLM-based KT framework, KCGen-KT, which leverages the textual content of KC descriptions. Through extensive experiments on a real-world student coding dataset, we showed that KCGen-KT outperforms human-written KC labels on KC generation and existing state-of-the-art KT methods on predicting future student performance. We also show that LLM-generated KCs lead to better learning curves than human-written ones. A human evaluation shows that the generated KCs and problem-KC mappings are reasonably accurate to programming instructors.

There are many avenues for future work. First, according to annotator suggestions, a human-in-the-loop KC generation method may address many limitations of KCGen-KT. Second, we can explore elevating the learning curve fit into the objective function and explicitly train the model to follow the power law of practice. Third, we can explore whether our methods can be applicable in other student modeling tasks and domains, including dialogues (Scarlatos et al., 2025), math (Ozyurt et al., 2024), and science (Moore et al., 2024).

Limitations

We identify several technical and practical limitations of our work. First, the main limitation of our automatic KC generation pipeline is its reliance on in-context examples, which necessitates at least one human-written example to generate KCs at the lowest abstraction level. Without such examples, zero-shot prompting with LLMs tends to produce overly general and high-level KCs, since the LLMs are not explicitly trained for the KC generation task. Second, since each problem is associated with multiple KCs, obtaining reliable ground truth KC labels is inherently challenging. The current process of assigning ground truth correctness labels is time-consuming and requires further validation to ensure label quality. Third, the inter-rater agreement in our human evaluation is not high enough, which suggests that there is significant subjectivity in the KC evaluation task among instructors. Future work is needed to examine where this disagreement comes from and revise the evaluation process, possibly by showing KC information in alternative ways or redefining the rubrics. Fourth, we evaluate our method on a single dataset in a single domain, computer science education; applying the KC generation pipeline and KCGen-KT model across multiple datasets and domains such as math would be valuable for assessing generalizability and robustness. Finally, even though we conducted a human evaluation, the real benefit of good KC labeling is to enable students to improve learning outcomes from personalization methods informed by these KCs. Therefore, classroom studies comparing the LLM-generated KCs with human-written KCs in facilitating student progress and maximizing learning outcomes are ultimately needed.

Ethical Considerations

Our goal in this work is to develop a system that can automatically generate knowledge components and integrate them into student modeling frameworks to track individual learning progress. The primary motivation is to reduce the manual effort required from educators in topic selection and KC design, thereby enabling more time and resources to be devoted to personalized student support. However, there is a concern that such systems could replace human educator jobs, which is a shared concern across most domains with AI applications. Another critical risk lies in the quality of the automatically generated knowledge components. If the generated KCs are inaccurate, overly abstract, or misaligned with instructional goals, they may negatively affect student learning outcomes by reinforcing misconceptions or misrepresenting the required skills. Because of these reasons, we recommend that the generated knowledge component be reviewed by experts before being deployed to real students.

References

Tiffany Barnes. 2005. The q-matrix method: Mining student response data for knowledge. In *American association for artificial intelligence 2005 educational data mining workshop*, pages 1–8. AAAI Press, Pittsburgh, PA, USA.

Norman Bier, Sean Lip, Ross Strader, Candace Thille, and Dawn Zimmaro. 2014. An approach to knowledge component/skill modeling in online courses. *Open Learning*, pages 1–14.

Challenge Organizers. 2021. The 2nd CSEDM Data Challenge. Online: https://sites.google.com/ncsu.edu/csedm-dc-2021/.

Albert Corbett and John Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model. User-adapted Interact.*, 4(4):253–278.

DataShop. 2021. Dataset: CodeWorkout data Spring 2019. Online: https://pslcdatashop.web.cmu.edu/Files?datasetId=3458.

Rafaella Sampaio de Alencar, Mehmet Arif Demirtas, Adittya Soukarjya Saha, Yang Shi, and Peter Brusilovsky. 2025. Integrating expert knowledge with automated knowledge component extraction for student modeling. In *Proceedings of the 33rd ACM International Conference on User Modeling, Adaptation and Personalization*.

Zhangqi Duan, Nigel Fernandez, Alexander Hicks, and Andrew Lan. 2025. Test case-informed knowledge tracing for open-ended coding tasks. In *Proceedings of the 15th Learning Analytics and Knowledge Conference, LAK 2025, Dublin, Ireland, March 3-7, 2025*. ACM.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Nigel Fernandez and Andrew Lan. 2024. Interpreting latent student knowledge representations in programming assignments. In *Proceedings of the 17th International Conference on Educational Data Mining*, pages 933–940, Atlanta, Georgia, USA. International Educational Data Mining Society.

Nigel Fernandez, Alexander Scarlatos, Wanyong Feng, Simon Woodhead, and Andrew Lan. 2024. DiVERT: Distractor generation with variational errors represented as text for math multiple-choice questions. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9063–9081, Miami, Florida, USA. Association for Computational Linguistics.

Aritra Ghosh, Neil Heffernan, and Andrew S Lan. 2020. Context-aware attentive knowledge tracing. In *Proc. ACM SIGKDD*, pages 2330–2339.

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

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*, pages 783–792.

Muntasir Hoq, Jessica Vandenberg, Bradford Mott, James Lester, Narges Norouzi, and Bita Akram. 2024. Towards attention-based automatic misconception identification in introductory programming courses. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, pages 1680–1681.

Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, volume 1009, pages 60–63. University of Pittsburgh.

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*.

Yun Huang, Vincent Aleven, Elizabeth McLaughlin, and Kenneth Koedinger. 2020. A general multi-method approach to design-loop adaptivity in intelligent tutoring systems. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part II 21*, pages 124–129. Springer.

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*, pages 3849–3862, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Xin Liu, Muhammad Khalifa, and Lu Wang. 2023. Bolt: Fast energy-based controlled text generation with tunable biases. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 186–200.

AI @ Meta Llama Team. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Cristina Maier, Ryan Baker, and Steve Stalzer. 2021. Challenges to applying performance factor analysis to existing learning systems.

Steven Moore, Robin Schmucker, Tom Mitchell, and John Stamper. 2024. Automated generation and tagging of knowledge components from multiple-choice questions. In *Proceedings of the eleventh ACM conference on learning@ scale*, pages 122–133.

Allen Newell and Paul S Rosenbloom. 2013. Mechanisms of skill acquisition and the law of practice. In *Cognitive skills and their acquisition*, pages 1–55. Psychology Press.

OpenAI. 2024. Hello gpt-4o.

Yilmazcan Ozyurt, Stefan Feuerriegel, and Mrinmaya Sachan. 2024. Automated knowledge concept annotation and question representation learning for knowledge tracing. *arXiv preprint arXiv:2410.01727*.

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

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*, pages 311–318.

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*, pages 255–266.

Zachary A Pardos and Anant Dadu. 2017. Imputing kcs with representations of problem content and context. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, pages 148–155.

Philip I Pavlik, Hao Cen, and Kenneth R Koedinger. 2009. Performance factors analysis–a new alternative to knowledge tracing. In *Artificial intelligence in education*, pages 531–538. Ios Press.

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.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

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. *Preprint*, arXiv:2009.10297.

Kelly Rivers, Erik Harpstead, and Kenneth R Koedinger. 2016. Learning curve analysis for programming: Which concepts do students struggle with? In *ICER*, volume 16, pages 143–151. ACM.

Alexander Scarlatos, Ryan S. Baker, and Andrew Lan. 2025. Exploring knowledge tracing in tutor-student dialogues using llms. In *Proceedings of the 15th Learning Analytics and Knowledge Conference, LAK 2025, Dublin, Ireland, March 3-7, 2025*. ACM.

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*, pages 50–61, Durham, United Kingdom. International Educational Data Mining Society.

Yang Shi, Robin Schmucker, Min Chi, Tiffany Barnes, and Thomas Price. 2023. Kc-finder: Automated knowledge component discovery for programming problems. *International Educational Data Mining Society*.

Yang Shi, Robin Schmucker, Keith Tran, John Bacher, Kenneth Koedinger, Thomas Price, Min Chi, and Tiffany Barnes. 2024. The knowledge component attribution problem for programming: Methods and tradeoffs with limited labeled data. *Journal of Educational Data Mining*, 16(1):1–33.

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.*, pages 490–496.

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):1.

Xinjie Sun, Qi Liu, Kai Zhang, Shen Shuanghong, Lina Yang, and Hui Li. 2025. Harnessing code domain insights: Enhancing programming knowledge tracing with large language models. *Knowledge-Based Systems*, 317:113396.

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, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

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*.

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

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. pages 783–794.

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*, pages 765–774.

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)*, pages 298–307.

A Related Work

A.1 Knowledge Component Generation

Traditional methods for KC creation and tagging rely on human domain experts to identify the knowledge requirements for solving a problem (Bier et al., 2014), a highly time-consuming process. Recent work has proposed automated approaches for KC discovery and tagging, employing data-driven approaches including the Q-matrix method (Barnes, 2005). In programming, (Hosseini and Brusilovsky, 2013) uses a rule-based parser to obtain ASTs with KCs identified at their lowest ontological level, (Rivers et al., 2016) define KCs as nodes in an AST followed by a learning curve analysis to identify KCs students struggle with the most in Python programming, (Hoq et al., 2024) uses an AST-based neural network to identify student misconceptions, (Shi et al., 2024) presents a deep learning approach for KC attribution, and (Shi et al., 2023, 2024) learn latent KCs, lacking textual descriptions, by training deep learning models on KT data enforced with priors from pedagogical theory. Recent advances in LLMs have inspired automated approaches for descriptive KC generation for dialogues (Scarlatos et al., 2025), and problems in math (Ozyurt et al., 2024), and science (Moore et al., 2024). However, we're among the first approaches to present a fully automated, LLM-based pipeline for KC generation and tagging for open-ended programming problems.

A.2 Knowledge Tracing

There exists a wide body of work on KT (Corbett and Anderson, 1994) in the student modeling literature. The classic KT task aims to estimate a student's mastery of KCs from their responses to past problems and use these estimates to predict their future performance. Classic Bayesian knowledge tracing methods (Pardos and Heffernan, 2010; Yudelson et al., 2013) use latent binary-valued variables to represent student KC mastery. With the widespread adoption of neural networks, multiple deep learning-based KT methods were proposed with limited interpretability since student knowledge is modeled as hidden states in these networks. Most of these methods use long short-term memory networks (Hochreiter and Schmidhuber, 1997) or variants (Piech et al., 2015; Shin et al., 2021), with other variants coupling them with memory augmentation (Zhang et al., 2017), graph neural networks (Yang et al., 2020), or attention networks (Ghosh et al., 2020; Pandey and Karypis, 2019). KT methods have been applied to many different educational domains, including programming (Hoq et al., 2023; Shi et al., 2022; Zhu et al., 2022). Recent work has attempted to leverage LLMs to develop generative KT methods predicting exact student responses to programming problems (Duan et al., 2025; Fernandez and Lan, 2024; Liu et al., 2022). (Sun et al., 2025) uses LLM to automatically construct Q-matrices capturing fine-grained KC relationship in knowledge tracing, while their work lacks comparison with code-specific KT baselines, and does not explore LLM-based KC generation. To the best of our knowledge, we are the first to present an LLM-based KT method for programming problems that leverages the textual content of KC descriptions, modeling interpretable student mastery levels on each KC, for improved KT performance.

B Experimental Setup

As detailed in 2.2, we use a LLM as our backbone and we use the instruction-tuned version of Llama 3 with 8B parameters as our model. We use the Parameter Efficient Fine-Tuning (PEFT) library from HuggingFace (Wolf et al., 2020) load Llama 3 and fine-tune it via Low-Rank Adaptation (LoRA) (Hu et al., 2022) ($\alpha = 256$, rank $= 128$, dropout $= 0.05$) using 8-bit quantization. We use the AdamW (Loshchilov and Hutter, 2019) optimizer for LLM and W_m parameter with a batch size of 32 and RMSprop optimizer for the LSTM and perform a grid search to determine the optimal learning rate. In KCGen-KT, we set different learning rates for different model components: $1e-5$ for Llama 3, $5e-4$ for the LSTM model, and $1e-4$ for the W_m and b_m parameters. KCGen-KT converges within 12 training epochs, with each epoch taking 80 minutes on an NVIDIA L40S 48GB GPU. We repeat all our experiments with 5 random train-validation-test data splits for cross-validation.

For the baseline model TIKTOC* which also uses a LLM as the backbone of the model. We use same setup for a fair comparison: we use same Llama model with PEFT to load it and fine-tune it with LoRA ($\alpha = 256$, rank $= 128$, dropout $=$ 0.05) using 8-bit quantization. The optimizers and learning rate used are the same as KCGen-KT except the learning rate for LSTM model is $1e-4$.

For metrics, we use the rou_auc_score and f1_score from scikit-learn library to compute AUC and F1 metrics, respectively. For codeBLEU we adopt the official implementation provided in the microsoft/CodeXGLUE. In addition, we use scipy library to perform the hierarchical agglomerative clustering. All software we use in the development of this work is open source. We are consistent with the terms and intended use of all software and with OpenAI API.

Model	AUC	F1	Acc	CodeBLEU
KCGen-KT	**0.812**	**0.723**	**0.724**	**0.569**
- Student submission = 0	0.789	0.674	0.704	0.529
- Student Code → AST	0.794	0.691	0.725	0.546
- Student Code → Generated	0.810	0.706	0.731	0.557

Table 5: Ablation study of KCGen-KT

C Ablation Study

Table 5 shows the results of the ablation study among all components of KCGen-KT on the dataset. For ablation conditions requiring code submissions for KC generation, we report results using two code solutions since we note that LLM-generated code tends to follow similar problem-solving strategies; we adopt the lowest level of KC abstraction and manually verify the correctness of all generated solutions. We see that including correct submissions is crucial; removing it results in performance decrease on both KT tasks, which suggests that it is difficult to capture all necessary KCs from just the problem statement alone. Using LLM-generated submissions instead of actual student submissions also results in a minor performance decrease, which suggests that actual student code is highly diverse and captures a more complete set of skills required in each problem. We also see that switching submissions to the abstract syntax tree (AST) representation as input to the LLM decreases performance, where we see the generated KCs tend to be less detailed. This result can be explained by LLMs not being heavily pre-trained on AST representations of code. See Appendix E for prompt used for KC generation with AST representation.

Furthermore, we explore the impact of the number of student submissions provided to the LLM on the quality of generated KCs and KT performance. For this ablation, we use the lowest KC abstraction level. Table 6 shows the results, where we see that using fewer submissions, such as 1 and 2, results in worse performance, while the performance increases and saturates after more than 5 submissions. This result can be explained by a smaller number of student submissions failing to capture diverse solution strategies for some problems, thus resulting in an incomplete KC set. As the number of submissions increase, the set of initial KCs before clustering more or less stays the same, and the performance mostly depends on the abstraction level instead.

No. of Solutions	AUC	F1	Acc	CodeBLEU
1	0.804	0.713	0.705	0.563
2	0.812	0.723	0.724	0.569
5	<u>0.815</u>	0.726	**0.737**	<u>0.572</u>
7	0.812	<u>0.726</u>	<u>0.727</u>	**0.573**
10	**0.816**	**0.727**	0.715	0.566

Table 6: Ablation study of KCGen-KT on different No. of solution provided during KC Generation. Best performance is in **bold** and second best is <u>underlined</u>.

D Human Evaluation Details

We conduct a human evaluation to assess the quality of generated KCs and problem-KC mapping accuracy. Our evaluators were volunteers contacted through a research partner and were not compensated monetarily. They were made aware that their annotations would be used in scientific research in AI. Annotators perform two separate tasks, each designed to evaluate a distinct aspect of the KC quality. Below, we describe the annotation rubrics and summarize the evaluation outcomes.

D.1 KC-Set Annotation Per Problem for Problem-KC mapping

The instruction provided to annotators is as follows:

> Which KC-set is better (1 for Generated, 2 for Baseline)? A KC-set is considered better if, overall, it more effectively captures the key concepts that are practiced in the problem and necessary for solving it. Imagine you are an instructor selecting problems for a homework assignment. Which set of concepts would be more informative in deciding whether to include this problem based on the skills it assesses?

To assess inter-rater reliability, we calculate Kappa scores. The overall agreement across all five annotators is relatively low, with an average Kappa score of 0.121. The highest pairwise agreement observed is 0.51, indicating moderate agreement between two annotators. This disagreement suggests differing interpretations among annotators regarding which KCs are most relevant to a given problem. In particular, the task excludes KC understandability as a criterion, focusing instead on relevance and completeness. These differences in evaluative emphasis and background knowledge likely contributed to the observed disagreement.

D.2 Single-KC Annotation Per Problem for Generated KC Quality

In the second task, annotators evaluate the quality of individual KCs generated for each problem by assessing their correctness and instructional relevance. This task focuses on the Problem-KC mappings accuracy.

The annotation rubric is as follows:

> Task: For each KC associated with a problem, provide a binary label for correctness, and if the KC is correct, further classify its level of essentiality.
>
> Correct: if the KC accurately reflects a concept required to understand or solve the problem, and it holds educational value in the context.
>
> Essentiality (only if the KC is Correct): Classify the KC based on its instructional importance using the following three categories:

- Essential: The KC is critical to the problem. It represents a core concept that the problem is explicitly designed to assess. Without understanding this concept, a student would likely struggle to solve the problem.
- Marginal: The KC is relevant but not central. It may influence the decision to assign the problem, but only as a secondary consideration. While the problem can help reinforce this concept, it is not the primary instructional goal.
- Non-Essential: The KC appears in the problem but plays a minimal instructional role. It is not a focus of the problem and would not factor significantly into an instructor's decision to use the problem for teaching purposes.

Results show that the average percentage of relevant KCs per problem is 92.0% for the LLM-generated set and 91.6% for the baseline. The average percentage of essential KCs per problem is 50.1% for the generated set and 49.5% for the baseline, the average percentage of marginal KCs is 31.9% for generated KCs and 33.7% for the baseline KCs and the average percentage of non-essential KCs is 31.9% and 33.7%, respectively. These findings suggest that the LLM-based KC tagging is reasonably accurate and LLM-generated KCs are largely relevant and instructionally meaningful, which implies that LLM is capable of identifying topic-specific concepts with comparable instructional value, but still have room to improve with human-AI collaboration approach.

Annotators complete the evaluation using a custom web-based interface, as shown in figure 5,that presented each problem alongside its associated KC set. For each KC, annotators labeled correctness and essentiality and selected their preferred KC set. The interface was designed to ensure consistency and ease of use across annotators.

E Prompt

E.1 Prompt for KC Generation Pipeline

We show the prompt used for the KC generation for each problem in Table 8, the prompt used for the AST ablation study in Table 9, and the prompt used for cluster summarization in Table 10.

E.2 Prompt for KC Correctness Labeling

We show the prompt used for KC correctness labeling for the learning curve analysis in Table 11.

E.3 Prompt for In-context Example Conversion

We show the prompt to convert the baseline KC list into natural language terms in Table 7.

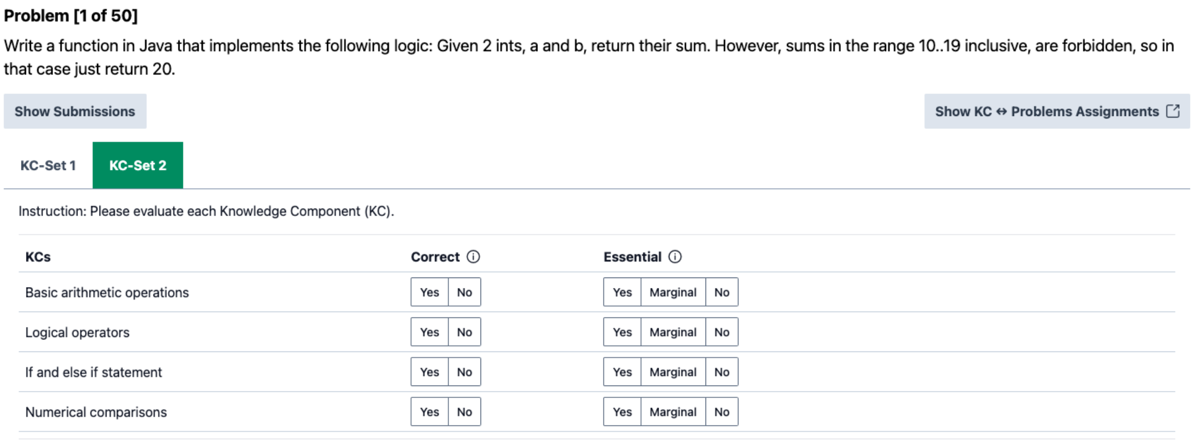

Figure 5: A demo of the interface for human evaluation

System Message:
You are an experienced computer science teacher and education expert. You are provided with a list of human-labeled knowledge components (KCs) associated with programming problems. Your task is to convert each KC into a equivalent natural language term.

User prompt:
The KC list is: [If/Else, NestedIf, While, For, NestedFor, Math+-*/, Math%, LogicAndNotOr, LogicCompareNum, LogicBoolean, StringFormat, StringConcat, StringIndex, StringLen, StringEqual, CharEqual, ArrayIndex, DefFunction]

Table 7: Example prompt for Baseline KC conversion

System Message:
You are an experienced computer science teacher and education expert. You are given a Java programming problem along with n sample solutions. Your task is to identify generalizable knowledge components (skills or concepts) necessary to solve such problems.

A knowledge component (KC) is a single, reusable unit of programming understanding, such as a language construct, pattern, or skill, that contributes to solving a programming problem and can be learned or mastered independently.

Please follow these steps:
1. Analyze each solution carefully, noting critical constructs.
2. Reflect step by step on how each solution maps to distinct programming KCs that are independent and reusable.
3. For each KC, generate a concise name and provide a one-sentence reasoning explaining why this KC is necessary based on the provided solutions. Use the provided examples as reference for the appropriate level of detail. Make sure KCs are generalizable and applicable to a wide range of similar programming problems without referencing problem-specific details.
4. Ensure each KC is atomic and not bundled with others.

Your final response must strictly follow this JSON template:
{ "KC 1": "reasoning": "Reasoning for this KC (exactly 1 sentence)", "name": "Knowledge component name", "KC 2": "reasoning": "Reasoning for this KC (exactly 1 sentence)", "name": "Another specific knowledge component name", ... }

User prompt:
Example 1:
Problem: Write a function in Java that implements the following logic: Given a string `str` and a non-empty word, return a version of the original string where all chars have been replaced by pluses (+), except for appearances of the word which are preserved unchanged.
Expected Output: KC 1: If and else statement KC 2: While loop KC 3: Numerical comparisons KC 4: String formatting KC 5: String concatenation KC 6: String indexing KC 7: String length KC 8: String equality comparison

Now analyze the following problem using their solution code.
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 the empty string `""` if there are not two pieces of bread.
First sample solution is:
```java
public String getSandwich(String str){
  String bread = "bread";
  if (str.contains(bread) && str.length() >= 10){
    int first = str.indexOf(bread);
    int last = str.lastIndexOf(bread);
    String between = str.substring(first + 5, last);
    return between;
  }
  else{
    return "";
  }
}
```
Follow the instructions in system message. First, carefully examine the solutions and identify the important elements and patterns. Then, explicitly reason about what underlying knowledge components are required based on these solution codes. Finally, take the examples as reference and summarize your analysis clearly into generalizable and concise knowledge components.

Table 8: Example prompt for KC generation with one in-context example and one student solution

System Message:

You are an experienced computer science teacher and education expert. You are given a Java programming problem along with n AST representation of sample solutions. Your task is to identify generalizable knowledge components (skills or concepts) necessary to solve such problems.

A knowledge component (KC) is a single, reusable unit of programming understanding, such as a language construct, pattern, or skill, that contributes to solving a programming problem and can be learned or mastered independently.

The AST shows the hierarchical structure of the Java code, where each node represents a code construct (like class or method) along with its position in the source file.

Please follow these steps:

1. Analyze each AST carefully, noting critical constructs.
2. Reflect step by step on how each AST maps to distinct programming KCs that are independent and reusable.
3. For each KC, generate a concise name and provide a one-sentence reasoning explaining why this KC is necessary based on the provided solutions. Use the provided examples as reference for the appropriate level of detail. Make sure KCs are generalizable and applicable to a wide range of similar programming problems without referencing problem-specific details.
4. Ensure each KC is atomic and not bundled with others.

Your final response must strictly follow this JSON template:

{ "KC 1": "reasoning": "Reasoning for this KC (exactly 1 sentence)", "name": "Knowledge component name", "KC 2": "reasoning": "Reasoning for this KC (exactly 1 sentence)", "name": "Another specific knowledge component name", ... }

User prompt:

Example 1:

Problem: Write a function in Java that implements the following logic: Given a string `str` and a non-empty word, return a version of the original string where all chars have been replaced by pluses (+), except for appearances of the word which are preserved unchanged.

Expected Output: KC 1: If and else statement KC 2: While loop KC 3: Numerical comparisons KC 4: String formatting KC 5: String concatenation KC 6: String indexing KC 7: String length KC 8: String equality comparison

Now analyze the following problem using their AST.

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 the empty string `""` if there are not two pieces of bread.

First sample solution AST is:

```
JavaCodeAST [0, 0] - [11, 1]
  program [0, 0] - [10, 1]
    method_declaration [0, 0] - [10, 1]
      modifiers [0, 0] - [0, 6]
          identifier [0, 33] - [0, 36]
        if_statement [2, 4] - [9, 50]
          parenthesized_expression [2, 7] - [2, 27]
            binary_expression [2, 8] - [2, 26]
              method_invocation [2, 8] - [2, 20]
                argument_list [2, 18] - [2, 20]
          return_statement [3, 8] - [3, 18]
```

Follow the instructions in system message. First, carefully examine the ASTs and identify the important elements and patterns. Then, explicitly reason about what underlying knowledge components are required based on these ASTs. Finally, take the examples as reference and summarize your analysis clearly into generalizable and concise knowledge components.

Table 9: Example prompt for KC generation with one in-context example and one student solution AST. Only part of the AST shown due to spatial constraints.

System Message: You are an experienced computer science teacher and education expert. You will be provided with a list of knowledge components (KCs) that may vary in wording but sometimes refer to the same or related underlying concepts or skills. The KCs will be given in the format: ["KC 1 name", "KC 2 name", ..., "KC k name"] Your task is to: 1. Carefully examine all the KCs in the list to ensure none are overlooked. 2. Reason explicitly whether the KCs collectively refer to the same underlying concept or skill, or if they are related but represent distinct or complementary aspects of a broader theme. 3. Based on your reasoning: - If the KCs refer to the same concept or skill, select one KC from the list that best represents the group — choose the one that is most clearly worded, generalizable, and inclusive of the others. - If the KCs are related but too distinct to be represented by a single KC, create a concise and meaningful summary name that captures the broader theme or category shared by the KCs. Return your output strictly in the following JSON format: { "reasoning": "...", // Exactly one sentence explaining your reasoning "representative kc": "..." , // Selected KC if applicable, otherwise null "summary name": "..." , // Summary name if representative KC not chosen, otherwise null } **User prompt:** The knowledge components list is: [for loop iteration, while loop, array iteration] Now follow the instructions in system message. First, examine the list carefully to understand their shared meaning. Second, explicitly reason about the fundamental skill or concept underlying these knowledge components. Third, based on the reasoning, either select one KC that best represents the group if they share the same concept or summarize your analysis into one clear and concise phrase that accurately captures the essence of this cluster.

Table 10: Example prompt for Cluster summarization

System Message:
You are an experienced computer science teacher and education expert. You are given a Java programming problem, an incorrect student submission, and a predefined list of general programming knowledge components (KCs) relevant to solving the problem.

The predefined KCs relevant to solving the problem will be given in a list using the format ["KC 1 name", "KC 2 name", ..., "KC k name"]

Your task is to:
1. Identify all key errors in the student's code, and describe each error in exactly one sentence.
2. Fix the code by correcting the identified errors and return a correct version of the code.
3. Assess the student's mastery of each provided KC in the list based on the incorrect submission.
- Reflect on the student's original incorrect code and your corrected code output fixing the identified errors.
- For each KC, return a binary label which equals 1 if the student makes an error on this KC, and equals 0 if not.
- Your label should be based solely on the student's incorrect code submission.

Your final response must strictly follow this JSON template:
{ "error reasoning": ["First error described in one sentence.", "Second error described in one sentence.", ...], "fixed code": "The corrected Java implementation as a single string, properly formatted.", "KC error": "KC 1 name": 0/1, "KC 2 name": 0/1, ... }

User prompt:
Now analyze the following programming problem and the student's incorrect code submission to identify all errors and evaluate which knowledge components the student has made an error

Problem:
Write a function in Java that implements the following logic: Given 2 ints, a and b, return their sum. However, sums in the range 10..19 inclusive, are forbidden, so in that case just return 20.

Incorrect submission:
```
public int sortaSum(int a, int b){
  if (a + b <= 10 && a + b >= 20)
    return 20;
  else
    return a + b;
}
```

The knowledge components relevant to this problem are: [Basic arithmetic operations, Logical operators, If and else if statement, Numerical comparisons]

Follow the instructions in system message. First, carefully examine the problem and the incorrect code to understand what the intended correct behavior should be. Second, identify and describe each error in the code in exactly one sentence. Third, fix the errors and provide the corrected Java implementation. Finally, based on the incorrect submission, assess each knowledge component in the provided list by assigning a binary label: 1 if the student has made an error, or 0 if not.

Table 11: Example prompt for KC correctness labeling