SPHERE: Supporting Personalized Feedback at Scale in Programming Classrooms with Structured Review of Generative AI Outputs

Xiaohang Tang
Virginia Tech
Blacksburg, Virginia, USA
xiaohangtang@vt.edu

Sam Wong
University of Washington
Seattle, Washington, USA
samw627@uw.edu

Marcus Huynh
Virginia Tech
Blacksburg, Virginia, USA
mjhuynh@vt.edu

Zicheng He
University of Virginia
Charlottesville, Virginia, USA
bgc4bx@virginia.edu

Yalong Yang
Georgia Institute of Technology
Atlanta, Georgia, USA
yalong.yang@gatech.edu

Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu

Abstract

This paper introduces SPHERE, a system that enables instructors to effectively create and review personalized feedback for in-class coding activities. Comprehensive personalized feedback is crucial for programming learning. However, providing such feedback in large programming classrooms poses significant challenges for instructors. While Large Language Models (LLMs) offer potential assistance, how to efficiently ensure the quality of LLM-generated feedback remains an open question. SPHERE guides instructors' attention to critical students' issues, empowers them with guided control over LLM-generated feedback, and provides visual scaffolding to facilitate verification of feedback quality. Our between-subject study with 20 participants demonstrates SPHERE's effectiveness in creating more high-quality feedback while not increasing the time spent on the overall review process compared to a baseline system. This work contributes a synergistic approach to scaling personalized feedback in programming education, addressing the challenges of real-time response, issue prioritization, and large-scale personalization.

CCS Concepts

• **Human-centered computing → Interactive systems and tools**; *Natural language interfaces*; • **Social and professional topics** → *Computing education*; • **Applied computing** → *Education.*

Keywords

Generative AI, Large Language Model, Programming Education at Scale, Feedback, Computing Education

ACM Reference Format:

Xiaohang Tang, Sam Wong, Marcus Huynh, Zicheng He, Yalong Yang, and Yan Chen. 2025. SPHERE: Supporting Personalized Feedback at Scale in Programming Classrooms with Structured Review of Generative AI Outputs. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3706599.3720203

CHI EA '25, Yokohama, Japan

ACM ISBN 979-8-4007-1395-8/25/04
https://doi.org/10.1145/3706599.3720203

1 Introduction

When learning to program, personalized feedback is among the most effective ways to help students bridge the gap between their current understanding and performance and their desired learning goals [4, 13, 14]. To create effective personalized feedback, instructors need to assess students' coding challenges in the context of class learning objectives, consider each student's competencies and help resources, and tailor guidance to both the specific problem and the student's cognitive state. However, in programming classrooms, it is challenging to provide effective personalized feedback due to the sheer volume of time-sensitive issues arising from ongoing class exercises, such as coding and small group discussions, far exceeding instructors' attention capacity.

Past research and our own formative work with programming instructors have shown a large gap between the effort they put into providing feedback to students during class exercises using existing methods and the increasing number of questions that students accumulate during the exercises. Whether in-person (e.g., raising hands) or technology-based (e.g., learning dashboards like VizProg [32] and VizGroup [26]), these methods fall short of addressing this challenge. In fact, we argue that in-person approaches often result in less-personalized feedback due to time constraints and social context during the exercises, while learning dashboards can lead to information overload in large classes. This overload, combined with human and UI biases, often leads instructors to focus on issues that are more familiar or quickly addressable, rather than those that may be most critical. These limitations highlight three critical support needs for instructors: prioritizing critical issues, providing real-time feedback on these issues, and ensuring personalization of feedback at scale.

In this paper, we present SPHERE[1], an interactive system designed to help programming instructors create personalized feedback at scale for class exercises, including writing code and discussing code in groups. SPHERE combines intelligent issue detection with structured feedback generation and review. Specifically,

[1]SPHERE is an acronym for **S**caling **P**ersonalized **HE**lp in **RE**al-time

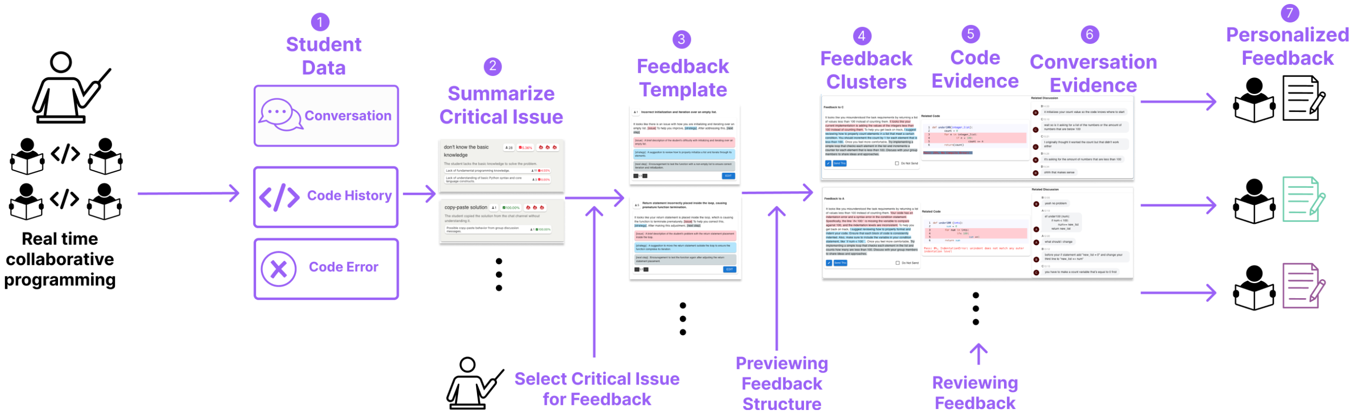


Figure 1: SPHERE's Workflow Overview. Once students' conversation logs, code history, and code errors come in (1), SPHERE continuously identifies critical issues and recommends them to the instructors (2). Instructors select the critical issues for feedback, which are then summarized and categorized to create Feedback Templates (3). These templates are previewed by instructors and further clustered (4) with relevant Code Evidence (5) and Conversation Evidence (6) after being filled out to provide context and support a rapid verification process. This results in personalized feedback being sent to each student (7).

SPHERE first employs a novel LLM architecture that continuously identifies key patterns in student coding and group activities. Then, SPHERE uses our "strategy-detail-verify" approach to ensure the quality of feedback created: 1) it allows instructors to *strategically* guide the LLM to generate high-quality feedback corresponding to these issues using key feedback components, and 2) it also provides information visualization bindings to facilitate instructors' rapid *verification* of the personalization and accuracy of feedback (*details*) in relation to corresponding issues. To evaluate the effectiveness of SPHERE, we conducted an in-lab, between-subject study with 20 instructors. The results underscore SPHERE's capacity to both streamline the feedback process and deepen instructors' engagement with feedback creation, which in turn can enable more impactful and aligned student guidance.

2 Related Work

Providing timely and effective feedback is critical for enhancing student learning [6, 14]. Recent AI-based approaches facilitate feedback creation for code understanding and auto generation [9, 19], yet studies show these methods can be only about 50% accurate in assessing the correct mistakes [8], and students frequently find the feedback confusing or in need of manual validation [21]. This underscores the necessity for scalable yet reliable feedback solutions.

In synchronous classrooms, analytics tools such as EduSense [1], AffectiveSpotlight [18], Glancee [16], Lumilo [15], VizProg [32], and Codeopticon [12] provide real-time insights into student engagement and performance, informing instructors of class-wide and individual needs. Systems like VizGroup [26], Pair-Up [31], Groupdynamics [23], and ClassInsights [20] further analyze collaborative behaviors, yet many instructors still struggle to determine how best to intervene based on these data. Human instructors remain irreplaceable for nuanced, real-time decisions in classroom settings [5, 7], but this requires significant effort and flexibility [27]. Therefore, there is a pressing need for strategies and tools that enable instructors to efficiently identify and address critical student issues at scale. Our work addresses this gap by providing an approach that supports educators in delivering prompt, personalized feedback, enhancing student engagement and learning while preserving the essential role of human instructors.

3 Design Goals

We conducted a targeted formative study to contextualize these challenges within the specific setting of in-class collaborative programming exercises at the university level. This study aimed to bridge the gap between general educational research and the unique demands of real-time, collaborative coding environments. We engaged four experienced instructors (22 years of teaching experience on average, $\sigma = 9.93$) of introductory programming courses at universities in in-depth interviews lasting 30 to 60 minutes each.

Based on our formative study and the identified gaps in existing literature, we established the following design goals to guide the development of SPHERE:

- **DG1: Effectively Guide Instructor Attention to Critical Issues in Real-Time.** To address the challenge of information overload in large-scale programming sessions, the system should help direct instructors' attention to the most significant issues and patterns emerging from students' collaborative activities with minimal effort.
- **DG2: Empower Instructors with Guided Control over LLM-Generated Feedback.** To bridge the gap between LLM capabilities and effective personalized feedback, the system should offer intuitive mechanisms for instructors to guide and control the feedback generation process.
- **DG3: Reduce the Time and Effort Required to Ensure the Quality of LLM-generated Feedback.** To mitigate risks associated with AI-generated content, such as hallucinations or misalignment with learning objectives, the system must facilitate quick identification and rectification of issues in AI-generated feedback.

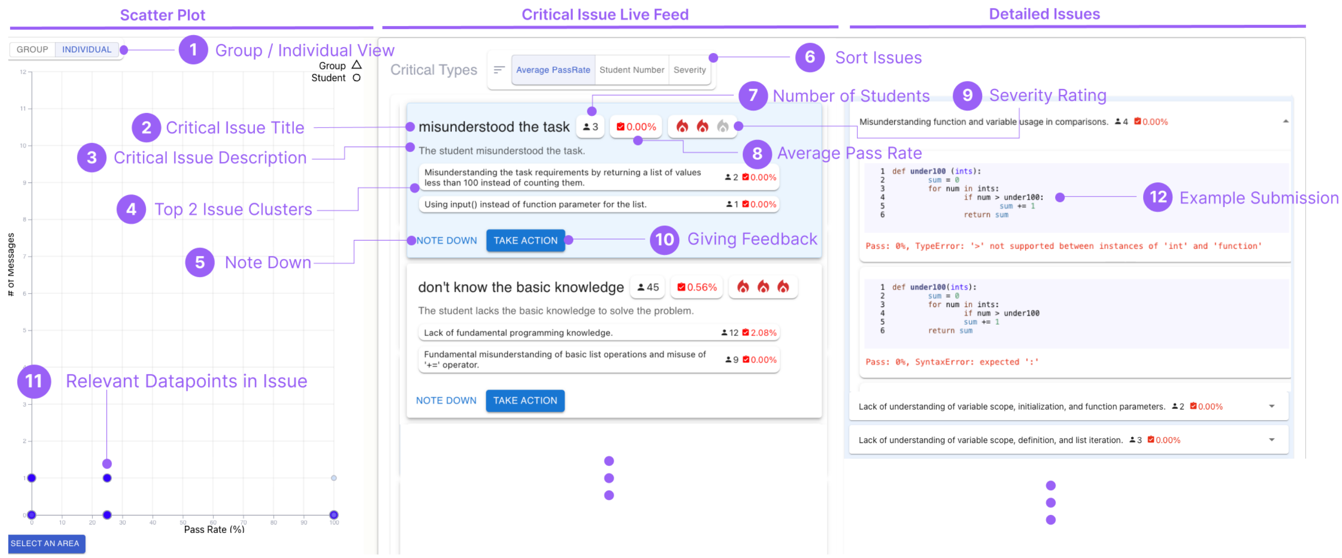


Figure 2: Critical Issue Recommendation Panel consists of Scatterplot, Critical Issue View and Detailed Issue. (1) Scatterplot changes based on group view. Critical Issue component consists of (2) Name of the Critical Issue, (3) Description of the Critical Issue, (4) Top 2 sub issue, (7) Number of Students with this critical issue, (8) Average passrate of students with this critical issue. Instructors can (5) note down and save the critical issue or (9) give feedback to the critical issue. Instructors can sort critical issue based on pass rate, number of students or severity. (10)Selecting a critical issue would display example submission and (11)highlight relevant datapoints.

4 SPHERE

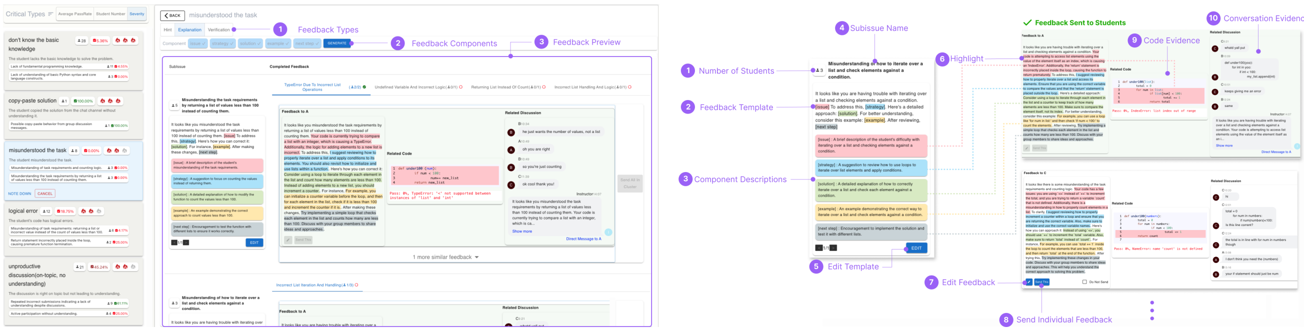
Figure 1 illustrates SPHERE's system workflow. SPHERE's UI consists of two main components: a Critical Issue Recommendation Panel (Figure 2), and a Structured Feedback Creation and Review Panel (Figure 3a).

4.1 Critical Issue Recommendation Panel

To effectively guide instructor attention to critical issues (DG1), SPHERE employs a custom LLM framework (Section 4.2) that continuously identifies patterns in students' coding activities and group interactions, classifies those that are critical, and then displays them on a teacher dashboard (Fig. 2). Teacher dashboards are visual displays that show student learning activities and progress. There are extensive studies on how to design an effective teacher dashboard and how they can help teachers inform their interventions and decision making [11, 28, 29]. Building upon prior work [2, 26, 32], SPHERE's dashboard contains a high-level scatter plot showing students'/groups' pass rates and the number of messages they have sent (left), a list of curated critical issues detected dynamically by the system (middle), and detailed information for each critical issue upon user click (right).

4.2 Critical Issue Recommendation Model

Prior work has shown LLMs' potential to identify and classify students' challenges [24]. To ensure that the LLM can effectively identify students with critical issues, we annotated two previously recorded programming exercise sessions collected from live collaborative programming sessions conducted at a local university, resulting in 502 data points in total. Two members of our research team annotated the recorded student activity data regarding (1) *whether an issue is critical or not* and (2) *types of issues* using an annotation tool (details in the Appendix) that allows annotators to watch a playback of and label different segments of student activity. The two annotators first conducted rounds of initial coding independently (including pilot coding sessions), and met multiple times to discuss their annotations for clarifications and to resolve any discrepancies. We then performed inter-rater reliability test for the 103 initially coded student activity data points which cover 20% in overlapped coding. Out of the 103 initially coded data, we report a Cohen-Kappa score for determining whether an issue is critical or not of 0.85. Across all labels (detailed in the Appendix, we report a mean Cohen-Kappa score of 0.78 ($S.E.$ = 0.08). Based on the minimum Cohen-Kappa score across all label (0.53 for Syntax Error label), the annotators reached a moderate agreement and thus the annotators proceeded with independent coding for the remaining 400 labels. After the labels were finalized, we randomly split the dataset into training and test sets, containing 80% and 20% of the original dataset, respectively. The training data are used for selecting few-shot examples and adjusting parameters. We used GPT-4o in the data training process, and state-of-the-art prompting engineering techniques, such as few-shot prompts [3] and AI-chains [30] were adopted. After evaluating the model with test data, we report an accuracy score of 0.71 for the LLM identifying issues that are critical and an $F1_{weighted}$ score of 0.57.



(a) (b)

Figure 3: *Left*: User Interface for creating and reviewing LLM-generated feedback. (1) Instructor can select different feedback types, which will select a subset of (2) feedback components. (3) Instructors can then generate feedback for preview. *Right*: Feedbacks are generated for each sub issue, which includes the name of the sub issue (4), the number of students in the sub issue (1), the feedback template with components as placeholders (2) and component description (3). Instructors are able to edit the template (5). Feedback is then generated in clusters that include code evidence (9) and conversation evidence (10). Feedback for each component is highlighted with their corresponding colors (6). Instructors can choose to edit or send individual feedback (7, 8).

4.3 Personalized Feedback Creation & Review Panel

When selecting an action, instructors can provide feedback on the currently selected issue. To facilitate personalized feedback generation (DG2), we developed a 'strategy-detail-verify' workflow. First, instructors can strategize by choosing the most appropriate feedback mode from three options: Hints, Explanation, or Verification (Figure 3a.1). Upon pressing the generate button, a template for the specific feedback type is created, personalized for each sub issue, with placeholders inserted at designated positions. Next, feedback is generated in smaller clusters for each subissue, allowing instructors to see the details of the feedback intended for each student. Related code examples and conversations are displayed alongside the feedback. Finally, instructors can verify the validity of the feedback before deciding whether to send it to students. This step ensures the appropriateness and accuracy of the feedback.

4.3.1 *Component Based Feedback Generation.*

Feedback is structured to consist of up to 5 components: Issue , Strategy , Solution , Example , Next step . These five components are derived from Hattie et al.'s feedback model [14], which states that effective feedback answers three questions: *(1) What are the goals? (2) What has been done to progress towards the goals? and (3) What activities need to be undertaken to make better progress?* Furthermore, each piece of feedback is directed at three levels respectively: Task Level, Process Level, and Self-Regulation Level according to the feedback model [14]. Instructors often give feedback to guide students who are stuck, clarify students' confusion, or check students' understanding. Hence, depending on each student's situation, instructors can provide different types of feedback (Hints, Explanations, or Verifications). Providing such feedback presets allows instructors to quickly generate feedback that fits their teaching strategy, thereby reducing the time required to draft feedback (DG3).

Step 1: Feedback Template Generation: SPHERE takes in the issue description, feedback type and chosen components, and generates a feedback template with selected content as placeholders (in [square brackets]), as well as feedback corresponding to each component (see Figure 3b.2).

Step 2: Detailed Feedback Composition SPHERE then takes the generated list of component feedback and applies it to the template, generating a complete feedback for students. It also takes in student information and elaborates on student's performance by referencing their coding and conversation history.

4.3.2 *Feedback Clusters with highlights on Evidence.*

Our design for displaying feedback clusters aligns with prior literature on designing UI elements for validating large-scale LLM outputs, which allows users to determine whether the AI-generated feedback meets their current needs [10]. Feedback is generated in clusters to enable instructors to easily review the generated content (DG3). Under each sub-issue within a critical issue (Figure 2.12), feedback is further clustered into smaller groups to ensure that instructors can easily review different feedback clusters. To facilitate easy verification of feedback (DG3), we incorporated highlighting as a way for instructors to review how each component is incorporated into the final feedback, as well as the related code or conversation that reflects the feedback.

5 Lab Study

We conducted an in-person between-subject user study to examine SPHERE's usability and effectiveness for identifying students' issues and sending feedback. The study is approved by the IRB at our institute. We recruited 20 participants (7 female, 11 male, 2 non-binary) who have both teaching and programming experiences at a four-year university via personal networks, local mailing lists, and snowball sampling. Each participant was compensated with $15 USD for their time and effort.

Condition	Generated			Sent			Edited**			
	Incorrect	Shallow	High	Incorrect	Shallow	High	Low-low	Low-high	High-low	High-high
Baseline	44.17%	10.00%	45.83%	45.00%	8.67%	46.33%	45.00%	20.00%	5.00%	30.00%
SPHERE	14.16%*	5.00%	80.83%**	9.17%**	10.67%	80.17%**	0.00%	22.22%	0.00%	77.78%

Table 1: Feedback Quality in the Sampled Dataset. Edited feedback messages are grouped by the change of qualities before and after the edit, where low-quality feedback includes both *incorrect feedback* and *shallow feedback*. Data of edited changes are tested by the *Fisher Exact Probability Test* ($p = 0.001$). $*$ indicates $p < 0.05$, while $**$ indicates $p < 0.01$.

Condition	Mental demand	Physical demand	Temporal demand	Performance	Effort	Frustration
Baseline	5.0 (4.60 ± 1.26)	2.0 (2.10 ± 1.20)	5.0 (4.60 ± 1.17)	4.0 (3.90 ± 1.37)	4.0 (4.00 ± 1.25)	2.5 (2.70 ± 1.57)
SPHERE	5.5 (4.80 ± 1.93)	2.0 (2.30 ± 1.49)	5.0 (4.60 ± 1.78)	3.0 (3.10 ± 1.10)	4.0 (4.20 ± 1.48)	2.5 (3.00 ± 1.76)

Table 2: Response to NASA TLX items. Format: median (mean ± standard deviation)

5.1 Protocol

5.1.1 Live Simulation. To simulate the real-time, in-class teaching experience of a large lecture, we asked participants to interact with a live playback of an in-class exercise session, inspecting and giving feedback on student behaviors. To ensure the authenticity of the data participants interacted with, we used a research tool [25] to capture real data from a large introductory-level university programming course's collaborative exercise session that involved 111 students. During the exercise session, students first worked on a Python programming exercise, and then they were divided into groups to discuss their issues and help each other on their laptops. The Python problem was to write a function to count the number of elements less than 100 in a given list.

5.1.2 Conditions. Each participant used the system under one of the following conditions:

- **Baseline:** a baseline version of SPHERE without any feedback type, feedback component, and feedback template for generation, and participants control the feedback generation by editing the prompt to instruct the LLM, which is one of the most common ways to use LLM. A default prompt was given to ask AI to generate feedback based on the student's recent code submissions and discussions. There is no visual augmentation for feedback review either, but the baseline version still had the complete Critical Issue Recommendation Panel.
- **SPHERE:** a full version of SPHERE with all its features.

5.1.3 Tasks. Each participant completed three tasks in the study:

- **T1 and T2:** Task 1 (T1) and Task 2 (T2) asked participants to help as many struggling students as they could by checking the critical issue recommendation panel and sending feedback. T1 and T2 were limited to 6 minutes each.
- **T3:** Task 3 (T3) asked participants to help students who were actively discussing but with low performance by selecting the plot and sending feedback on a certain time stamp.

T1 covers the individual programming stage of the exercise session, while T2 and T3 focus on the collaborative programming stage. For all three tasks, we emphasized the importance of the feedback quality to participants and had them understand the stake level of the tasks by asking them to imagine this is the live class setting. T1 and T2 were designed to mimic the real-world circumstances that instructors need to address students' issues while facing the dynamically changed issue set and limited time. Similarly, T3 was designed to simulate a more focused issue-resolving process while still needing to provide timely support to students.

5.1.4 Study Procedure. Each study was conducted in person in a lab setting and lasted around 45 minutes. At the beginning of each study session, we collected informed consent from the participants after introducing the goal and the process of the study. After that, we gave participants an explanation of the context of the data and tasks used in the study. Following the general introduction and explanation, we then offered a detailed tutorial of the system and a warm up session.

Once they were familiar with the system, participants were asked to complete the three designed tasks. At the end of the study, participants completed a survey with Likert scale questions and participated in a semi-structured interview. All studies were screen- and audio-recorded and participants were asked to think aloud while completing the survey questions. The order of conditions and tasks were counter-balanced and randomized.

5.2 Results

By recording the feedback participants generated, edited, and sent in the user study, we collected 5871 feedback messages in total (5049 generated, 107 edited, 715 sent). To evaluate the quality of the feedback, two researchers used a custom annotation tool to annotate the feedback data (details in the Appendix). The annotation process was as follows: First, we sampled feedback data from each participant and each task proportionally (1% of generated feedback, 20% of edited feedback, and 10% of sent feedback) and randomly shuffled the sampled data. Then, each sampled data point was coded into one of the following three categories: *incorrect feedback*, *shallow feedback*, and *high-quality feedback*. A description of the coding standards is provided in the Appendix. After initializing the coding scheme, the two researchers first conducted a pilot coding meeting and discussion to clarify and resolve conflicts on the coding standards. Then, they began independently coding the same sampled dataset. For the independent annotations, the agreement on *incorrect feedback*, *shallow feedback*, and *high-quality feedback* was 85.96%, 53.33%, and 96.15%, respectively. To achieve a high level of consensus [17], the two researchers then negotiated and resolved each disagreement through discussion until 100% agreement was reached on the coded sample dataset (Table1).

5.2.1 SPHERE helped participants create better feedback. Participants using SPHERE sent significantly more *high-quality feedback* (SPHERE: $\mu = 80.17\%, \sigma = 0.17$; baseline: $\mu = 46.33\%, \sigma = 0.28$; $p < 0.01$), while they also sent less *incorrect feedback* (SPHERE: $\mu = 9.1717\%, \sigma = 0.15$; baseline: $\mu = 45.00\%, \sigma = 0.31$; $p < 0.01$) in the sampled sent dataset. We did not find evidence of a significant difference in the number of *shallow feedback* between two conditions (SPHERE: $\mu = 10.67\%, \sigma = 0.17$; baseline: $\mu = 8.67\%, \sigma = 0.14$).

5.2.2 SPHERE helped participants control feedback generation. Based on the sampled generated dataset, under SPHERE condition, participants generated significantly more (SPHERE: $\mu = 80.83\%, \sigma = 0.22$; baseline: $\mu = 45.83\%, \sigma = 0.28$; $p < 0.01$) *high-quality feedback* than the baseline while generated less (SPHERE: $\mu = 14.16\%, \sigma = 0.15$; baseline: $\mu = 44.17\%, \sigma = 0.32$; $p < 0.05$) *incorrect feedback*. On average, there are 5.00% ($\sigma = 0.11$) *shallow feedback* in feedback generated by SPHERE and 10.00% ($\sigma = 0.16$) *shallow feedback* in feedback generated by the baseline. During the study, participants using SPHERE attempted 2.40 ($\sigma = 2.07$) times modifying feedback type and component to generate new feedback, while participants using the baseline attempted 1.8 ($\sigma = 2.30$) times editing the prompts to control the generation results. The survey results show that participants reported high-level perceived controls of the feedback generation process for both SPHERE ($Median = 6.00, \sigma = 1.71$) and the baseline system ($Median = 5.50, \sigma = 1.76$), as well as the perceived ease of control level (SPHERE: $Median = 5.50, \sigma = 1.17$; Baseline: $Median = 6.00, \sigma = 1.17$).

Though we could not obtain a significant difference in the number of attempts to control the generation process, participants' behaviors reflect the difference in their intentions. Under the baseline condition, participants' intentions of controlling generation are verbally expressed in the modified prompt, e.g., *"Try and shorten the response", "Don't give away answers"* (P8). The goals are limited to low-level requirements about the length of the generation and not to include the direct solution in the result. Whereas, using SPHERE, participants tended to think about the type of help they would like to offer to students, which is unseen in the baseline condition. For instance, P9 included example generation in the feedback for students who "lacked basic knowledge" in order to help them get started with the problem. P5 suggests that they would consider excluding some components in the feedback depending on the severity of the issue. Moreover, participants also commented on the feedback SPHERE generated as like *"from a senior instructor"*(P7).

5.2.3 SPHERE helped participants improve feedback quality. During the user study, participants made 6.20 ($\sigma = 8.75$) edits using SPHERE, and participants using the baseline edited 4.50 ($\sigma = 3.10$) messages on average. To compare the quality of feedback messages before and after the edits, we take both *incorrect feedback* and *shallow feedback* as *low-quality feedback*. After that, there would be four types of quality change: low-to-low, low-to-high, high-to-low, and high-to-high. While the number of edits is similar for the two conditions, in the sampled edited dataset, there are 22.22% low-to-high and 77.78% high-to-high changes under SPHERE condition, while there are 45.00% low-to-low, 20.00% low-to-high and 5.00% high-to-low, and 30.00% high-to-high changes under the baseline condition. Moreover, using the Fisher Exact Probability Test [22], we found that participants were significantly more likely to convert lower-quality feedback (i.e., *incorrect feedback* and *shallow feedback*) into higher-quality feedback ($p = 0.001$) using SPHERE.

6 Limitation and Future Work

While our evaluation was done in a lab setting instead of in a real classroom – and thus, a reduced set of distractions may impact the available attention that the instructor has to validate and generate feedback – we believe it clearly shows that the overall net effect of SPHERE is positive. Our focus was on the impact of SPHERE on instructors' ability to provide feedback that they felt was effective, but we did not explore the impact on students directly. Future work will explore how different feedback variations affect both perceived and real student outcomes over time, and how this feedback can be fed back to instructors to give them deeper insights and enable more comprehensive personalization.

7 Conclusion

In this paper, we present SPHERE, an interactive system designed to help programming instructors create personalized feedback at scale for class exercises. Our study shows that SPHERE helped instructors create more high-quality feedback, and enable them to transform initially low-quality feedback into higher-quality versions more frequently. These results offer insights into AI-assisted educational tools, demonstrating how to scale personalized feedback in programming courses while addressing real-time response, prioritizing issues, and engaging instructors more deeply.

References

[1] Karan Ahuja, Dohyun Kim, Franceska Xhakaj, Virag Varga, Anne Xie, Stanley Zhang, Jay Eric Townsend, Chris Harrison, Amy Ogan, and Yuvraj Agarwal. 2019. EduSense: Practical classroom sensing at Scale. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 3 (2019), 1–26.

[2] Vincent Aleven, Jori Blankestijn, LuEttaMae Lawrence, Tomohiro Nagashima, and Niels Taatgen. 2022. A dashboard to support teachers during students' self-paced AI-supported problem-solving practice. In *European Conference on Technology Enhanced Learning*. Springer, 16–30.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[4] Deborah L Butler and Philip H Winne. 1995. Feedback and self-regulated learning: A theoretical synthesis. *Review of educational research* 65, 3 (1995), 245–281.

[5] Çiğdem Suzan Çardak and Kıymet Selvi. 2016. Increasing teacher candidates' ways of interaction and levels of learning through action research in a blended course. *Computers in Human Behavior* 61 (2016), 488–506.

[6] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Wang, Sang Won Lee, Walter S Lasecki, and Steve Oney. 2020. EdCode: Towards Personalized Support at Scale for Remote Assistance in CS Education. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[7] Anna Comas-Quinn. 2011. Learning to teach online or learning to become an online teacher: An exploration of teachers' experiences in a blended learning course. *ReCALL* 23, 3 (2011), 218–232.

[8] Iria Estévez-Ayres, Patricia Callejo, Miguel Ángel Hombrados-Herrera, Carlos Alario-Hoyos, and Carlos Delgado Kloos. 2024. Evaluation of LLM Tools for Feedback Generation in a Course on Concurrent Programming. *International Journal of Artificial Intelligence in Education* (2024), 1–17.

[9] Hagit Gabbay and Anat Cohen. 2024. Combining LLM-Generated and Test-Based Feedback in a MOOC for Programming. In *Proceedings of the Eleventh ACM Conference on Learning@ Scale*. 177–187.

[10] Elena L. Glassman, Ziwei Gu, and Jonathan K. Kummerfeld. 2024. AI-Resilient Interfaces. arXiv:2405.08447 [cs.HC] https://arxiv.org/abs/2405.08447

[11] Wolfgang Greller and Hendrik Drachsler. 2012. Translating learning into numbers: A generic framework for learning analytics. *Journal of Educational Technology & Society* 15, 3 (2012), 42–57.

[12] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. ACM, Association for Computing Machinery, New York, NY, USA, 599–608. https://doi.org/10.1145/2807442.2807469

[13] John Hattie. 2012. *Visible learning for teachers: Maximizing impact on learning.* Routledge.

[14] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[15] Kenneth Holstein, Gena Hong, Mera Tegene, Bruce M McLaren, and Vincent Aleven. 2018. The classroom as a dashboard: Co-designing wearable cognitive augmentation for K-12 teachers. In *Proceedings of the 8th international conference on learning Analytics and knowledge*. 79–88.

[16] Shuai Ma, Taichang Zhou, Fei Nie, and Xiaojuan Ma. 2022. Glancee: An adaptable system for instructors to grasp student learning status in synchronous online classes. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–25.

[17] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (nov 2019), 23 pages. https://doi.org/10.1145/3359174

[18] Prasanth Murali, Javier Hernandez, Daniel McDuff, Kael Rowan, Jina Suh, and Mary Czerwinski. 2021. Affectivespotlight: Facilitating the communication of affective responses from audience members during online presentations. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–13.

[19] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[20] Tricia J. Ngoon, S Sushil, Angela E.B. Stewart, Ung-Sang Lee, Saranya Venkatraman, Neil Thawani, Prasenjit Mitra, Sherice Clarke, John Zimmerman, and Amy Ogan. 2024. ClassInSight: Designing Conversation Support Tools to Visualize Classroom Discussion for Personalized Teacher Professional Development. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 663, 15 pages. https://doi.org/10.1145/3613904.3642487

[21] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis) read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–26.

[22] Michel Raymond and François Rousset. 1995. An exact test for population differentiation. *Evolution* (1995), 1280–1283.

[23] Arissa J Sato, Zefan Sramek, and Koji Yatani. 2023. Groupnamics: Designing an interface for overviewing and managing parallel group discussions in an online classroom. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–18.

[24] Wannapon Suraworachet, Jennifer Seon, and Mutlu Cukurova. 2024. Predicting challenge moments from students' discourse: A comparison of GPT-4 to two traditional natural language processing approaches. In *Proceedings of the 14th Learning Analytics and Knowledge Conference* (Kyoto, Japan) *(LAK '24)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3636555.3636905

[25] Xiaohang Tang, Xi Chen, Sam Wong, and Yan Chen. 2023. VizPI: A Real-Time Visualization Tool for Enhancing Peer Instruction in Large-Scale Programming Lectures. In *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23 Adjunct)*. Association for Computing Machinery, New York, NY, USA, Article 17, 3 pages. https://doi.org/10.1145/3586182.3616632

[26] Xiaohang Tang, Sam Wong, Kevin Pu, Xi Chen, Yalong Yang, and Yan Chen. 2024. VizGroup: An AI-assisted Event-driven System for Collaborative Programming Learning Analytics. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 93, 22 pages. https://doi.org/10.1145/3654777.3676347

[27] Janneke Van de Pol, Monique Volman, and Jos Beishuizen. 2010. Scaffolding in teacher–student interaction: A decade of research. *Educational psychology review* 22 (2010), 271–296.

[28] Anouschka van Leeuwen, Stephanie D Teasley, and Alyssa Friend Wise. 2022. Teacher and student facing learning analytics. *Handbook of learning analytics* (2022), 130–140.

[29] Alyssa Friend Wise and Yeonji Jung. 2019. Teaching with analytics: Towards a situated model of instructional decision-making. *Journal of Learning Analytics* 6, 2 (2019), 53–69.

[30] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582

[31] Kexin Bella Yang, Vanessa Echeverria, Zijing Lu, Hongyu Mao, Kenneth Holstein, Nikol Rummel, and Vincent Aleven. 2023. Pair-up: prototyping human-AI co-orchestration of dynamic transitions between individual and collaborative learning in the classroom. In *Proceedings of the 2023 CHI conference on human factors in computing systems*. 1–17.

[32] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. VizProg: Identifying Misunderstandings By Visualizing Students' Coding Progress. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

Issue	Definition
Copy-Paste Solution	This critical situation occurs when a student pastes a solution from another source.
Don't know the basic knowledge	In this situation, the current student lacks the necessary knowledge of the programming language and its concepts. Most commonly in the form of typing pseudo-syntax (syntax that they've made up to accomplish the task, or combining existing syntax in invalid ways).
Syntax error	Syntax errors that be evaluated based on factors such as submission frequency, coexistence with other issues, and error quantity.
Logical error	Flaws in the student's code that prevent them from passing all the test cases. Usually, these flaws stem from a misunderstanding of the problem or lack of basic knowledge.
Chat-bot solution	Look for moments where there are no solutions in the chat from any other students, and the current student will produce a perfect solution after visibly struggling for some time.
Off-topic	This critical situation occurs when there are ongoing discussions in the chat that are not related at all to the current task.
Unproductive discussion (related but not the right approaches)	This type of scenario occurs when students try to help each other, but it does not result in success or change in the student's solution.
Unproductive discussion (Right on topic but not leading to understanding)	Multiple attempts of students helping each other, but the student cannot grasp an understanding regardless of if the other student is explaining it perfectly or not. Or the student trying to help gives out the solution directly, without aiding the struggling student in understanding the concept.
Interpersonal Conflicts	When there is disagreement among the students in chat. Clashes between students.
Misunderstood the task	This critical situation happens when the student fails to grasp the requirements or objectives of the assignment.

Table 3: Rubric for annotating student coding and conversation data

A Rubric for annotating feedback

Incorrect Feedback (-1): This type of feedback leads to confusion, frustration, or lack of confidence in the student. It may also result in students applying incorrect practices. *Characteristics:*

- The feedback is factually incorrect, wrong, or misleading.
- Provides direct answers without explanation, preventing learning or understanding.
- Uses confusing or unclear language, making it hard for students to apply the feedback.
- Offers overly negative or harsh criticism without actionable steps for improvement.
- Contains errors that can misguide students or reinforce incorrect understanding of concepts.

Shallow Feedback (0): This type of feedback may leave the student feeling unsure of how to proceed or improve. While technically correct, it doesn't significantly help the student grow. *Characteristics:*

- The feedback is factually correct but lacks depth or clarity.
- Offers minimal insight or vague suggestions, not addressing specific issues with enough detail.
- Does not guide the student toward improvement or deeper understanding.
- Is clear but not particularly encouraging, leaving the student without a clear sense of next steps.
- Avoids technical errors, but lacks motivation or educational value.

High-quality Feedback (1): High-quality feedback helps students understand what they did right, where they went wrong, and how to improve. It fosters a growth mindset and promotes deeper learning, building both technical and problem-solving skills. *Characteristics:*

- The feedback is clear, specific, and actionable, helping the student understand both their strengths and areas for improvement.
- Encourages learning by explaining why something is incorrect and offering suggestions for how to fix it.
- Provides context for why a certain approach is better and explains the concepts behind it.
- Balances criticism with encouragement, helping build the student's confidence while addressing errors.
- Links feedback to learning objectives, reinforcing key programming concepts in a positive and constructive manner.

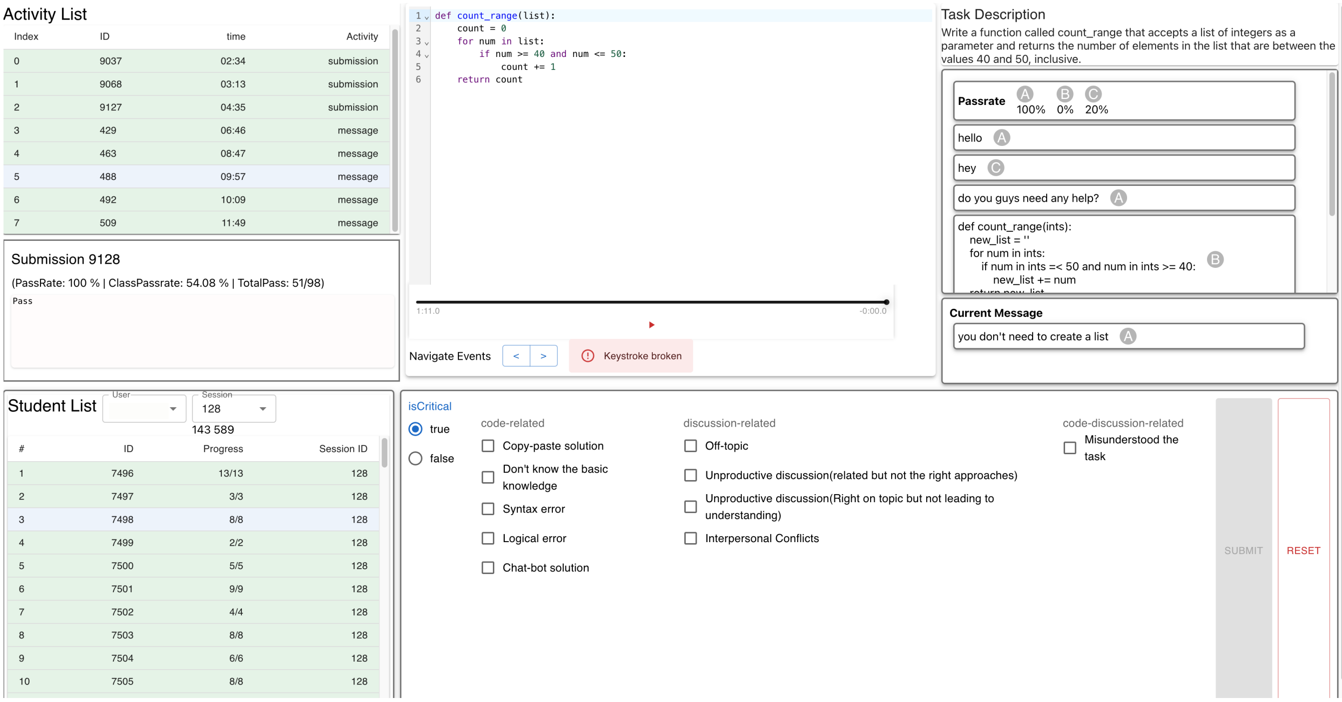


Figure 4: User Interface for annotator tool. Annotators select an activity from each student (Left) and provide a label by reviewing the playback for the selected activity.

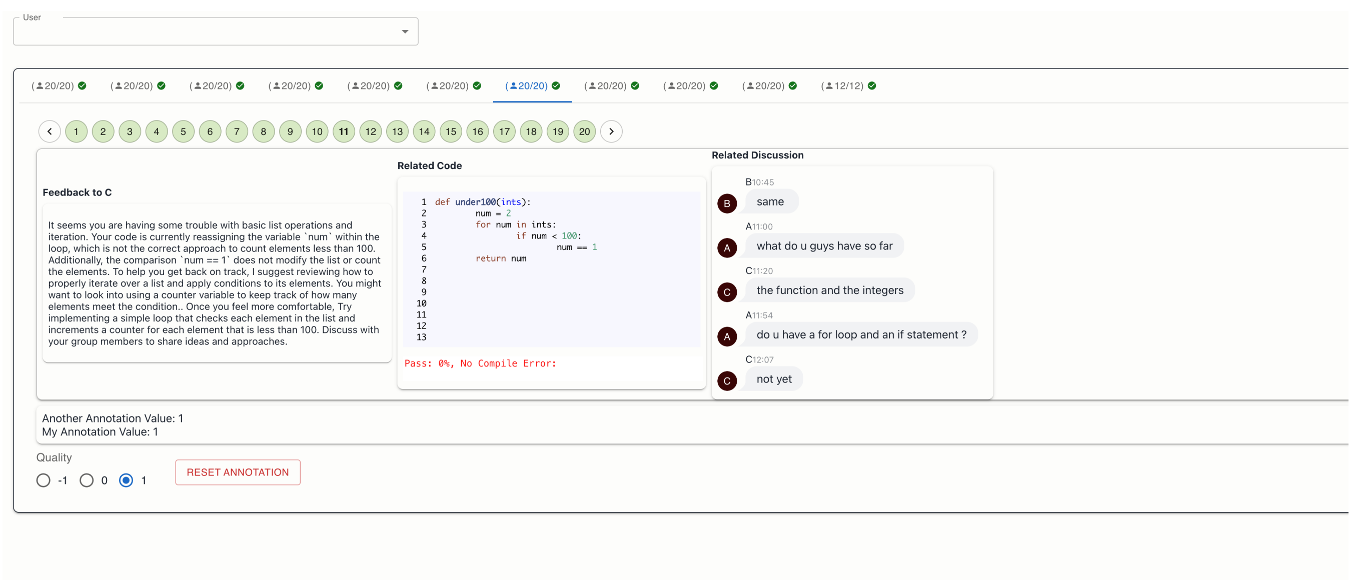


Figure 5: User Interface for annotating feedback. Annotators select a feedback message from the message list (Top) and provide a label by reviewing the content of the message and related codes and discussions.

You are a programming instructor tasked with identifying students who may be struggling due to technical or communication issues. Please first identify the learning objectives by checking the programming problem. You have access to the student's latest code submission, the student's previous code submissions, and the student's group discussion history. Using this context—which includes their code, discussion contributions, error messages, pass rate, and overall class performance—determine if the student is in a critical situation, which means that the student needs your attention and could benefit from your assistance.

The programming problem students are working on: Write a function called under100 that accepts a list of integers and returns the number of values in the list that are less than 100.

Input Format(JSON):

A collection containing the student's basic information, latest activity(code submission/message), previous code submissions, and group discussion history:

- student ID: [string] the student's ID
- latestActivity:[Object] containing information about the student's latest submission, including:
  - activityType:[string] type of activity
  - time: [numeric] time the activity was made (second)
  - activityContent:[string] code or message
  - errorType: [string] type of the error of the code submission if the activity type is submission
  - errorMessage: [string] error message of the code submission if the activity type is submission
  - passRate:[numeric] the student's current passrate
  - currentMessageCount:[numeric] number of messages from the student
  - groupPassRate:[numeric] the group's current passrate
- submissionHistory:[Array] a list of previous code submission records of the student, each containing:
  - time: [numeric] time the submission was made (second)
  - passRate: [numeric] the passrate of this submission
  - errorType: [string] type of the error of the code submission
  - errorMessage: [string] error message of the code submission
- messageHistory: [Array] a list of message records in the group, each containing:
  - time: [numeric] time the message was sent (second)
  - message: [string] the content of the message
  - sender_id: [string] the id of the student who sent the message
  - senderPassRate: [numeric] the pass rate of the sender right now

```
Input Example:
{
"id": "9027_submission",
"type": "submission",
"student_id": "7512",
"timestamp": "2024-04-23T13:35:22.133773+00:00",
"classAveragePassRate": 6.530612244897959,
"student_name": "B",
"latestSubmission": {
"type": "submission",
"time": "2024-04-23T13:35:22.133773+00:00",
"code": "def count_range(integers_list):\n    if 40 <= num <= 50:\n        num = count(integers_list)\n        return num\n    ",
"errorMessage": "File \"<exec>\", line 9, in count_range\nUnboundLocalError: cannot access local variable 'num' where it is not associated with a value",
"passRate": 0,
"currentMessageCount": 0,
"groupPassRate": 0
},
"submissionHistory": [],
"messageHistory": []
}
```

Output Format(valid JSON):

A collection containing the classification result and the reason:

- isCritical: [boolean] the classification result of whether the student is in a critical situation.
- reason: [string] explanation of the classification

```
Output Example:
{
"isCritical": true,
"reason": "The student has a pass rate of 0 for their latest submission, indicating they are unable to produce functional code. The error message indicates an
'UnboundLocalError', which suggests a misunderstanding of variable scope and initialization. Additionally, the student has no previous submission history and h
as not participated in any group discussions, as shown by the empty message history. This lack of engagement and consistent failure to submit working code indi
cates that the student is in a critical situation and requires immediate assistance."
}
```

Untitled 1

Figure 6: Prompt for identifying critical issue

Issue Prompt

You are a programming instructor tasked with identifying students' struggles based on their behaviors. You have access to the student's latest code submission, the student's previous code submissions, and the student's group discussion history. Using this context—which includes their code, discussion contributions, error messages, pass rate, and overall class performance—determine what kinds of struggles the students are facing right now. The struggle types and their rubrics are: [

"Don't know the basic knowledge": "Student lacks fundamental programming knowledge, often evident through pseudo-syntax or misuse of core language constructs (e.g., incorrectly using def or parameters).",

"Syntactic and Runtime Errors": "Common errors, but not always critical. Consider factors like the number of submissions, other errors (e.g., logical), and frequency of syntax issues before labeling as critical.",

"Logical error": "Occurs when a student's code has no other errors but fails to meet the task's requirements due to a misunderstanding or lack of basic knowledge.",

"Chat-bot solution": "Similar to copy-paste, but the student suddenly produces a perfect solution after struggling. AI-generated solutions often have flawless grammar and comments.",

"Misunderstood the task": "Student misunderstands the assignment's requirements, often evident in incorrect returns or variable initialization.",

"Copy-paste solution": "Occurs when a student pastes a solution from the chat channel. Signs include sudden drastic changes in their solution or pasting code into chat".
]

The programming problem students are working on: Write a function called under100 that accepts a list of integers and returns the number of values in the list that are less than 100.

Input Format(JSON):

A collection containing the student's basic information: the student's latest activity(code submission/message), the student's previous code submissions, and group discussion history:
-student ID: [string] the student's ID

- latestActivity:[Object] containing information about the student's latest submission, including:
  - activityType:[string] type of activity
  - time: [numeric] time the activity was made (second)
  - activityContent:[string] code or message
  - errorType: [string] type of the error of the code submission if the activity type is submission
  - errorMessage: [string] error message of the code submission if the activity type is submission
  - passRate:[numeric] the student's current passrate
  - currentMessageCount:[numeric] number of messages from the student
  - groupPassRate:[numeric] the group's current passrate
- submissionHistory:[Array] a list of previous code submission records of the student, each containing:
  - id: [string] id of the submission
  - time: [numeric] time the submission was made (second)
  - passRate: [numeric] the passrate of this submission
  - errorType: [string] type of the error of the code submission
  - errorMessage: [string] error message of the code submission
- messageHistory: [Array] a list of message records in the group, each containing:
  - id: [string] id of the message
  - time: [numeric] time the message was sent (second)
  - message: [string] the content of the message
  - sender_id: [string] the id of the student who sent the message
  - senderPassRate: [numeric] the pass rate of the sender right now

Output Format(valid JSON):

A collection containing the classification result and the reason:

- struggles: [Array] list of classified struggles of the students based on the student's behavior. Elements are Objects containing:
  - name: [string] name of the struggle
  - reason: [string] reason of the classification.

Issue Prompt 1

Figure 7: Prompt for identifying student's struggles 1

User Prompt


```
{
        "id": "05ollRMtoAhLVmpbKP3F",
        "type": "submission",
        "student_id": "iFJgbW65I3OeqKLQAHRy",
        "latestSubmission": {
            "type": "submission",
            "time": 413,
            "code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n
if number < 100:\n                        split.it.remove(number)\n                else:\n                        break\n        return split_it",
            "errorType": "No Compile Error",
            "errorMessage": "",
            "passRate": 0,
            "currentMessageCount": 0,
            "groupPassRate": 33.333333333333336
        },
        "submissionHistory": [
            {
                "id": "81IDaR5ep0QZEfDWSqs9",
                "time": 217,
                "passRate": 0,
                "errorType": "No Compile Error",
                "errorMessage": ""
            },
            {
                "id": "TEgPWjQeE3xo0hjpohIN",
                "time": 230,
                "passRate": 0,
                "errorType": "No Compile Error",
                "errorMessage": ""
            },
            {
                "id": "OwbWPSykxs7mfxWdVz5a",
                "time": 331,
                "passRate": 0,
                "errorType": "No Compile Error",
                "errorMessage": ""
            },
            {
                "id": "ojs1CHiw4fLLfMHruWgs",
                "time": 393,
                "passRate": 0,
                "errorType": "No Compile Error",
                "errorMessage": ""
            }
        ],
        "messageHistory": []
    }
```


Assistant Prompt


```
{
  "struggles": [
    {
      "name": "Don't know the basic knowledge",
      "reason": "The student is attempting to use input and string manipulation to solve a problem that requires basic knowledge of list operations and compari
sons. The approach taken does not align with the task requirements."
    },
    {
      "name": "Logical error",
      "reason": "The logic in the code is flawed; the student is trying to remove elements from a list while iterating over it, which can lead to unexpected be
havior. Additionally, the condition for checking if a number is less than 100 is incorrectly applied to strings instead of integers."
    },
    {
      "name": "Unproductive discussion(Right on topic but not leading to understanding)",
      "reason": "The student has not engaged in any group discussions, which may indicate a lack of seeking help or clarification on the task, leading to conti
nued misunderstanding."
    }
  ]
}
```


Untitled 1

Figure 8: Input and Output Example for identifying student's struggle 1

You are a programming instructor tasked with identifying students' struggles based on their behaviors. You have access to the student's latest code submission, the student's previous code submissions, and the student's group discussion history. Using this context—which includes their code, discussion contributions, error messages, pass rate, and overall class performance—determine what kinds of struggles the students are facing right now. The struggle types and their rubrics are: [
"Off-topic": "Chat discussions unrelated to the task at hand.",
"Unproductive Discussion (Wrong Approach)": "Students help, but the guidance provided is incorrect, leading to no improvement in the solution after multiple attempts.",
"Unproductive Discussion (On-Topic, No Understanding):": "Help is given, but the struggling student cannot grasp the explanation, or the helper gives the solution directly without fostering understanding.",
"Interpersonal Conflicts": "Disagreements or clashes between students in the chat.",
]

The programming problem students are working on: Write a function called under100 that accepts a list of integers and returns the number of values in the list that are less than 100.

Input Format(JSON):

A collection containing the student's basic information: the student's latest activity(code submission/message), the student's previous code submissions, and group discussion history:
-student ID: [string] the student's ID

- latestActivity:[Object] containing information about the student's latest submission, including:
  - activityType:[string] type of activity
  - time: [numeric] time the activity was made (second)
  - activityContent:[string] code or message
  - errorType: [string] type of the error of the code submission if the activity type is submission
  - errorMessage: [string] error message of the code submission if the activity type is submission
  - passRate:[numeric] the student's current passrate
  - currentMessageCount:[numeric] number of messages from the student
  - groupPassRate:[numeric] the group's current passrate
- submissionHistory:[Array] a list of previous code submission records of the student, each containing:
  - id: [string] id of the submission
  - time: [numeric] time the submission was made (second)
  - passRate: [numeric] the passrate of this submission
  - errorType: [string] type of the error of the code submission
  - errorMessage: [string] error message of the code submission
- messageHistory: [Array] a list of message records in the group, each containing:
  - id: [string] id of the message
  - time: [numeric] time the message was sent (second)
  - message: [string] the content of the message
  - sender_id: [string] the id of the student who sent the message
  - senderPassRate: [numeric] the pass rate of the sender right now

Output Format(valid JSON):

A collection containing the classification result and the reason:

- struggles: [Array] list of classified struggles of the students based on the student's behavior. Elements are Objects containing:
  - name: [string] name of the struggle
  - reason: [string] reason of the classification.

Figure 9: Prompt for identifying student's struggles 2

User Prompt


```
{
"id": "05ollRMtoAhLVmpbKP3F",
"type": "submission",
"student_id": "iFJgbW65I3OeqKLQAHRy",
"latestSubmission": {
"type": "submission",
"time": 413,
"code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n
if number < 100:\n                        split.it.remove(number)\n                else:\n                        break\n        return split
_it",
"errorType": "No Compile Error",
"errorMessage": "",
"passRate": 0,
"currentMessageCount": 0,
"groupPassRate": 33.333333333333336
},
"submissionHistory": [
{
"id": "81IDaR5ep0QZEfDWSqs9",
"time": 217,
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "TEgPWjQeE3xo0hjpohIN",
"time": 230,
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "OwbWPSykxs7mfxWdVz5a",
"time": 331,
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "ojs1CHiw4fLLfMHruWgs",
"time": 393,
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
}
],
"messageHistory": []
}
```


Assistant Prompt


```
{
  "struggles": [
    {
      "name": "Unproductive discussion(Right on topic but not leading to understanding)",
      "reason": "The student has submitted multiple code attempts with no improvement in their understanding of the problem, as indicated by
a consistent pass rate of 0 and no messages in the group discussion to clarify their understanding."
    },
    {
      "name": "Unproductive discussion(related but not the right approaches)",
      "reason": "The student's code attempts show a misunderstanding of the problem requirements, such as using input() and attempting to mod
ify the list incorrectly, indicating that any help they may have received was not effective."
    }
  ]
}
```


Figure 10: Input and Output Example for identifying student's struggle 2

You are a programming instructor tasked with writing feedback to students based on their issues. Your task is:

1. Based on the issue, the given feedback type, and the given components (e.g., [issue], [solution], [strategy], [example], [next step]), write a high-level template containing the structure of the feedback to bridging the chosen components in the input. Your feedback must be concise.
2. For each component, write a short description of what would it be about in the format described below.

Your feedback needs to contain information about the attainment of learning goals related to the task or performance (in [issue]), information about progress and/or about how to proceed (in [issue]/[strategy]/[example]), and more information, more tasks, and more expectations (in [example]/[next step]). Meanwhile, your feedback needs to be regarding three levels: Task level (how well tasks are understood/performed), Process Level the main process needed to understand/perform tasks), and Self-regulation Level (Self-monitoring, directing, and regulating of actions).

The programming problem students are working on: Write a function called under100 that accepts a list of integers and returns the number of values in the list that are less than 100.

Input Format(valid JSON):

A collection containing:

- issue:[string] description of the issue
- feedbackType:[string] the type of the feedback, including [Hint], [Explanation], [Verification]. [Hint] is to give a minimum cue to help the student back in the right direction, and [Explanation] is to give the student a considerable amount of help to proceed. While [Verification] is to provide the students with another similar programming question with the same learning objectives to verify the student's understanding.
- chosenComponents: [Array] a list of components to be included in the feedback.

A collection containing the student's basic information: the student's latest activity(code submission/message), the student's previous code submissions, and group discussion history:

- student ID: [string] the student's ID

Input Example:

```
{
"issue": "Lack of understanding of basic list operations and iteration." ,
"feedbackType": "Hint",
"chosenComponents: ["issue", "strategy", "next step"]
}
```

Output Format(valid JSON):

A collection containing:

- feedbackContent:[string] the content of the feedback with the chosen components properly bridged. The component must be displayed by a [PLACEHOLDER] such as [issue], [solution], [strategy], [example], [next step]. The details of components MUST NOT be included and the feedback MUST be concise.
    - componentDescription:[array] a list of objects containing a short description of the content of the component:
    - component:[string] name of the component
    - description:[string] a short description

Output Example:

```
{
"feedbackContent": "It seems you are having some trouble with basic list operations and iteration. [issue] To help you get bac
k on track, [strategy]. Once you feel more comfortable, [next step]",
"componentDescription": [
{
"component": "issue",
"description": "A brief description of the student's difficulty with basic list operations and iteration."
},
{
"component": "strategy",
"description": "A suggestion to review list iteration and applying conditions to list elements."
},
{
"component": "next step",
"description": "Encouragement to implement a simple loop to check each element in the list."
}
]
}
```

Untitled 1

Figure 11: Prompt for generating feedback templates

You are a programming instructor tasked with writing feedback to students based on their issues. You are conducting peer instruction in your lecture, where students are divided into small groups to have discussions and solve the programming task. Your task is:

1. Complete the feedback by filling in the placeholder of feedback components (e.g., [issue], [solution], [strategy], [example], [next step]) following the component description in the input. The details you add should be based on students' recent code submissions and discussion history in the input. You should consider the group dynamic and motivate engagement in the group discussion if the discussion is not active when writing the feedback.
2. For each component, repeat the raw content in the format described below.

Your feedback needs to contain information about the attainment of learning goals related to the task or performance (in [issue]), information about progress and/or about how to proceed (in [issue]/[strategy]/[example]), and more information, more tasks, and more expectations (in [example]/[next step]). Meanwhile, your feedback needs to be regarding three levels: Task level (how well tasks are understood/performed), Process Level the main process needed to understand/perform tasks), and Self-regulation Level (Self-monitoring, directing, and regulating of actions). Your feedback MUST not contain the direct solution to the entire programming problem. Do motivate discussion when discussion stage is started ("discussionStart": true).

The programming problem students are working on: Write a function called under100 that accepts a list of integers and returns the number of values in the list that are less than 100.

Input Format(valid JSON):

A collection containing:

- feedbackType:[string] the type of the feedback, including [Hint], [Explanation], [Verification]. [Hint] is to give a minimum cue to help the student back in the right direction, and [Explanation] is to give the student a considerable amount of help to proceed. While [Verification] is to provide the students with another similar programming question with the same learning objectives to verify the student's understanding.
- chosenComponents: [Array] a list of components to be included in the feedback.
- feedbackTemplate: [string] a template of feedback to be completed by filling the [PLACEHOLDER]s by following the description of components.
- componentDescription:[array] a list of objects containing the short description of the component:
    - [component]:[string] short description of the component
- studentDetails: [Object] A collection containing the student's basic information: the student's latest activity(code submission/message), the student's previous code submissions, and group discussion history:
    - student ID: [string] the student's ID
    - type: [string] type of activity ("submission" or "message")
    - classAveragePassRate: [int] class average passRate
    - discusssionStart: [boolean] if true, the discussion stage starts
    - student_name: [string] name of the student
- latestActivity:[Object] containing information about the student's latest submission, including:
    - activityType:[string] type of activity
    - time: [numeric] time the activity was made (second)
    - activityContent:[string] code or message
    - errorType: [string] type of the error of the code submission if the activity type is submission
    - errorMessage: [string] error message of the code submission if the activity type is submission
    - passRate:[numeric] the student's current passrate
    - currentMessageCount:[numeric] number of messages from the student
    - groupPassRate:[numeric] the group's current passrate
- submissionHistory:[Array] a list of previous code submission records of the student, each containing:
    - id: [string] id of the submission
    - time: [numeric] time the submission was made (second)
    - passRate: [numeric] the passrate of this submission
    - errorType: [string] type of the error of the code submission
    - errorMessage: [string] error message of the code submission
- messageHistory: [Array] a list of message records in the group, each containing:
    - id: [string] id of the message
    - time: [numeric] time the message was sent (second)
    - message: [string] the content of the message
    - sender_id: [string] the id of the student who sent the message
    - senderPassRate: [numeric] the pass rate of the sender right now

Input Example:

```
{
"feedbackType": "Hint",
"chosenComponents: ["issue", "strategy", "next step"],
"feedbackTemplate": "It seems you are having some trouble with basic list operations and iteration. [issue] To help you get back on track, [strategy]. Once you feel more comfortable,
[next step].",
"componentDescription": [
{
"component": "issue",
"description": "A brief description of the student's difficulty with basic list operations and iteration."
},
{
"component": "strategy",
"description": "A suggestion to review list iteration and applying conditions to list elements."
},
{
"component": "next step",
"description": "Encouragement to implement a simple loop to check each element in the list."
}
],
"studentDetails": {
"id": "05ollRMtoAhLVmpbKP3F",
"type": "submission",
"student_id": "iFJgbW65I3OeqKLQAHRy",
"classAveragePassRate": 66,
"discussionStart": true,
"student_name": C
"latestSubmission": {
"type": "submission",
"time": 413,
"code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n                if number < 100:\n
split.it.remove(number)\n                else:\n                        break\n        return split_it",
"errorType": "No Compile Error",
"errorMessage": "",
"passRate": 0,
"currentMessageCount": 0,
"groupPassRate": 33.333333333333336
},
```

```
"submissionHistory": [
{
"id": "81IDaR5ep0QZEfDWSqs9",
"time": 217,
"code": "def under100 (list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n                if number < 100:\n
split.it.remove(number)\n\n",
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "TEgPWjQeE3xo0hjpohIN",
"time": 230,
"code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n                if number < 100:\n
split.it.remove(number)\n\n",
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "OwbWPSykxs7mfxWdVz5a",
"time": 331,
"code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n                if number < 100:\n
split.it.remove(number)\n        return split_it\n\n",
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
},
{
"id": "ojs1CHiw4fLLfMHruWgs",
"time": 393,
"code": "def under100(list):     \n        ask_list = input()\n        split_it = ask_list.split()\n        for number in split_it:\n                if number < 100:\n
split.it.remove(number)\n                else:\n                        break\n        return split_it\n\n",
"passRate": 0,
"errorType": "No Compile Error",
"errorMessage": ""
}
],
"messageHistory": []
},
}
```

Output Format(valid JSON):

A collection containing:

- componentDetails:[array] a list of objects containing the content of the component:
- component:[string] name of the component
- description: [string] content of the component. MUST be concise. Talk about engaging in discussion if discussionStart is true.
- relatedInfo: [object] only in the 'issue' component; it contains the corresponding code/message mentioned in the description.
- type:[string] type of information, either "code" or "message"
- content:[array] the related information. For "code", it should be a list of related lines of code(string, one per line). For "message", it should be the text of one related message(string).
- feedbackContent:[string] the content of the completed feedback with the chosen components properly filled by following the description and based on student details. You MUST NOT modify anything other than the components' [PLACEHOLDER]s. There should be no [PLACEHOLDER] in the complete feedback.

Output Example:

```
{
"componentDetails": [
{
"component": "issue",
"description": "Specifically, your code is attempting to remove elements from the list while iterating over it, which can lead to unexpected behavior. Additionally, the use of input
() and split() is not necessary for this task.",
"relatedInfo": {
"type": "code",
"content": [
"        ask_list = input()\n",
"        split_it = ask_list.split()\n",
"        for number in split_it:\n",
"                if number < 100:\n",
"                        split.it.remove(number)\n"
]
}
},
{
"component": "strategy",
"description": "I suggest reviewing how to properly iterate over a list and apply conditions to its elements. You might want to look into using a simple loop to count the elements th
at meet a certain condition without modifying the list during iteration."
},
{
"component": "next step",
"description": "try implementing a simple loop that checks each element in the list and counts how many are less than 100. Discuss with your group members to share ideas and approach
es."
}
],
"feedbackContent": "It seems you are having some trouble with basic list operations and iteration. You seem to be having trouble with basic list operations and iteration. Specificall
y, your code is attempting to remove elements from the list while iterating over it, which can lead to unexpected behavior. Additionally, the use of input() and split() is not necess
ary for this task. To help you get back on track, I suggest reviewing how to properly iterate over a list and apply conditions to its elements. You might want to look into using a si
mple loop to count the elements that meet a certain condition without modifying the list during iteration. Once you feel more comfortable, try implementing a simple loop that checks
each element in the list and counts how many are less than 100. This will help you understand the correct approach to solving this problem."
}
```


Figure 12: Prompt for generating feedback content