A Survey on Feedback Types in Automated Programming Assessment Systems

Eduard Frankford (University of Innsbruck, Department of Computer Science, Innsbruck, Austria; eduard.frankford@uibk.ac.at)

Tobias Antensteiner (University of Innsbruck, Department of Computer Science, Innsbruck, Austria; tobias.antensteiner@uibk.ac.at)

Michael Vierhauser (University of Innsbruck, Department of Computer Science, Innsbruck, Austria; michael.vierhauser@uibk.ac.at)

Clemens Sauerwein (University of Innsbruck, Department of Computer Science, Innsbruck, Austria; clemens.sauerwein@uibk.ac.at)

Vivien Wallner (University of Salzburg, Department of Artificial Intelligence and Human Interfaces, Salzburg, Austria; vivien.wallner@plus.ac.at)

Iris Groher (Johannes Kepler University Linz, Institute of Business Informatics - Software Engineering, Linz, Austria; iris.groher@jku.at)

Reinhold Plösch (Johannes Kepler University Linz, Institute of Business Informatics - Software Engineering, Linz, Austria; reinhold.ploesch@jku.at)

Ruth Breu (University of Innsbruck, Department of Computer Science, Innsbruck, Austria; ruth.breu@uibk.ac.at)

DOI: XX.XXXX/XXXXXXX.XXXXXXX

ABSTRACT

With the recent rapid increase in digitization across all major industries, acquiring programming skills has increased the demand for introductory programming courses. This has further resulted in universities integrating programming courses into a wide range of curricula, including not only technical studies but also business and management fields of study. Consequently, additional resources are needed for teaching, grading, and tutoring students with diverse educational backgrounds and skills. As part of this, Automated Programming Assessment Systems (APASs) have emerged, providing scalable and high-quality assessment systems with efficient evaluation and instant feedback. Commonly, APASs heavily rely on predefined unit tests for generating feedback, often limiting the scope and level of detail of feedback that can be provided to students. With the rise of Large Language Models (LLMs) in recent years, new opportunities have emerged as these technologies can enhance feedback quality and personalization. To investigate how different feedback mechanisms in APASs are perceived by students, and how effective they are in supporting problem-solving, we have conducted a large-scale study with over 200 students from two different universities. Specifically, we compare baseline Compiler Feedback, standard Unit Test Feedback, and advanced LLM-based Feedback regarding perceived quality and impact on student performance. Results indicate that while students rate unit test feedback as the most helpful, AI-generated feedback leads to significantly better performances. These findings suggest combining unit tests and AI-driven guidance to optimize automated feedback mechanisms and improve learning outcomes in programming education.

1 INTRODUCTION

With digitalization becoming a major topic, many industries are increasingly adopting digital solutions at a staggering pace. Therefore, the need for fundamental programming skills has increased considerably. Programming enables individuals to automate processes, analyze large datasets, and develop custom applications, making it an essential skill for many tasks. This increasing need has also been reflected in many university curricula, where courses such as computational thinking, introductory algorithm design, and basic programming classes are now included in a wider range of programs [33, 46, 21, 16]. As a result, the number of students enrolling in programming classes and lectures has grown significantly, creating a pressing demand for high-quality assessment and feedback to support their learning and improve their programming skills [26]. However, the time frame for assessments to be evaluated and graded is relatively short, resulting in the need to grade several dozen or even hundreds of assessments per week, depending on the course size. Furthermore, solely relying on feedback and assessment from human graders poses the threat of erroneous or inconsistent grading. Additionally, feedback quality might vary, particularly when grading is performed by tutors or student teaching assistants [20, 8].

In this context, APASs have become an essential part of computer science education, offering scalable and efficient means to evaluate programming assignments and provide immediate feedback to students [25, 30]. Typically, APASs generate feedback for students by executing pre-defined unit tests on the students' code and returning the output as feedback to the students. However, with the introduction and widespread availability of LLMs, new possibilities for speeding up grading, enhancing feedback quality,

and personalization have emerged [17, 6].

As of today, only a few empirical studies systematically analyze different types of automated feedback for introductory programming education. Comparisons among baseline Compiler Feedback, standard Unit Test Feedback, and next-generation LLM-based feedback remain largely unexplored, particularly regarding perceived quality and effects on student performance and engagement. To fill this gap, we have conducted a large-scale survey along with an experiment with first-year bachelor students in introductory programming courses at two different universities.

The main goal of these experiments was to answer the following two research questions:

- **RQ1**: How do students perceive different feedback forms when solving programming tasks?
- **RQ2**: How do different forms of feedback in APASs affect students' success in solving tasks?

For this purpose, we conducted (1) a qualitative analysis, soliciting student feedback as part of a multipart survey, and (2) a quantitative analysis, where we investigated the students' performance after working on exercises with the different feedback types. The findings reveal significant differences in how students perceive feedback and perform under different feedback conditions. The survey indicates varying levels of satisfaction and perceived usefulness among the feedback types. The success rates of the students and the quality of their solutions also varied, offering insights into the effectiveness of different feedback mechanisms.

The remainder of this paper is organized as follows. Section 2 reviews related work on feedback types in programming education and the application of AI and LLMs for feedback generation. Section 3 describes the applied research methodology, including survey design, exercises, feedback types, data collection methods, data cleaning, and data analysis. Section 4 presents the demographics of the students surveyed. Section 5 presents the results regarding the students' individual perceptions of the feedback types, followed by Section 6, which presents the results of the impact of each type of feedback on student performance. Section 7 discusses the implications of the findings, and Section 8 acknowledges the limitations of the study. Finally, Section 9 concludes the paper and suggests directions for future research.

2 BACKGROUND & RELATED WORK

In this section, we provide a brief introduction to the area of automated assessment tools and programming education and discuss related work in three main areas. First, research on feedback types in automated assessment systems discussing general research on different types of feedback. This includes, for example, more traditional rule-based techniques and new adaptive strategies using AI methods and LLM feedback generation. In the second part, we focus on different types of feedback, and how these can affect student performance and can be used to create personalized and meaningful insights into students' programming processes. Finally, we discuss related studies that have evaluated feedback approaches. Together, these subsections provide a comprehensive analysis of the current state of the art, highlighting the existing gap and potential of current feedback mechanisms in supporting student learning in programming education. Note that this section focuses on a representative subset of feedback types, particularly those commonly used in programming education, and does not aim to cover the full range of automated feedback forms, such as delayed or purely positive feedback.

2.1 Automated Assessment Tools & Feedback Types

Automated programming assessment tools are essential in computer science education, offering scalable and efficient evaluation of programming assignments. According to Paiva and Leal [34], these systems play a vital role in fostering practical programming skills by analyzing complex program features, supporting diverse task formats, ensuring security, and providing targeted feedback [34]. Systems like Edgar [30], Artemis [25], Web-CAT [14], Fitchfork [35], CodeRunner [28], CodeBoard [42], CodeOcean [41], PythonTutor [19] enable instant feedback to students.

In most cases, the feedback mechanism operates through several key steps. Students first submit their solutions via a Version Control System (VCS) [50], using either an online code editor provided by the APAS or their local development environment. Once the code is submitted, it triggers a build process on the Continuous Integration (CI) [32] server, which compiles the code and runs predefined unit tests to evaluate the submission. Students can review the test feedback, make corrections, and resubmit their solutions. These systems also support multiple iterations, allowing students to learn from their mistakes and progressively improve their code. Instructors can then monitor students' progress and common issues students face, allowing them to provide targeted assistance and make real-time adjustments to the learning process.

Beyond technical robustness, the effectiveness of feedback generation depends on its impact on student learning behaviors. Ahmed *et al.* [2] explored the pedagogical benefits of adaptive feedback for novice programmers struggling with compilation errors. While their large-scale randomized study demonstrates improved efficiency in error resolution during lab sessions, it revealed no corresponding improvement in conceptual understanding according to exam performance [2]. This suggests that automated feedback systems should complement logistical support with strategies that promote deeper learning and conceptual mastery.

The scalability of feedback generation in large courses adds another layer of complexity, particularly in addressing diverse student errors. Singh *et al.* [40] introduced a method for automated feedback that utilizes a reference implementation and an error model to derive minimal corrections. A reference implementation is a correct and complete version of a programming solution provided by instructors or system designers. To correct the number of incorrect submissions, it analyzes the student's solution, applies the error model to explore possible corrections, and systematically identifies the minimal set of changes needed

to make the solution semantically equivalent to the reference implementation. However, it was less able to address conceptual errors, highlighting the need for more nuanced feedback mechanisms that address underlying misconceptions [40]. The work of Marin *et al.* [29] demonstrates the use of a semantic-aware technique to enhance feedback personalization in introductory Java Massive Open Online Courses (MOOCs). Using extended program dependency graphs and subgraph matching, their system provided semantically rich, personalized feedback at scale. However, their approach also highlights the complexity of creating feedback that is both individualized and broadly applicable across diverse programming solutions [29].

Recent advancements in Machine Learning (ML) and Artificial Intelligence (AI) have enabled the development of systems that provide tailored student feedback, significantly enhancing learning outcomes and engagement. Kochmar *et al.* [24] proposed an ML approach using Natural Language Processing (NLP) to deliver personalized hints and explanations based on individual needs, demonstrating improvements in both learning and students' perceptions of feedback quality [24]. Similarly, Ahmed *et al.* [1] introduced TEGCER, an AI-powered tool for novice programmers that employs supervised classification to match code submissions with previously observed error patterns, leveraging a dataset of over 15,000 error-repair examples [1]. TEGCER provides tailored solutions to specific errors, enabling faster debugging. In a study with over 230 participants, students using TEGCER resolved errors more efficiently than those relying on human tutors, underscoring the potential of AI-driven tools to enhance programming education through personalized and efficient support.

Building on these advancements, Frankford *et al.* [17] utilized LLMs to generate context-aware feedback, demonstrating their capability to deliver instantaneous, personalized support at scale [17]. Bassner *et al.* [6] further validated the effectiveness of LLMs in enhancing the learning experience by tailoring feedback to individual student submissions [6].

These findings highlight the potential of LLMs as powerful tools in modern, scalable educational systems. Complementing this, Barrow *et al.* [5] investigated feedback mechanisms in Intelligent Tutoring Systems (ITSs), focusing on the inclusion of positive reinforcement alongside traditional negative feedback. Their study found that students receiving both positive and negative feedback solved problems more quickly and with fewer attempts, all while mastering the same number of concepts as those in a control group receiving only negative feedback [5]. These results emphasize the importance of balanced feedback strategies in AI-driven educational tools, showcasing how a combination of reinforcement types can enhance learning efficiency and overall performance.

2.2 Effects of Different Feedback Types on Students' Performance

Several studies have shown that immediate and detailed feedback can significantly improve learning outcomes of students [49, 4, 12]. Paiva *et al.* [34] conducted a systematic review on automated feedback in programming environments, highlighting the importance of timely, specific, and actionable feedback [34]. Ulla *et al.* [44] mentioned that automated assessment systems have an impact on novice programmers. This impact is crucial for students in introductory programming courses as it provides immediate feedback and eases the teacher's workload when the number of students is high, so that the development of essential skills can be supported [44]. Cavalcanti *et al.* [9] mentioned that feedback is crucial for learning, especially in online courses where instructors and students are physically separated. Automatic feedback systems have been proposed to address this, and a systematic review [9] reveals that automatic feedback increases student performance. Keuning *et al.* [23] compared how feedback is generated and found that the most common technique is automated testing using unit tests, which is used by 58.4% of the tools, followed by program transformation techniques (37.6%) and static analysis (36.6%) [23].

Chen *et al.* [10] conducted an experiment involving 76 computer science (CS) students to investigate the impact of different feedback designs on student learning and interaction. They examined three groups, each receiving one of the following feedback categories: (1) Knowledge of Results (KR): includes only information on which test cases failed; (2) Knowledge of Correct Responses (KCR): includes KR plus additional information on where the failure occurred within the test case; (3) Elaborated Feedback (EF): includes KCR plus a hint on what the student might need to do to fix their code. Across three complex programming assignments, they found that students who received higher levels of feedback (KCR and EF) significantly outperformed those who received only KR feedback. The EF feedback was implemented as a one-level hint addressing the top five mistakes from prior student submissions, with additional explanations provided [10]. The study, comparing three different feedback types, shows that more detailed feedback significantly improves student performance.

Qian and Lehman [38] investigated the impact of targeted feedback on addressing misconceptions in high school students' Java programming. Their study demonstrated that tailored feedback, designed to address specific errors and misunderstandings, not only helped students correct their mistakes but also reduced the occurrence of intermediate errors in subsequent attempts. By offering explanations of reasons for errors and guiding students toward proper problem-solving strategies, the feedback promoted deeper conceptual understanding. These findings emphasize the value of targeted feedback in improving learning outcomes and align with the growing interest in AI-driven feedback generation, which offers the potential to deliver personalized, context-aware support at scale [38]. The feedback used here can be categorized as targeted and corrective, aimed specifically at conceptual misconceptions in a high school programming context. Its effectiveness was reflected in reduced error rates and deeper learning.

Similarly, Clegg *et al.* [11] highlighted how characteristics of test suites, such as coverage, size, and redundancy, can significantly impact grading outcomes. Their analysis focused on optimizing test suites to achieve comprehensive code coverage while minimizing redundancy, ensuring consistent and equitable feedback [11]. Such optimization not only enhances the accuracy of automated grading systems but also fosters student trust by main-

taining grading integrity. This study highlights how the design of assessment infrastructure affects the reliability and fairness of outcome-oriented feedback in grading contexts.

Assessment systems have also proven effective in supporting continuous improvement and encouraging timely assignment completion. For example, Yan *et al.* [48] evaluated the ProgEdu system and found that its iterative feedback mechanisms significantly enhanced student learning outcomes and code quality [48]. However, their study also identified areas for improvement, such as enhancing feedback precision and usability, highlighting the need for ongoing refinement to meet evolving learner needs. The feedback provided in this study can be classified as process-oriented, helping students improve their programming skills with repeated submissions.

Comparative analyses of automated grading tools illustrate the trade-offs between different assessment methodologies. Bey *et al.* [7] compared Algo+, a static analysis-based system, with the EPFL grader, which relies on unit tests [7]. While Algo+ excelled at identifying partially correct solutions, it sometimes underestimated novel correct submissions absent from its reference set. This finding emphasizes the importance of aligning grading methodologies with specific educational objectives to maximize their pedagogical impact.

2.3 Methodologies Used to Evaluate Feedback Forms

Understanding the effectiveness of feedback in APASs is essential for improving learning outcomes, fostering motivation, and supporting skill development. Research in this area has explored various approaches to feedback delivery, analyzing their impact on students' educational experiences and instructors' workflows.

One widely adopted approach for evaluating feedback mechanisms in APASs is through student surveys. Messer *et al.* [31] conducted an extensive review of 121 papers published between 2017 and 2021, focusing on evaluation methodologies used in automated grading tools for programming education. Their findings highlighted the frequent use of student surveys to assess aspects such as satisfaction with instant feedback, the clarity and usefulness of error messages, and the opportunities for iterative learning enabled by resubmissions. These surveys typically include Likert-scale questions, open-ended prompts, and comparative assessments in which students evaluate automated tools against traditional teaching methods. The results consistently show that students value the immediacy and consistency of automated feedback, particularly its effectiveness in identifying specific coding issues and providing actionable suggestions for improvement. These findings underscore the critical role of feedback beyond correctness, emphasizing its importance in supporting formative learning processes that foster reflection and improvement [31].

The integration of gamification into APASs introduces an innovative dimension to feedback evaluation. Polito and Temperini [37] developed a gamified learning platform that combines immediate feedback with motivational features such as experience points, badges, and leaderboards to enhance engagement and skill development. Their experimental study demonstrated that real-time feedback, combined with gamification, significantly boosts student engagement [37]. A detailed post-experiment survey of 36 questions and a 5-point Likert scale assessed students' experiences, engagement levels, and perceptions of the platform's usefulness in skill development. The survey also explored the impact of gamification elements and students' willingness to adopt similar systems in other educational contexts. The results were overwhelmingly positive, with students rating their overall experience highly with 4.19 out of 5. Features like immediate feedback and personalized leaderboards were particularly appreciated for their motivational impact. These findings align with improved academic performance, showcasing the potential of gamified APASs to facilitate iterative learning and promote deeper student engagement in programming education.

Another example analyzed by J. English and T. English is the Checkpoint system, which provides detailed, iterative feedback with unlimited submission attempts [15]. A survey of 141 students, incorporating 15 Likert-scale questions and open-ended prompts, revealed a strong appreciation for Checkpoint's usability, reliability, and support for self-paced learning. Students particularly valued the opportunity to make multiple attempts, as it helped them identify and correct mistakes through detailed, actionable feedback. Faculty also reported reduced grading workloads and access to granular performance analytics for targeted interventions. However, some students expressed frustrations with the rigidity of the feedback and suggested greater flexibility to accommodate diverse learning styles. These findings highlight the strengths of automated systems like Checkpoint in enhancing the learning process while pointing to the need for balancing automation with adaptability [15].

Substantial research compares various feedback mechanisms with traditional non-APAS-based approaches [23, 11], yet the differences in student performance and perceived usability for compiler feedback, unit-test feedback, and AI-based feedback in APASs remain unexplored. Notably, no large-scale, controlled study has compared these feedback types with a representative sample. This research addresses that gap by systematically evaluating three feedback conditions: (1) no support as a baseline with compiler feedback only, then (2) unit-test feedback, and lastly, (3) AI-generated feedback, and examining their impacts on student performance, usability, intention to use, and output quality. To achieve this, it applies diverse survey methodologies [31, 37, 15, 22, 3] that capture students' perceptions and responses, offering valuable insights into each feedback type's effectiveness.

While several authors have investigated different types of feedback and their impact on student performance, there is still a lack of research on assessing and comparing these types of feedback, particularly regarding their evaluation through student surveys with representative sample sizes. So far, only a few studies have systematically analyzed how these feedback types differ concerning ease of use, perceived usefulness, and impact on learning outcomes. Furthermore, the use of LLMs for feedback generation is a novel approach and has thus never been directly investigated in a study comparing all three feedback types. Therefore, our

work presented in this article helps to create a comprehensive overview to better understand, compare, and optimize feedback mechanisms in programming education.

3 RESEARCH METHODOLOGY

To systematically investigate the use of different forms of feedback in APASs for introductory programming education, we adopted the Goal Question Metric (GQM) approach to define our research goal and questions [45]. The GQM framework provided a structured means to evaluate the perceptions and effectiveness of various feedback types on students' learning outcomes. The overall goal, the derived research questions, and constituent metrics are outlined in Fig. 1.

The study's overall goal, as illustrated in Fig. 1, was to explore the effects of feedback types on both student perceptions and their success in solving programming tasks. To achieve this, two primary research questions (RQs) were developed, each with specific sub-questions:

- **RQ1: How do students perceive different feedback forms when solving programming tasks?**
  - **RQ1.1:** How do students rate the quality and usefulness of the different feedback types?
  - **RQ1.2:** What feedback types were perceived as most useful and why?
- **RQ2: How do different forms of feedback in APASs affect students' success in solving tasks?**
  - **RQ2.1:** What effect do different feedback types have on performance?
  - **RQ2.2:** What effect do different feedback types have on the way students solve problems?

The study was conducted at two universities in Austria from Autumn 2023 to Spring 2024, integrating the survey into students' coursework to capture a broad set of student responses. The exercises and experimental environment were implemented in the online editor provided by the APAS *Artemis*[1], enabling real-time feedback and interaction with the platform. The feedback was generally triggered by students clicking on the submit button in Artemis. The students could click on the submit button as many times as they want.

The research methodology combined a *Technology Acceptance Model (TAM) questionnaire-based survey* with *quantitative analysis of student performance*:

- **Survey Integration (RQ1):** Students answered TAM-based surveys immediately after solving each programming exercise in Artemis to assess their perceptions of feedback quality, usefulness, and ease of use. A final open-ended questionnaire gathered qualitative insights into their preferences and experiences with the different feedback types.
- **Experiment with Three Exercises (RQ2):** All students experienced all feedback types as they worked on three programming exercises, where each exercise was paired with one of the three feedback types:
  - *Compiler Feedback* (baseline: syntax and runtime errors only),
  - *Unit Test Feedback* (structured correctness verification),
  - *AI-generated Feedback* (adaptive, hint-based guidance).

  All code submissions, time between attempts, and final correctness scores were logged for detailed analysis.
- **Data Collection and Metrics:**
  - *M1.1:* TAM constructs perceived usefulness (PU), ease of use (EU) were rated using Likert-scale questions to measure subjective feedback quality.
  - *M1.2:* Post-experiment feedback was qualitatively analyzed to understand why students favored certain feedback types.
  - *M2.1:* Unit test results were used to evaluate the correctness of students' final solutions under different feedback conditions.
  - *M2.2:* Submission logs provided insights into problem-solving strategies, including iteration behavior and time spent refining solutions.

The study ensured diverse representation by including students from two universities, enabling broader generalizability of findings. This methodological approach provided a comprehensive evaluation of both subjective and objective impacts of feedback types in APASs. Survey materials, including exercises, tests, and questionnaires, are described in more detail in the following subsections.

3.1 Exercise Design and Selection

To ensure a valid comparison across feedback types, we designed a set of exercises that were consistent in size and complexity. As the experiments were conducted in introductory Java courses at both universities in the first half of the semester, we selected the topic of Arrays, which aligned with both course curricula.

Specifically, we selected examples covering aspects of one-dimensional and two-dimensional arrays, as this provided sufficient complexity to assess competencies in this area compared to earlier lectures covering topics such as operators, variables, loops, or simple conditions. As the topic of arrays had been covered in the first half of the semester in both universities, all students had similar prior experience with arrays through previous assignments. Each exercise was developed to target similar competencies based on an established competence model for introductory Java courses [36]. The initial drafts were created by one instructor and then reviewed and refined by several faculty members from both universities who were involved in the introductory programming courses. This iterative refinement step ensured the consistency and similar levels of complexity of the exercises. In the following, we provide a brief description of the three exercises included in the study and the respective development steps.

[1] https://github.com/ls1intum/Artemis

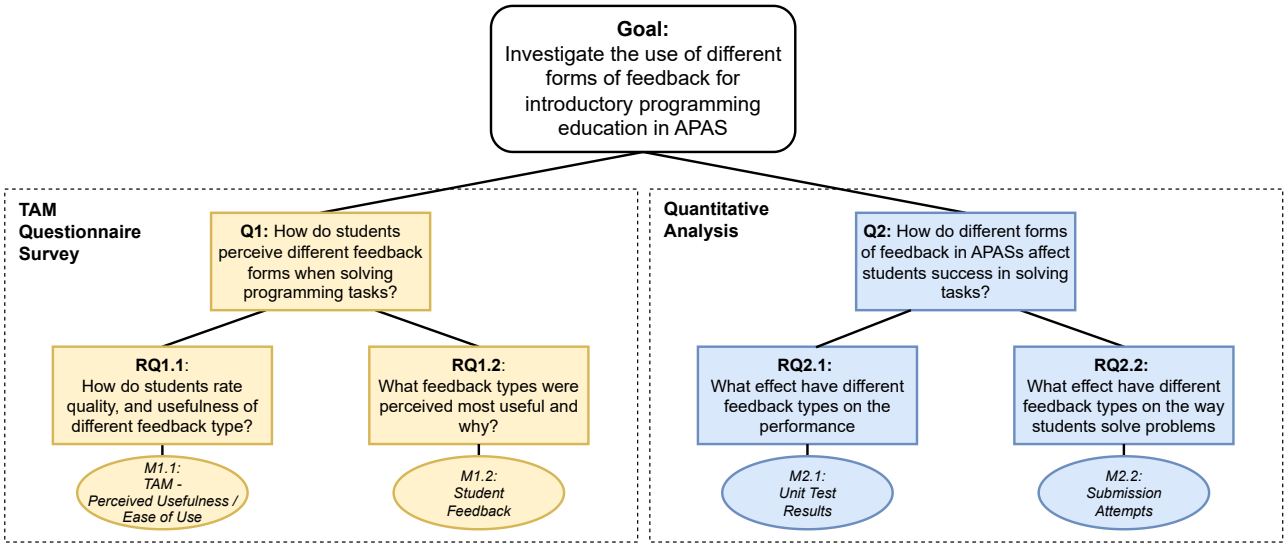

Figure 1: Goal Question Metric (GQM) Framework for Investigating Feedback Types in APASs

3.1.1 Exercise 1: Mars Rover

This exercise involves helping a Mars Rover navigate a defined zone. The Mars surface is represented as a matrix, where `'*'` indicates a free space and `'R'` marks the rover's current position. The task requires students to write a method `'moveRover'` that moves the rover within the matrix based on given row and column values. If the movement takes the rover outside the matrix, it stops at the edge. The method updates the matrix by changing the start position from `'R'` to `'*'` and the end position from `'*'` to `'R'`, and then returns the updated matrix. If the matrix is null or empty, the method returns null. Positive row values move the rover right, negative row values move it left. Positive column values move it down, and negative column values move it up. The exercise provides the helper methods,`'searchRover'` and `'displayMapFromMatrix'`, for finding the rover's position and testing output.

3.1.2 Exercise 2: Array Transformation

In this exercise, students write a method `'calc4All'` that applies an arithmetic operation to each element in an integer array using a specified operand, storing the results in a new array. If the input array is empty or null, or if the operation is invalid (‘+‘, ‘-‘, ‘*‘, ‘/‘ and ‘%‘ are the valid operators), the method returns null. Special cases where the operand cannot be zero are handled by returning null. Students use the provided `'arrayToString'` method to test their output.

3.1.3 Exercise 3: Matrix Validation

This exercise requires students to write a method `'isValidNumberMatrix'` to verify if a 3x3 matrix contains the numbers 1-9 without duplicates. The matrix should not contain other numbers, and if it fails these criteria or is null/not 3x3, the method returns false. This task reinforces students' understanding of array handling and validates their matrix manipulation skills.

3.1.4 Exercise Development Process

Each exercise was developed using the same structured process to ensure comparable difficulty and alignment with regard to the needed competencies to solve the exercises. For the development of the exercises, we followed the process described by Plösch et al. [36]. The stages were as follows:

1. **Initial Draft and Solution Development**: An instructor selected the competencies from the competence model and prepared initial drafts of the exercises, including task descriptions and sample solutions.
2. **First Review**: Two researchers reviewed each exercise, provided feedback, and independently implemented solutions. Based on this feedback, the exercises were revised for consistency in complexity and code length. The researchers also reviewed the mapping of the competencies to the exercises and sample solutions.
3. **Second Review by Faculty**: Faculty from both universities reviewed the mapped competencies and verified the appropriateness of the exercises. Minor adjustments were made, such as adding required checks for boundary and invalid inputs, to ensure alignment across exercises.
4. **External Review and Final Adjustments**: Two external instructors, experienced in teaching introductory Java courses but not involved in the initial creation, conducted an independent review. They suggested minor changes to wording and formatting, which were incorporated to improve clarity.

Each finalized exercise provided students with a clear problem description, along with example input and expected output. Exercises 1 and 3 involved two-dimensional arrays, while Exercise 2 focused on a one-dimensional array with additional operations. Our collaborative review process helped ensure that each exercise was balanced in terms of difficulty, enabling a reliable assessment of feedback types across institutions.

After having developed well-reviewed exercise specifications

including a mapping to competencies, we started to design and implement competency-based unit tests. Again, we followed the process proposed in [36]. For each assigned competence, we implemented at least one unit test. For complex competencies (e.g., iterating over two-dimensional arrays) more unit tests were necessary. We added further tests necessary to check the correctness of the solution. Coverage analysis of the sample solution helped us to identify missing test cases.

3.2 Study Execution and Data Collection Process

At both universities, the study was carried out in the first half of the semester as part of a regular 90-minute in-person class session. As illustrated in Fig. 2, the study was designed to systematically collect survey and performance data from students at multiple stages, gathering both quantitative and qualitative insights. This structured data collection process allowed for an in-depth evaluation of student experiences with each feedback type. The 90-minute session was split into three parts, starting with a 15-minute introduction describing the structure of the survey and exercise, and asking for consent. The study contained four key data collection points (in Fig. 2 defined as DC 1-4) (three during the 90-minute session (DC1-3), and one (DC4) post-study questionnaire distributed after the session to give the students additional time to reflect on the exercises and feedback mechanisms). Each data collection point targeted specific aspects of the study:

3.2.1 Data Collection Point 1: Demographics Questionnaire

After the initial briefing, before starting the exercises, students completed a 5-minute demographics questionnaire to gather baseline information, which included the following questions:

- **Gender:** Options included Male, Female, Diverse/Other, Prefer not to say.
- **Age:** Students specified their age in years.
- **Study Program Enrollment:** Various Bachelor's and Master's programs were listed, with an option to specify other programs.
- **Current Semester:** Options ranged from 1st to 6th semester, with an option to specify other semesters.
- **Programming Languages Used Before This Course:** Multiple options were provided for students to indicate prior experience with specific programming languages, along with a field for additional languages.
- **Years of Programming Experience:** Students specified their years of experience.
- **Highest Educational Qualification:** Options included different levels from high school to doctorates, with an option to specify additional qualifications.
- **Number of Times Taking the Course:** Students indicated how many times they had enrolled in the course.
- **Experience with Artemis:** Students were asked if they had previously used the Automated Programming Assessment System (Yes/No).

Additionally, students were prompted to create a personal code, to enable anonymous tracking across all survey responses. This was to ensure that only valid responses were included in the subsequent analysis of results.

3.2.2 Data Collection Point 2: System Logs and Exercise Submission Data

As students worked on the programming exercises, each submission to the APAS (Artemis) was logged with the following data. Students were permitted to submit any number of "intermediary" solutions and received feedback for each version of the (partial) submission.

- **Code Submitted:** A snapshot of the student's solution at the time of each submission.
- **Feedback Provided:** The type of feedback the student received for each submission.
- **User Identifier and Timestamp:** Each submission was tagged with a unique user ID and timestamp to track progress and interaction.

After the study, all final submitted solutions were further evaluated through an additional set of unit tests created by independent instructors, ensuring an unbiased assessment of solution quality.

3.2.3 Data Collection Point 3: Post-Exercise Feedback Questionnaires

Following each exercise, students completed a TAM-based questionnaire tailored to the specific feedback type. Each survey included the following questions, with responses on a 7-point Likert scale from 'Strongly Disagree' to 'Strongly Agree':

- **Speed:** "The use of Artemis with [Feedback Type] enables me to complete programming tasks faster."
- **Simplification:** "The use of Artemis with [Feedback Type] makes it easier for me to accomplish programming tasks."
- **Usefulness:** "The use of Artemis with [Feedback Type] is useful for accomplishing programming tasks."
- **Future Use:** "In the future, I would use Artemis with [Feedback Type] to solve programming tasks."
- **Clarity:** "The interaction with Artemis and [Feedback Type] is clear and understandable."
- **Effort:** "The interaction with Artemis and [Feedback Type] does not require much mental effort."
- **Ease of Use:** "Artemis with [Feedback Type] is easy to use."
- **Functionality:** "I find it easy to get Artemis with [Feedback Type] to do what I want it to do."
- **Feedback Quality I:** "The quality of the received [Feedback Type] is high."
- **Feedback Quality II:** "I had no problems with the quality of the [Feedback Type]."

Each post-exercise questionnaire took approximately 2-3 minutes, enabling immediate reflection on the feedback type used in each exercise.

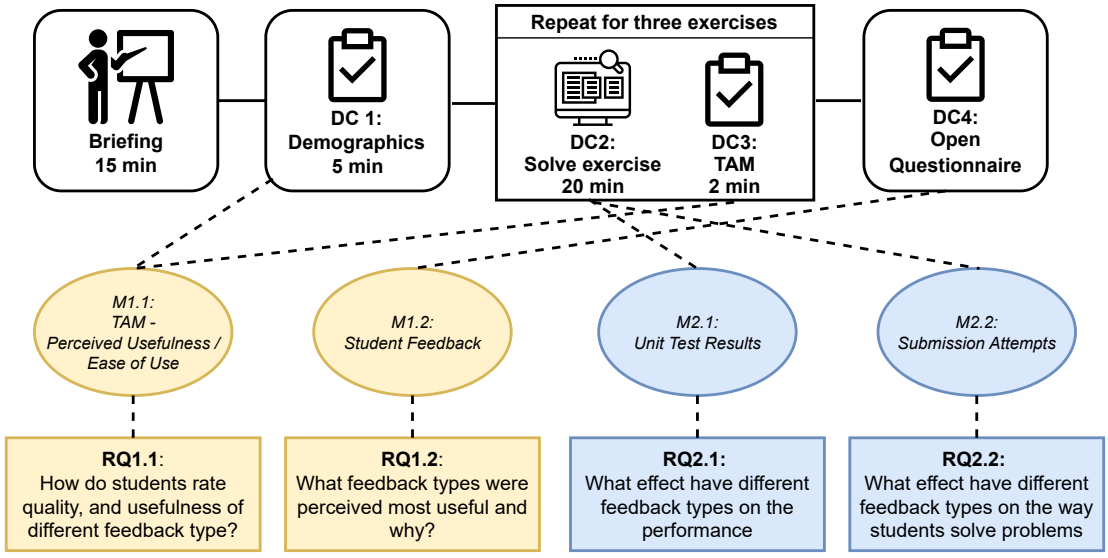

Figure 2: Overview Study Workflow

3.2.4 Data Collection Point 4: Post-Experiment Open-Ended Questionnaire

After completing the main part of the exercises, students were asked to complete a final online survey designed to capture qualitative insights into their overall experiences and preferences and collect additional feedback and comments. The survey included the following items:

- **Personal Code:** Students entered their unique code to connect responses with the demographics questionnaire.
- **General Comments:** "Please provide any general comments about the study, including your thoughts on the feedback types and how they were used."
- **Most Helpful Feedback Type:** Students were asked to select the feedback type they found most helpful from AI Feedback, Compiler Feedback, or Unit Test Feedback.
- **Reason for Feedback Preference:** "Explain why you found the selected feedback type the most helpful. Describe specific aspects or functions you found beneficial."
- **Suggestions for Additional Feedback Types:** "Are there any other feedback types not currently available in Artemis that you would find helpful? If so, please describe them."

This post-experiment questionnaire allowed students to express their preferences, highlight the strengths and weaknesses of each feedback type, and provide suggestions for future feedback enhancements in Artemis.

3.3 Feedback Types

To assess the effectiveness of various feedback mechanisms, we employed three distinct feedback types and implemented means of support for the programming tasks within the Artemis APAS.

- **No Feedback Support (Compiler Output Only):** This basic level of feedback provides the baseline of support to be measured and includes only standard compiler output, such as syntax errors, warnings, and runtime errors. It lacks additional hints or solutions, thereby simulating a "minimal support environment", as students, for example, would also experience in a traditional desktop-based IDE (without any additional extended support plug-ins, such as CoPilot [18]). This feedback type allowed us to investigate students' problem-solving strategies when relying solely on standard debugging tools.
- **Traditional Unit Test Feedback:** This feedback type employs predefined unit tests (JUnit in our setting) to validate students' code for correctness against specific test cases. Traditional Unit Test Feedback is commonly used in many APASs and offers structured feedback that indicates whether the code meets the functional requirements, identifying any failed test cases.
- **AI-Generated Feedback:** Using OpenAI's GPT-3.5-Turbo [2] model, this adaptive feedback type acts as a supportive "tutor". It provides students with targeted hints and guidance to improve their code, encouraging independent debugging rather than directly revealing the solution.

3.3.1 Compiler Feedback

Compiler Feedback refers to the default outputs generated by the compiler upon code submission. This includes syntax errors, runtime messages, and any print statements that students may use to debug their code. To simulate Compiler Feedback within our APAS, we configured the system to display program output via a pop-up window (Fig. 3). Since Artemis does not support direct code execution without test cases, we implemented a placeholder failing test case to trigger the display of all outputs. Students were informed about this setup, ensuring they understood that they should continue refining their code until it met the exercise requirements.

[2] https://platform.openai.com/docs/models/gpt-3.5-turbo

If a submission failed due to a compile error, a build error message was displayed in the pop-up (see Figure 4 for an example of a missing semicolon error). This setup allowed students to receive basic feedback without any structured guidance on code correctness.

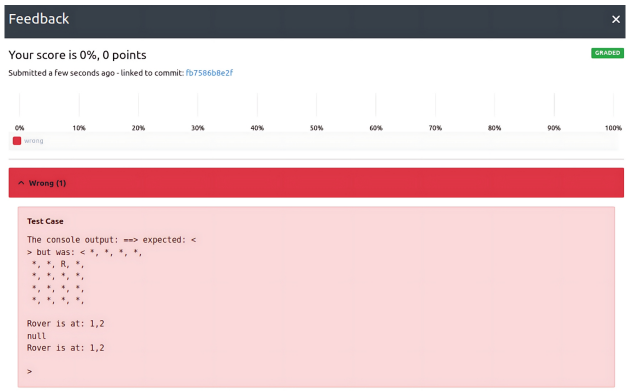

Figure 3: Example of Compiler Feedback Output

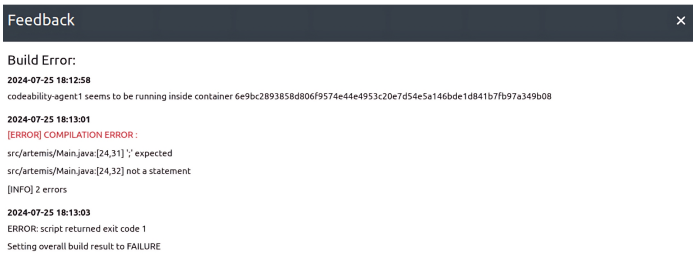

Figure 4: Example of Compile Error Feedback

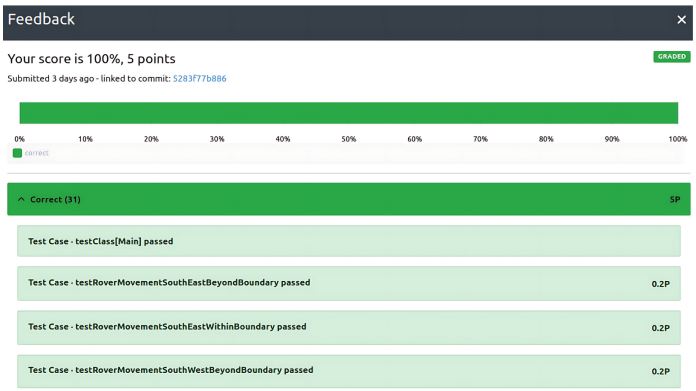

Figure 5: Example of Traditional Unit Test Feedback

3.3.2 Unit-Test Feedback

For traditional feedback, unit tests were used to evaluate the functional correctness of students' code submissions. Upon submission, the system executed a set of predefined unit tests developed by instructors to validate key aspects of the solution. Students received feedback on passed and failed test cases, with detailed error messages indicating specific areas of failure, as shown in Fig. 5. This feedback type mirrors the structured, task-oriented feedback used in many APASs, helping students align their implementations with specific functional requirements.

To ensure quality, the unit tests underwent a verification process, similar to the exercise descriptions, by the instructors who authored the exercises (cf. exercise creation process above).

3.3.3 AI-Generated Feedback

AI Feedback was generated by the GPT-3.5-Turbo model and designed to provide adaptive, tutor-like guidance. When a student submitted a solution, the server sent the student's solution and the exercise description via an API request to OpenAI's GPT-3.5-Turbo model with the instructions to return hints rather than solutions (see Prompt 1). This format allowed targeted feedback that encouraged self-driven debugging and improvement. GPT-4 and newer models, which offer potential improvements in natural language understanding and feedback generation, were not utilized in this study because they were not yet available at the time the study was conducted.

Prompt 1

Act as a tutor and help the student debug his or her code. Do not provide the complete solution; only give hints and speak to the student directly. If you think the code already fulfills the problem statement, inform the student. The code currently looks like this: **[Inserted Code]**
This is the problem the student is trying to solve: **[Inserted Problem Statement]**

Prompt 1: GPT-3.5-Turbo Prompt for AI-Generated Feedback

The AI model's responses were displayed within the Artemis interface (Fig. 6), helping students identify issues without offering explicit solutions. The feedback request was initiated via an API call, executed within a failing test case, as the Artemis system requires test cases to display outputs. Students were briefed to disregard the failing status and focus on the AI's feedback hints.

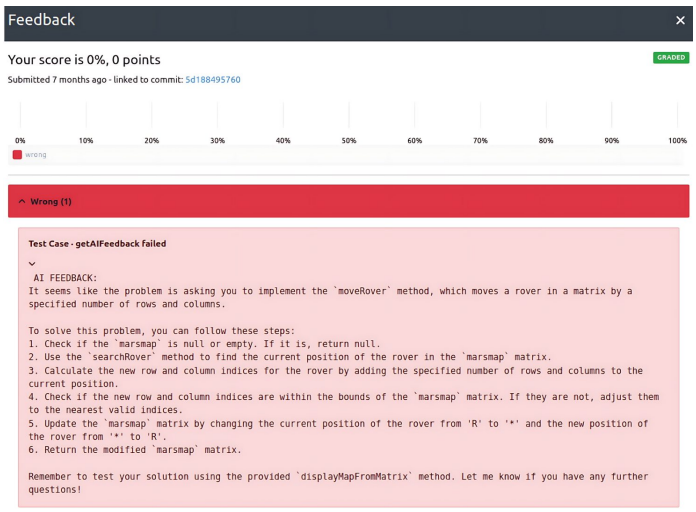

Figure 6: Example of AI-Generated Feedback

We chose not to enable custom interactions with the AI model for several reasons:

1. **Ease of Use:** Predefined prompts streamlined the interaction process, removing the need for students to formulate questions and ensuring more consistency in feedback.

2. **Controlled Environment:** This also prevented students from trying to trick the system by engineering prompts that tell the system to reveal solutions.

3. **Quality Control:** The static prompt allowed us to pre-evaluate responses for relevance and quality.

4. **Data Privacy:** Restricting custom interactions minimized the risk of students inadvertently sharing personal information in prompts.

3.4 Data Cleaning

We applied a strict data cleaning process across all collection points. Initially, any student who did not complete a data collection point was removed from the dataset to maintain consistency. For instance, if a student did not submit a solution for Exercise 2, all data associated with their other exercises and questionnaires were removed.

For outlier detection and removal, we applied the interquartile range (IQR) method, commonly used when the distribution of data deviates from normality [43]. Specifically, we calculated the first quartile (Q1) as well as the third quartile (Q3) for each key variable and determined the IQR as the difference between Q3 and Q1. Any data points falling below $Q1 - 1.5 \cdot IQR$ or above $Q3 + 1.5 \cdot IQR$ were flagged as outliers and subsequently removed. This approach helped reduce skew and ensure that the cleaned dataset accurately reflected typical student performance and feedback responses.

3.5 Data Analysis

To investigate the impact of each feedback type on student performance and experiences, we conducted a series of statistical analyses.

- **Descriptive Statistics:** For each feedback type, we calculated the mean, median, and standard deviation, providing a foundational understanding of central tendencies and variability within each group.
- **Normality Test:** To assess data distribution, we used the Shapiro–Wilk test [39], which evaluates whether the data follows a normal distribution. The results of this test indicated p-values well below the significance threshold ($p < 0.01$), confirming that the data deviates significantly from normality and validating our selection of non-parametric tests for subsequent analyses.
- **Kruskal-Wallis H-test:** Given the lack of normality and heterogeneous variances, we applied the Kruskal–Wallis test [27], a robust non-parametric test, to examine significant differences in correctness scores across feedback types. This test is particularly suited for comparing independent samples in non-normally distributed data.
- **Post-hoc Dunn's Test:** When the Kruskal–Wallis test indicated statistically significant differences, we conducted Dunn's test [13] with Bonferroni correction to perform pairwise comparisons and identify specific group differences. This post-hoc test determined which feedback types were statistically different from each other in terms of student performance.

To enhance interpretation, we visualized the data using box plots:

- **Box Plots:** Box plots were used to provide a clear visual summary of each feedback type's data distribution, including the median, quartiles, and outliers. This visualization facilitated comparison across feedback types, highlighting variations in performance and feedback effectiveness.

Overall, the combination of non-parametric statistical tests and visual analysis ensured a comprehensive examination of the data, enabling reliable insights into the effects of feedback types on student outcomes.

For the open-ended survey answers (DC4), we applied a simple thematic analysis. All free-text statements were collected and read in full. If a statement occurred only once, we still report. If multiple students made similar statements, we grouped them into themes and counted the number of times they appeared. Figure 10 in Section 5.2 therefore shows only descriptive statistics of preferences, while the narrative reports themes and frequencies derived from this analysis.

4 PARTICIPANTS

A total of 212 university students participated in the survey (74 participants from the first university, and 138 participants from the second university). Among them, 170 identified as male and 42 as

female, with ages ranging from 17 to 62 years (mean = 22.15 years, SD = 4.11 years, median = 21 years). Four participants did not specify their age. Regarding educational qualifications, the majority (186 participants) reported holding an A-level degree as their highest qualification. Other qualifications included a bachelor's degree (15 participants), a master's degree (5 participants), and a completed apprenticeship (2 participants). Additionally, four participants indicated holding other unspecified qualifications.

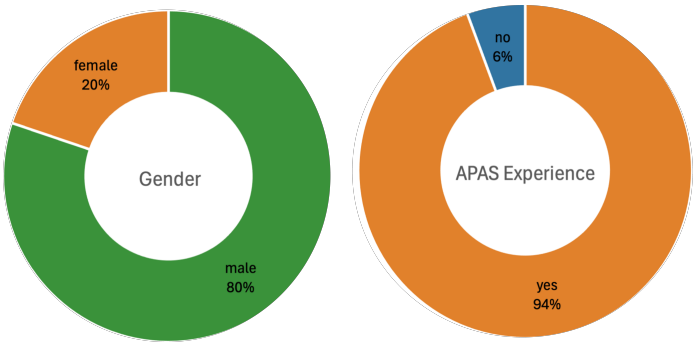

Figure 7: Overview of Students' Gender Distribution and Experience with APASs

More than half of the participants (118 students) were enrolled in bachelor's programs in computer science, while 54 students were enrolled in the bachelor's program in business informatics. One participant reported being in a master's program in business informatics. Other fields of study included business economics (three students) and statistics and data science (14 students). 22 participants were enrolled in other programs, 12 of whom stated they were pursuing an extension program in computing alongside their main program. Three were simultaneously pursuing a bachelor's degree in computer science and another bachelor's degree, while one participant was training as a student teacher in computing. Two participants were studying economics, and the remaining four were enrolled in medicine, mechatronics, psychology, and an extraordinary study program, respectively.

At the time of the study, the majority of participants were in their first or second semester, with 69 students in their first semester and 98 in their second. The remaining participants were distributed across various semesters: seven in the third semester, 14 in the fourth semester, three in the fifth semester, and 12 in the sixth semester. Additionally, nine participants were in higher semesters, including three in the seventh semester and six in the eighth semester. Students were also asked how many times they had attended the introductory course in programming already. Of the 212 participants, 189 (89%) were attending the course for the first time, 20 (10%) for the second time, and 3 (1%) for the third time. Programming experience among participants ranged from none (0 years) to 11 years (mean = 2.10 years, SD = 2.30 years, median = 1 year), with five students not specifying their level of experience.

When asked about prior familiarity with programming languages, the most commonly cited language was C/C++ (143 respondents), followed by Java (87 respondents), Python (80 respondents), and SQL or database systems (72 respondents). C# and JavaScript/Angular were also frequently mentioned, with 42 and 46 respondents, respectively. Notably, 36 participants indicated having no experience with programming languages, while 44 reported proficiency in other languages, collectively naming 25 additional languages in 77 responses. Regarding familiarity with the APAS used in the study, most participants (200 students, 94%) reported prior experience with the system. Only 12 participants (6%) indicated that they had not used the system before the experiment (cf. Fig. 7). Fig. 8 provides an overview of the different fields of study and programming languages in which students have indicated experience.

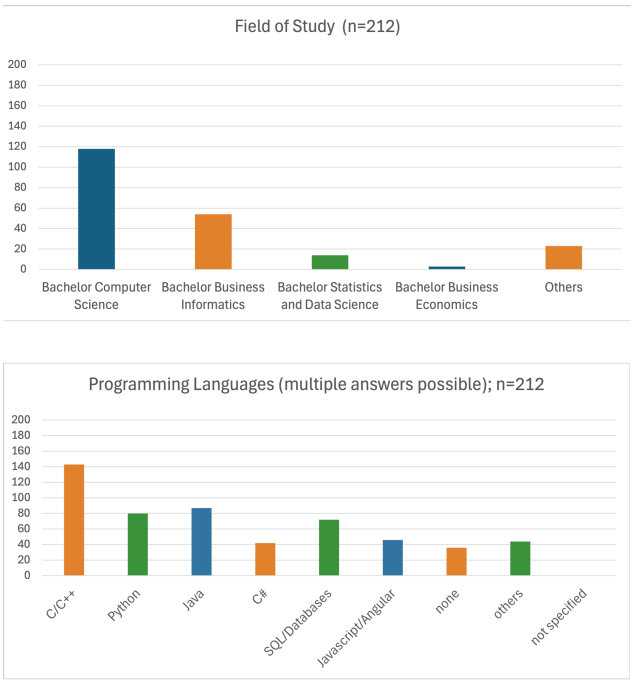

Figure 8: Students' experienced programming languages and their current study programs.

5 PART 1: STUDENT PERCEPTIONS OF FEEDBACK TYPES

This section presents findings pertaining to **RQ1**, which explores students' perceptions of different feedback types regarding quality and usefulness (RQ1.1), and overall perceived utility (RQ1.2).

5.1 TAM – Quality and Usefulness of Feedback Types

Fig. 9 shows the comparative scores for quality and usefulness of feedback types as rated by students, illustrating the distribution of scores for AI, Compiler, and Unit Test Feedback. This data is based on analyzing results from the three TAM questionnaires (DC3) handed out after each exercise. For each of the three constructs (PU, EU, OQ), we analyzed the results for each of the three feedback types.

Statistical Analysis: The reliability of the scale was assessed using Cronbach's α, Guttman's λ_6, and average inter-item correlation. Cronbach's α was approximately 0.93, and Guttman's λ_6 was 0.97, indicating excellent internal consistency. The average inter-item correlation was 0.34, suggesting good internal consistency without excessive redundancy. The 95% confidence intervals for the reliability coefficient were narrow and above 0.90, further reinforcing the reliability of the scale. For all items, removing any one does not change the overall alpha significantly. Most of the values remain at approximately 0.92, which indicates that each item is contributing positively to the scale's reliability.

TAM Results: For AI Feedback, the mean (μ) is 4.2, the median (M) is 4.5, and the standard deviation (σ) is 1.9. For Compiler Feedback, the mean (μ) is 3.4, the median (M) is 3.3, and the standard deviation (σ) is 1.8. For Unit Test Feedback, the mean (μ) is 4.8, the median (M) is 5.0, and the standard deviation (σ) is 1.7.

The comparison of means indicates that Unit Test Feedback was rated the highest, followed by AI Feedback, with Compiler Feedback receiving the lowest ratings. This suggests that students found Unit Test Feedback most useful and consistent, while AI Feedback had more variability in responses.

For all three examined TAM constructs, i.e., Perceived Usefulness, Perceived Ease of Use, and Output Quality, across the three feedback types AI, Compiler, and Unit Test, the Shapiro-Wilk test p-values are well below the significance level of 0.001, indicating that the data for each construct does not follow a normal distribution. As a result, the Kruskal-Wallis test, a non-parametric alternative to ANOVA, was used for further comparison between feedback types.

The Kruskal-Wallis test ($\chi^2 = 535.29$, $p < .001$) showed statistically significant differences between the three feedback types. Pairwise comparisons using Dunn's test with Bonferroni correction revealed the following results seen in Table 1:

Comparison	p-adj	Significant
AI Feedb. vs. Compiler Feedb.	< .001	Yes
AI Feedb. vs. Unit Test Feedb.	< .001	Yes
Compiler Feedb. vs. Unit Test Feedb.	< .001	Yes

Table 1: Post-hoc Dunn's test results (p-adj) for the quality and usefulness scores by feedback type.

The results indicate significant differences between all pairs of feedback types. AI Feedback was rated significantly higher than Compiler Feedback but lower than Unit Test Feedback. Compiler Feedback received the lowest ratings.

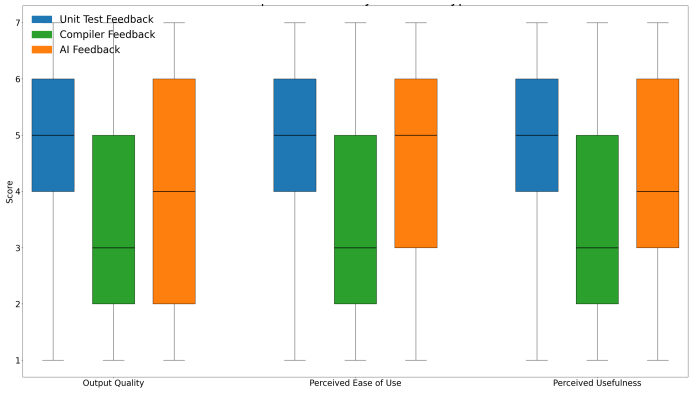

Figure 9: Comparative Scores for Quality and Usefulness by Feedback Type

5.2 Perceived Utility of Feedback Types

In addition to results obtained from the TAM questionnaires, we collected additional feedback using the post-survey questionnaire (DC4), to gain additional insights into, for example, why students deemed a feedback type particularly helpful and useful, or why not. A total of 85 of the 212 students who initially participated in the experiment also participated in the post-survey questionnaire. The survey asked students about their preferred feedback form. As illustrated in Fig. 10, 56% of participants favored the Unit Test Feedback, followed by 26% that favored AI Feedback, while Compiler Feedback, 18%, was the least preferred.

We further asked the students to provide feedback regarding the different feedback forms in the experiment. In the following, we summarize the students' responses. Ten students across the universities found the AI Feedback not helpful, with issues such as providing incorrect or hallucinated error messages. AI Feedback was initially helpful for most students, but became less useful when it provided confusing or incorrect suggestions once the code

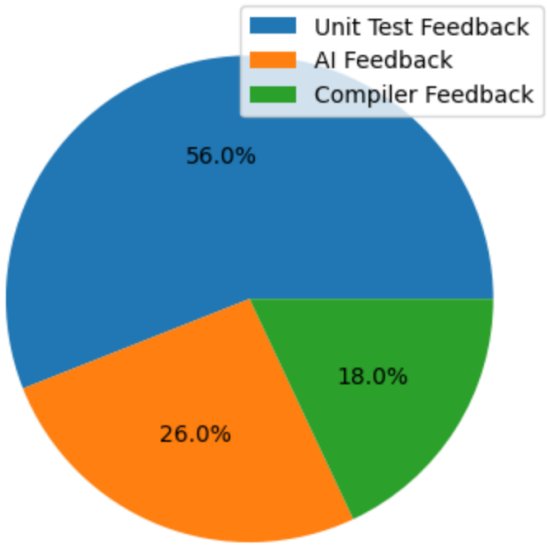

Figure 10: Best Feedback Type According to Students

was mostly correct. Two students found AI Feedback beneficial for optimization suggestions and learning new implementation techniques. Students also liked its ability to adapt to the code and provide personalized feedback. One student also noted that AI Feedback can sometimes provide quicker solutions, but it may reduce critical thinking by offering step-by-step solutions to students. Suggestions for improving the AI Feedback in the future included improving the clarity of the feedback, providing more concrete responses, and the ability to ask questions directly.

Six students found the Compiler Feedback helpful for identifying and correcting trivial errors, while four found it confusing and not helpful. In general, Compiler Feedback was valued for identifying and correcting simple errors. It was less helpful for more complex issues due to unclear messages. Compiler Feedback often caused more confusion than assistance, with five students having problems understanding and fixing errors within the given timeframe. For three students, Compiler Feedback was straightforward and aligned well with basic debugging practices.

Unit tests were generally seen as the most effective form of feedback for achieving correct results. Students liked the correctness verification provided by unit tests, though some found the feedback impractical under time constraints. Students found unit tests easier to understand and more helpful in identifying specific issues. They reported that unit tests provided clear feedback on what was expected versus what was produced, also highlighting exactly where problems occurred and supporting error detection and correction. Many students were familiar with unit tests from previous experiences, and they found them efficient for debugging. The students noted that unit tests allow for flexibility in finding solutions without imposing specific methods, fostering creativity and independent problem-solving.

In the survey, we also asked the students to report on other forms of feedback they would find helpful. Students requested that the compiler should further provide immediate feedback directly to save time, rather than having to wait for a complete evaluation from the Artemis environment.

It was also mentioned that the Artemis programming environment should use visual aids such as red underlines to highlight errors in the code, similar to other editors like Visual Studio Code. A combination of feedback forms was seen as the most helpful by several students. They suggested a combination of Unit Tests and AI Feedback. Also, AI Feedback should stop providing suggestions once the code is correct to prevent unnecessary changes. A combination of Unit Tests and Compiler Feedback was regarded as helpful as well, to see both what is expected for submission and to enable print statement-based debugging for personalized outputs.

In the survey, we also asked the students to give general feedback regarding the experiment setup. In general, the students noted that they did not encounter major problems during the experiment. The Artemis programming environment was criticized for being slow by some students. A recurring topic across the universities was the need for more time to engage more deeply with the different feedback mechanisms.

Main Findings for RQ1:

- Unit Test Feedback received the highest ratings for quality and usefulness.
- AI Feedback was rated higher than Compiler Feedback but lower than Unit Test Feedback.
- Compiler Feedback, while helpful for trivial errors, was often unclear for more complex issues.
- Students preferred Unit Test Feedback for clarity and effectiveness, and some suggested combining it with AI Feedback for optimal results.
- AI Feedback provided useful optimization tips and personalized guidance, but occasionally gave incorrect or confusing suggestions.

6 PART 2: IMPACT OF FEEDBACK TYPES ON STUDENT PERFORMANCE

In this section, we present results from our quantitative analysis of the collected exercise data (DC2). Addressing **RQ2**, we focus on how different feedback forms influence students' performance (RQ2.1) and problem-solving strategies (RQ2.2).

6.1 Feedback Types and Student Performance

Fig. 11 provides an overview of the comparative final correctness scores per student by feedback type after running the extra set of unit tests on the students' solutions. The box plots illustrate the distribution of final correctness scores across different feedback types, providing insights into how each form of feedback influences the correctness of students' final submissions.

Statistical Analysis: The Shapiro-Wilk test for normality yielded a statistic of 0.824 with a p-value $< 1 \times 10^{-25}$, indicating that the data significantly deviates from normality. Therefore, non-parametric tests were used.

Score Results: For AI Feedback, the mean (μ) is 65.44, the median (M) is 85.70, and the standard deviation (σ) is 38.67. For Compiler Feedback, the mean (μ) is 41.41, the median (M) is 23.80, and the standard deviation (σ) is 40.45. For Unit Test Feedback, the mean (μ) is 57.52, the median (M) is 70.00, and the standard deviation (σ) is 37.77.

The comparison of means shows that AI Feedback has the highest mean score, followed by unit test support, and no support exhibits the lowest mean score. This suggests that students with AI Feedback tend to achieve higher correctness in their final submissions. To verify whether there is a significant difference in the final correctness scores depending on the feedback type, we performed the Kruskal-Wallis H-test and Dunn's post-hoc test.

Comparison	p-adj	Significant
AI Feedb. vs. Compiler Feedb.	< .001	Yes
AI Feedb. vs. Unit Test Feedb.	0.035	Yes
Compiler Feedb. vs. Unit Test Feedb.	< .001	Yes

Table 2: Post-hoc Dunn's test results for the final correctness scores by feedback type

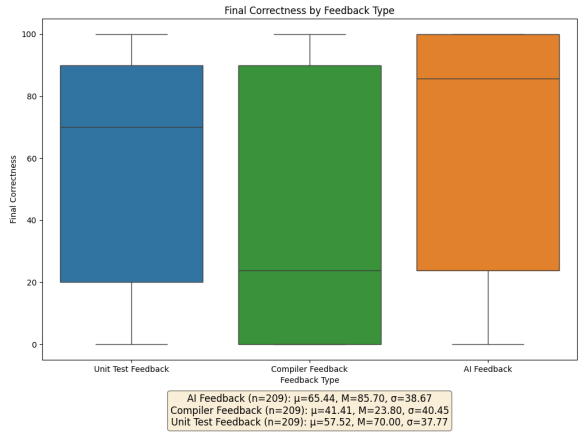

Figure 11: Comparative Final Correctness Scores by Feedback Type

6.2 Problem-Solving Approaches with Different Feedback Types

This subsection analyzes the effect of different feedback types on students' problem-solving strategies and methods, examining the number of submissions and the time lapsed between submissions.

Fig. 12 shows the comparative number of submissions per student, again grouped by the respective feedback type. Illustrating the distribution of the number of attempts made by students across different feedback types provides interesting insights into how each form of feedback influences the persistence and iteration behavior of students during programming tasks.

After applying IQR filtering, the data size was reduced slightly (from 640 to 624), ensuring the exclusion of extreme outliers. The Shapiro-Wilk test for normality yielded a statistic of 0.97 with a p-value $< 1 \times 10^{-10}$, indicating that the data significantly deviates from normality. Therefore, non-parametric tests were used.

For AI Feedback Support, the mean (μ) is 8.95, the median (M) is 8.00, and the standard deviation (σ) is 3.41.

For Compiler Feedback Support, the mean (μ) is 7.98, the median (M) is 8.00, and the standard deviation (σ) is 3.37.

For Unit Test Feedback Support, the mean (μ) is 8.08, the median (M) is 8.00, and the standard deviation (σ) is 3.37.

The comparison of means shows that the means of the three groups are close, with AI Feedback having the highest mean attempts, followed by unit test support, and no support having the lowest mean. This suggests that students with AI Feedback tend to iterate more on their solutions. The spread and outliers indicate that while there are individual differences in persistence, the overall impact of feedback type on the number of attempts is consistent. The medians suggest that the central tendency of attempts is similar across all support types, with students typically making around 8 attempts regardless of the support type.

The Kruskal-Wallis H-test ($H = 9.299$, $p = 0.009$) confirmed a statistically significant difference in the distributions of attempts across feedback types. Dunn's post-hoc test with Bonferroni correction was applied for pairwise comparisons (Table 3).

Comparison	p-adj	Significant
AI Feedb. vs. Compiler Feedb.	0.016	Yes
AI Feedb. vs. Unit Test Feedb.	0.042	Yes
Compiler Feedb. vs. Unit Test Feedb.	1.0000	No

Table 3: Post-hoc Dunn's test results for the number of attempts by feedback type

The results indicate significant differences in attempts between AI Feedback and both Compiler Feedback and Unit Test Feedback. The effect size (Eta Squared = 1.55) suggests that feedback type has a meaningful impact on students' iteration behavior. This suggests that students receiving AI Feedback use more attempts to solve an exercise compared to those receiving no support or unit test support. This finding is important as it implies that AI Feedback might encourage more iteration and exploration, potentially leading to a deeper understanding of the programming tasks.

In the following, we analyzed the effect of different feedback types on the time between submissions by students. After applying IQR filtering, the dataset size was reduced from 4199 to 3464, ensuring the exclusion of extreme outliers. The Shapiro-Wilk test for normality yielded a statistic of 0.766 with a p-value of 0.0, indicating that the time interval data significantly deviates from normality. Thus, non-parametric tests were applied. Fig. 13

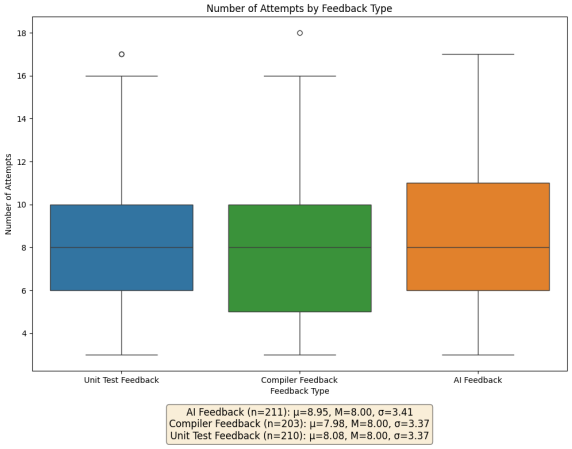

Figure 12: Comparative Number of Attempts per Student by feedback type

presents the comparative time intervals between submissions by feedback type.

For AI Feedback, the mean (μ) is 113.13 seconds, the median (M) is 79.80 seconds, and the standard deviation (σ) is 100.26 seconds.

For Compiler Feedback, the mean (μ) is 96.87 seconds, the median (M) is 64.29 seconds, and the standard deviation (σ) is 97.49 seconds.

For Unit Test Feedback, the mean (μ) is 94.68 seconds, the median (M) is 63.27 seconds, and the standard deviation (σ) is 93.36 seconds.

The comparison of means indicates that AI Feedback leads to the longest average time intervals between submissions, followed by no support, with unit test support resulting in the shortest intervals. To determine whether these differences are statistically significant, a Kruskal-Wallis H-test ($H = 49.17$, $p = 2.10 \times 10^{-11}$) was performed. Post-hoc pairwise comparisons using Dunn's test were conducted, and the results are presented in Table 4.

Comparison	p-adj	Significant
AI Feedb. vs. Compiler Feedb.	< .001	Yes
AI Feedb. vs. Unit Test Feedb.	< .001	Yes
Compiler Feedb. vs. Unit Test Feedb.	1.0000	No

Table 4: Post-hoc Dunn's test results for time intervals between submissions by feedback type.

The results indicate significant differences between AI Feedback and both Compiler Feedback and Unit Test Feedback. However, the difference between Compiler Feedback and Unit Test Feedback is not statistically significant. The effect size (Eta Squared = 8.19) suggests a substantial impact of feedback type on the time intervals.

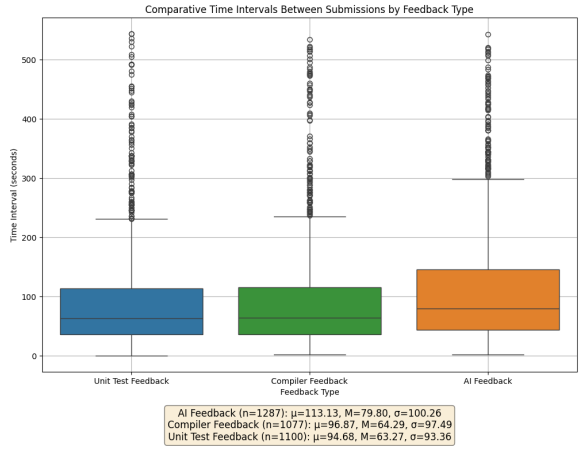

Figure 13: Comparative Time Intervals Between Submissions by feedback type

The analysis reveals that AI Feedback leads to longer and more variable intervals between submissions, suggesting that students take more time to iterate and refine their work. Compiler feedback results in shorter intervals, indicating quicker successive submissions. Unit Test Feedback falls in between, with variability comparable to AI Feedback. Fig. 14 shows the number of submissions by feedback type over time since the first submission. The line plots illustrate how the frequency of submissions decreases over time for each feedback type, providing insights into the submission patterns and persistence of students under different forms of feedback. Submissions with Compiler Feedback are initially more frequent but exhibit a steeper decline over time compared to the other feedback types. In contrast, AI Feedback shows a lower initial submission rate but maintains a relatively higher level across time, suggesting sustained usefulness.

Main Findings for RQ2:

- Students receiving AI Feedback achieved the highest correctness scores, significantly outperforming those with Unit Test Feedback or Compiler Feedback support.
- Unit Test Feedback also improved correctness scores compared to no support, though not as strongly as AI Feedback.
- AI Feedback led to students making more attempts and taking longer intervals between submissions.
- Compiler Feedback led to quicker successive submissions but lower correctness and less sustained problem-solving effort over time.

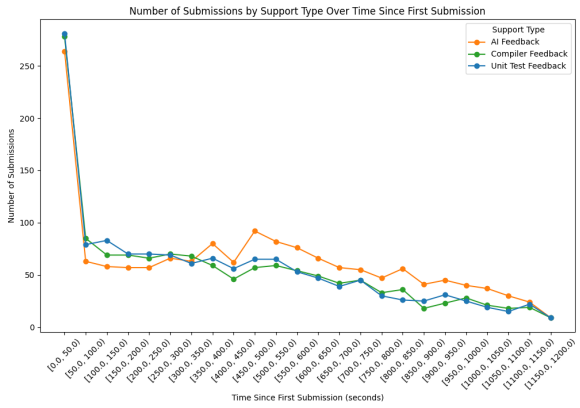

Figure 14: Number of Submissions by feedback type Over Time Since First Submission

7 DISCUSSION

The results of this study show that not all feedback forms are equally helpful to students. Traditional compiler feedback, while quick to point out simple syntax mistakes, often leaves students struggling when it comes to more complex issues. In

contrast, both unit tests and AI-generated feedback offered clearer guidance and supported students in producing more accurate solutions.

Familiarity with Unit Tests: An interesting outcome is the discrepancy between students' perceptions and their performances. Students expressed a strong preference for unit test feedback, describing it as clear, straightforward, and useful. However, those who received AI-generated feedback ultimately achieved the highest correctness scores. This suggests that while unit tests may feel more familiar and reassuring, AI Feedback encourages deeper engagement because students guided by AI hints took more time between submissions, paused longer before resubmitting, and revisited their code more frequently.

More Advanced AI Feedback Mechanisms: While unit tests provide stable and reliable verification of correctness, they may not inherently foster the depth of reasoning and concept mastery that AI Feedback can inspire. At the same time, although the adaptive and tutor-like guidance of AI Feedback can lead to higher correctness and more sustained problem-solving, it occasionally produces confusing or erroneous suggestions. These inconsistencies underline the importance of improving the reliability of AI-based feedback mechanisms.

As reflected in students' free-text reports in Section 5.2, occasional inaccuracies and hallucinations can erode trust even when overall performance improves. Since our setup relied on a fixed, non-interactive prompt yet still exhibited such cases, we view reliability and trust as central concerns for AI feedback in APASs. Building on recent work on adaptive and personalized feedback generation [6], more advanced mechanisms are needed to ensure consistency. Potential strategies include grounding responses in verifiable artifacts (e.g., failing unit tests or specific code locations), limiting output length and code generation to reduce error-prone verbosity, and enabling abstention when reliable hints cannot be generated (i.e., the system refrains from giving potentially misleading advice). Additionally, a second-order validation step, where an auxiliary LLM evaluates whether a response is appropriate and consistent with the task, could further filter out misleading outputs and strengthen student trust.

Beyond these technical safeguards, students themselves emphasized the value of interaction design. Several suggested that a more interactive, chatbot-like interface would allow them to ask follow-up questions and clarify uncertainties in real time, thereby further enhancing both the utility and the trustworthiness of AI feedback.

Combined Feedback Mechanisms: These findings highlight the value of a balanced feedback environment. Unit tests excel at verifying that students meet specific requirements and provide a sense of closure and direction. AI-generated hints, on the other hand, can offer context-sensitive, exploratory support that encourages students to reflect, adapt, and explore alternative solution paths. Integrating both approaches could leverage the clarity and verification of unit tests and the adaptive, personalized scaffolding of AI Feedback, resulting in richer opportunities for skill development and deeper conceptual understanding.

From a pedagogical perspective, integrating AI hints with unit tests may influence how instructors design courses and assignments. The presence of richer, more adaptable feedback can free educators from repetitive, individualized guidance and allow them to focus on higher-level interventions, curriculum refinement, and addressing common conceptual misunderstandings observed in aggregate. For students, especially novices, having both immediate correctness checks and guidance on how to think differently about their problems can accelerate the transition from simple debugging to thoughtful, strategic problem-solving.

Key Findings and Opportunities for Advancement: These results open multiple avenues for future exploration. Investigating how different subgroups of students, varying in their prior experience, confidence, or learning styles, respond to AI hints versus unit test feedback could help tailor feedback mechanisms to more individual needs. Additionally, exploring long-term impacts on learning outcomes, such as whether students who engage more deeply with AI Feedback subsequently perform better in advanced courses or in exams where there is no AI Feedback, would provide valuable insights.

In summary, these findings suggest that feedback in APASs can be improved significantly by combining unit tests for reliable checks with AI for dynamic, adaptive hints that nurture deeper engagement. Such a balanced environment holds promise for supporting students in becoming more confident, capable programmers who not only meet assignment requirements but also internalize essential problem-solving competencies. With ongoing refinements and careful consideration of trust, scalability, and infrastructure, these insights point toward more sophisticated and pedagogically effective feedback ecosystems that benefit both learners and educators.

8 LIMITATIONS AND THREATS TO VALIDITY

In line with the guidelines proposed by [47], we discuss our study's limitations across four primary categories: *Construct Validity*, *Internal Validity*, *External Validity*, and *Conclusion Validity*.

8.1 Construct Validity

Construct validity concerns the degree to which the study accurately captures and measures the constructs it aims to investigate, in our case, the impact of different feedback forms in APASs on student performance and engagement.

Choice of Exercises and Feedback Types. While the exercises and test cases used were carefully peer-reviewed to align with the core competencies of introductory programming courses taught at the two universities, they may not fully encompass the breadth of difficulties, topics, or problem-solving approaches found in all programming curricula. More diverse or context-rich exercises could produce different interaction patterns, particu-

larly with certain feedback types (e.g., AI-based hints). Similarly, our selection of feedback mechanisms may not represent the full range of feedback modalities available in all APASs, limiting the ability to generalize to other forms of guidance (e.g., video or peer feedback).

Operationalization of "AI-Guided Feedback". We used GPT-3.5-Turbo to provide on-demand suggestions and clarifications. Although this model generally produced high-quality output, the nature of AI-generated responses—ranging from concise hints to more elaborate explanations—can vary with prompt formulation and server load. Occasional ambiguity or inaccuracies in these responses could have affected students' trust and willingness to rely on AI Feedback, influencing both engagement and outcomes.

8.2 Internal Validity

Internal validity pertains to whether the observed effects (e.g., performance gains, time spent on tasks) are truly attributable to the different feedback forms rather than confounding factors.

Teaching Setting and Instructional Method. Although the experiment was designed with standardized guidelines, variations in how different instructors integrated and emphasized the APAS in their curricula could have introduced discrepancies in student motivation and preparedness. Furthermore, students received bonus points for participating in the experiment; this incentive may have triggered a self-selection bias, as students highly motivated by extra credit or more comfortable programming could be overrepresented. Despite these concerns, such incentives helped secure a sufficiently large sample size and are common practice in educational research.

Familiarity and Comfort with the APAS. Students entered the experiment with varying levels of prior experience using the APAS (Artemis). Those who had used it extensively may have spent less cognitive effort on interface navigation and more on solving the core programming challenges. Conversely, students with minimal exposure may have been disproportionately distracted by platform logistics. Although we required all participants to complete at least one practice exercise on Artemis before the experimental session, the depth and quality of that engagement could not be fully controlled.

Reliability of AI Feedback. The AI-generated feedback itself can also be viewed as a source of potential bias. Inconsistencies in the model's performance, influenced by prompt phrasing or model updates, may have affected the reliability of suggestions offered to different students at different times. This variation in guidance could alter both perceived and actual improvements in code quality or problem-solving strategy, thus confounding the internal validity of the comparison between feedback conditions.

Timing measures and message length. Longer intervals between submissions in the AI condition may reflect more content to read in AI responses rather than deeper processing. Our unit tests were defined rigorously with detailed failure messages, which should narrow any gap in reading time relative to AI outputs. However, we did not statistically analyze the length or verbosity of AI responses, nor did we normalize timing by message length. Accordingly, the timing results are correlational and should be interpreted with caution.

8.3 External Validity

External validity addresses the generalizability of the findings to other settings, populations, and APASs.

Representativeness of the Student Sample. While the experiment was conducted across multiple universities, the sample primarily consisted of students enrolled in introductory programming courses. Differences in educational backgrounds, learning styles, and personal motivations still exist within this population, but additional variety, such as professional developers or non-computer-science majors, could yield different insights. Moreover, only a small fraction of participants reported extensive prior programming experience, and this subgroup was too small and heterogeneous for meaningful analysis. Future studies involving more diverse learner groups may reveal more nuanced effects of AI-driven and traditional feedback.

Representativeness of Artemis as an APAS. Artemis shares many features common to modern APASs (e.g., automated grading, iterative feedback, analytics dashboards). However, variations in interface design, integration with external development tools, and customizable feedback pipelines mean that not all APASs function exactly like Artemis. Consequently, our observations, while indicative of broader trends, may not transfer perfectly to platforms with different architectures, user interfaces, or pedagogical philosophies.

8.4 Conclusion Validity

Conclusion validity pertains to the degree of confidence we can have in the relationship between the treatments (different feedback forms) and the observed outcomes (student performance, engagement metrics).

Measurement of Student Engagement and Performance. Our primary metrics included time between submissions, total number of submissions, and final solution correctness. While these indicators provide valuable quantitative insights, they do not fully capture the qualitative dimensions of engagement—such as depth of understanding or the specific cognitive processes students employ. The increased time between submissions in the AI-guided group, for instance, might partly reflect reading and interpreting AI Feedback rather than deeper conceptual engagement.

Interpretation of Observed Patterns. Although significant differences emerged between AI-driven and traditional feedback, caution is warranted in attributing improvements solely to the type of feedback. Other unmeasured variables, such as problem difficulty or a student's prior knowledge of the specific topic, may have influenced outcomes. Future studies employing additional qualitative measures (e.g., think-aloud protocols and interviews) could help clarify the causal pathways behind these observed patterns.

9 CONCLUSION & FUTURE WORK

In this study, we present a detailed examination of feedback mechanisms in automated programming assessment systems, focusing on compiler-based, unit test-based, and AI-driven feedback. The results highlight the importance of tailored feedback in enhancing student engagement, improving problem-solving strategies, and fostering deeper learning outcomes.

Unit test feedback was perceived as the most reliable and familiar by students, aligning closely with their expectations for correctness verification. However, it was the AI-driven feedback that led to the highest correctness scores and encouraged more iterative and thoughtful problem-solving. Compiler feedback, while effective for simple syntax and runtime errors, showed limited utility for addressing complex programming challenges, which negatively affected both student performance and engagement.

The findings suggest that combining unit test and AI Feedback can offer significant pedagogical advantages. Unit tests provide clear, objective correctness checks, while AI Feedback delivers adaptive, context-aware guidance. This combination can support students in developing not only technical accuracy but also critical thinking and more sophisticated skills. Implementing a hybrid feedback mechanism in APASs could create a more supportive learning environment, particularly for novice programmers, by addressing diverse needs and learning styles.

The results also point to key challenges and opportunities for further research. For instance, while AI Feedback demonstrated the potential for fostering deeper engagement, inconsistencies and occasional hallucinations in suggestions highlight the need for improved reliability. Integrating real-time error detection and chatbot-like interaction capabilities would enhance the usability and trustworthiness of AI-driven feedback [17, 6]. Additionally, further exploration is needed to tailor feedback to different student cohorts, such as those with varying levels of prior programming experience or confidence.

As part of our ongoing and future research, we plan to further investigate the long-term effects of combined feedback mechanisms on programming education. Evaluating how students who use such systems perform in advanced courses or real-world programming tasks could provide valuable insights into the actual impact of these feedback approaches throughout subsequent programming courses. Moreover, analyzing how different subgroups, such as students from diverse academic backgrounds, benefit from these feedback types may help refine and optimize APAS designs. Future work should also evaluate a more interactive, chat-based AI tutor that allows personalized follow-ups and measure its effects on engagement and performance. Another interesting direction for future work is to investigate how more extensive prior programming experience influences students' preferences for different feedback types. Although only a very small share of our participants reported substantial prior experience, future studies with more heterogeneous cohorts could provide deeper insights. This may also open possibilities for adapting AI tutor responses to students' varying skill levels and backgrounds, thereby offering more tailored support.

From a technical perspective, enhancing the scalability and cost-efficiency of AI Feedback systems will be crucial for their broader adoption. As large language models continue to evolve, optimizing their deployment within educational contexts will require balancing performance, resource consumption, and accessibility. Collaborative efforts between academia, industry, and educational institutions will play a pivotal role in addressing these challenges.

REFERENCES

[1] U. Z. Ahmed, R. Sindhgatta, N. Srivastava, and A. Karkare. Targeted example generation for compilation errors. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering*, page 327–338. IEEE Press, 2020.

[2] U. Z. Ahmed, N. Srivastava, R. Sindhgatta, and A. Karkare. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. ICSE-SEET '20, page 139–150, New York, NY, USA, Sept. 2020. ACM.

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

[4] J. Alemán. Automated assessment in a programming tools course. *IEEE Transactions on Education*, 54:576–581, 2011.

[5] D. K. Barrow, A. Mitrovic, S. Ohlsson, and M. Grimley. Assessing the impact of positive feedback in constraint-based tutors. In *Proc. of the International Conference on Intelligent Tutoring Systems*, pages 250–259. Springer, 2008.

[6] P. Bassner, E. Frankford, and S. Krusche. Iris: An ai-driven virtual tutor for computer science education. In *Proc. of the 2024 Annual Conference on Innovation and Technology in Computer Science Education*, page 394–400, New York, NY, USA, 2024. ACM.

[7] A. Bey, P. Jermann, and P. Dillenbourg. An empirical study comparing two automatic graders for programming. moocs context. In E. Lavoue, H. Drachsler, K. Verbert, J. Broisin, and M. Perez-Sanagustín, editors, *Data Driven Approaches in Digital Education*, page 537–540, Cham, 2017. Springer International Publishing.

[8] S. Bloxham, B. Den-Outer, J. Hudson, and M. Price. Let's stop the pretence of consistent marking: exploring the multiple limitations of assessment criteria. *Assessment & Evaluation in Higher Education*, 41(3):466–481, 2016.

[9] A. P. Cavalcanti, A. Barbosa, R. Carvalho, F. Freitas, Y.-S. Tsai, D. Gašević, and R. F. Mello. Automatic feedback in online learning environments: A systematic literature review. *Computers and Education: Artificial Intelligence*, 2:100027, 2021.

[10] H.-M. Chen, B.-A. Nguyen, Y.-X. Yan, and C.-R. Dow. Analysis of learning behavior in an automated programming assessment environment: A code quality perspective. *IEEE Access*, 8:167341–167354, 2020.

[11] B. S. Clegg, P. McMinn, and G. Fraser. The influence of test suite properties on automated grading of programming exercises. In *Proc. of the 32nd Conference on Software Engineering Education and Training*, page 1–10, Nov. 2020.

[12] F. M. V. der Kleij, R. Feskens, and T. Eggen. Effects of feedback in a computer-based learning environment on students' learning outcomes. *Review of Educational Research*, 85:475 – 511, 2013.

[13] O. J. Dunn. Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252, 1964.

[14] S. H. Edwards and M. A. Perez-Quinones. Web-cat: automatically grading programming assignments. In *Proc. of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 328–328, 2008.

[15] J. English and T. English. Experiences of using automated assessment in computer science courses. *Journal of Information Technology Education. Innovations in Practice*, 14:237, 2015.

[16] J. C. Farah, S. Ingram, and D. Gillet. Supporting developers in creating web apps for education via an app development framework. In *Proc. of the 8th International Conference on Higher Education Advances*, pages 883–890. Editorial Universitat Politècnica de València, 2022.

[17] E. Frankford, C. Sauerwein, P. Bassner, S. Krusche, and R. Breu. Ai-tutoring in software engineering education. In *Proc. of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, page 309–319, New York, NY, USA, 2024. ACM.

[18] GitHub. Github copilot ai editor (https://github.com/features/copilot), 2024.

[19] P. J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proc. of the 44th ACM Technical Symposium on Computer Science Education*, pages 579–584, 2013.

[20] C. Henderson, E. Yerushalmi, V. H. Kuo, P. Heller, and K. Heller. Grading student problem solutions: The challenge of sending a consistent message. *American Journal of Physics*, 72(2):164–169, 2004.

[21] X. Hou, Z. Ren, J. Wang, W. Cheng, Y. Ren, K.-C. Chen, and H. Zhang. Reliable computation offloading for edge-computing-enabled software-defined iov. *IEEE Internet of Things Journal*, 7(8):7097–7111, 2020.

[22] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proc. of the 10th Koli Calling International Conference on Computing Education Research*, page 86–93, New York, NY, USA, 2010. ACM.

[23] H. Keuning, J. Jeuring, and B. Heeren. A systematic literature review of automated feedback generation for programming exercises. *Proc. of the ACM Transactions on Computing Education*, 19(1):1–43, 2018.

[24] E. Kochmar, D. D. Vu, R. Belfer, V. Gupta, I. Serban, and J. Pineau. Automated personalized feedback improves learning gains in an intelligent tutoring system. *Artificial Intelligence in Education*, 12164:140 – 146, 2020.

[25] S. Krusche and A. Seitz. Artemis: An automatic assessment management system for interactive learning. In *Proc. of the 49th ACM Technical Symposium on Computer Science Education*, pages 284–289, 2018.

[26] S. Krusche, N. von Frankenberg, L. M. Reimer, and B. Bruegge. An interactive learning method to engage students in modeling. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, pages 12–22, 2020.

[27] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[28] R. Lobb and J. Harlow. Coderunner. *ACM Inroads*, 7(1):47–51, 2016.

[29] V. J. Marin, T. Pereira, S. Sridharan, and C. R. Rivero. Automated personalized feedback in introductory java programming moocs. In *Proc. of the 33rd International Conference on Data Engineering*, page 1259–1270, Apr. 2017.

[30] I. Mekterović, L. Brkić, B. Milašinović, and M. Baranović. Building a comprehensive automated programming assessment system. *IEEE Access*, 8:81154–81172, 2020.

[31] M. Messer, N. C. C. Brown, M. Kölling, and M. Shi. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 24(1), Feb. 2024.

[32] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.

[33] J. A. Noshin and S. I. Ahmed. Teaching programming to non-programmers at undergraduate level. *International Journal of Engineering and Management Research*, 8(3):191–194, 2018.

[34] J. C. Paiva, J. P. Leal, and A. Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.*, 22(3), jun 2022.

[35] V. Pieterse. Automated assessment of programming assignments. *CSERC*, 13:4–5, 2013.

[36] R. Plösch, I. Groher, and A. Hofer. Competence-based assessment of programming assignments. In *Proc. of the 36th International Conference on Software Engineering Education and Training*, pages 1–10, 2024.

[37] G. Polito and M. Temperini. A gamified web based system for computer programming learning. *Computers and Education: Artificial Intelligence*, 2:100029, 2021.

[38] Y. Qian and J. Lehman. Using targeted feedback to address common student misconceptions in introductory programming: A data-driven approach. *SAGE Open*, 9, 2019.

[39] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 15–26, New York, NY, USA, June 2013. ACM.

[41] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel. Codeocean-a versatile platform for practical programming excercises in online environments. In *Proc. of the 2016 IEEE Global Engineering Education Conference*, pages 314–323. IEEE, 2016.

[42] S. Truniger. Codeboard: Verbesserung und erweiterung der automatisierten hilfestellungen. In *ZHAW Zürcher Hochschule für Angewandte Wissenschaften*, 2023.

[43] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[44] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, 26(6):2328–2341, 2018.

[45] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, 2002.

[46] G. Vial and B. Negoita. Teaching programming to non-programmers: the case of python and jupyter notebooks. *Proc. of the International Conference on Information Systems*, 2018.

[47] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[48] Y.-X. Yan, J.-P. Wu, B.-A. Nguyen, and H.-M. Chen. The impact of iterative assessment system on programming learning behavior. In *Proc. of the 9th International Conference on Educational and Information Technology*, page 89–94, New York, NY, USA, Apr. 2020. ACM.

[49] F. Zampirolli, J. M. B. Josko, M. F. Venero, G. Kobayashi, F. Fraga, D. Goya, and H. R. Savegnago. An experience of automated assessment in a large-scale introduction programming course. *Computer Applications in Engineering Education*, 29:1284 – 1299, 2021.

[50] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018. Proc. of the 3rd International Conference on Computer Science and Computational Intelligence: Empowering Smart Technology in Digital Era for a Better Life.