From Repositories to Practice: LLM-Based Personalization in Programming Education

Marek Horváth
Department of Computers and Informatics, Technical University of Košice, Slovakia

Michaela Ďurkovičová
Department of Computers and Informatics, Technical University of Košice, Slovakia

Lenka Bubeňková
Department of Computers and Informatics, Technical University of Košice, Slovakia

Emília Pietriková
Department of Computers and Informatics, Technical University of Košice, Slovakia

Abstract

Many programming education systems personalize practice mainly from test results, progress indicators, or explicitly selected user parameters. This paper instead uses students' existing source-code repositories as evidence for personalized programming practice. It presents the design and implementation of a web application that connects to a university version control system, analyzes student-selected repositories, and generates programming challenges targeted at weaknesses identified in the submitted source code. The prototype analyzes the current state of the selected repositories using LLM-based assessment of code-quality indicators such as readability, modularity, duplication, input handling, edge-case handling, and algorithmic complexity. Students can also generate parameterized tests, submit solutions to programming challenges, and receive an automated score with textual feedback. The system stores student activity, test history, challenge history, achieved scores, and generated feedback as an operational student profile shown to students and teachers. The prototype was evaluated with 11 students at the Technical University of Košice. The evaluation demonstrates technical feasibility, usability, and functional operation of the prototype, but it does not measure learning gains, retention, or improvement in programming performance.

2012 ACM Subject Classification Applied computing → Computer-assisted instruction; Applied computing → Interactive learning environments; Software and its engineering → Software creation and management

Keywords and phrases Programming Education, Personalized Programming Practice, Student Profiles, Formative Assessment, Automated Feedback, Learning Analytics, Software Engineering Education

Digital Object Identifier 10.4230/OASIcs.ICPEC.2026.9

Funding This work was supported by project KEGA No. 061TUKE-4/2025 "Building Bridges between University and High School ICT Education".

1 Introduction

The acquisition of programming skills in software engineering is widely recognized as a complex and cognitively demanding process [5]. Students often struggle not only with understanding theoretical concepts but also with applying them in practical contexts, which requires problem-solving abilities, analytical thinking, and systematic work with source code [23]. In addition, learners differ in their prior knowledge, experience, and learning pace, which can affect how well traditional teaching approaches meet their needs. These factors motivate systems that personalize practice using evidence from students' own programming artifacts.

Despite the rapid growth of e-learning platforms, many systems still provide largely uniform learning content or personalize practice mainly from basic progress information, test results, or user-selected parameters [19]. This limitation is particularly relevant in programming education, where students benefit from feedback that refers to their actual solutions, design choices, and recurring coding practices. Automated feedback for programming exercises has therefore become an important research direction, but feedback quality, specificity, and pedagogical reliability remain open challenges [14].

Recent advances in artificial intelligence, especially large language models (LLMs), have introduced new possibilities for generating programming tasks, explaining errors, and supporting students during practice [6, 13, 15]. At the same time, the use of generative AI in education raises concerns about reliability, transparency, over-reliance, and the need for human oversight [22]. Systems that use LLMs for educational feedback must therefore distinguish between technical feasibility and validated learning impact.

A useful source of information for personalization is the student's source code stored in a version control system. Repository-level code can provide evidence about programming habits that are not captured by ordinary multiple-choice tests, such as decomposition, naming, input validation, defensive programming, and repeated design patterns. In the current implementation presented here, the system analyzes the final or current state of the repositories selected by the student; it does not analyze commit history or infer progress over time from repository evolution.

The main objective of this work is to design and implement a web-based prototype for personalized programming practice that integrates repository-level source-code analysis with AI-generated programming challenges and automated textual feedback. In this paper, personalization and adaptivity refer specifically to repository-analysis-based task generation: the system selects a target weakness from the analyzed repositories and generates a corresponding programming challenge. This should not be interpreted as a validated adaptive tutoring model. The contribution of the paper is a system that combines: integration with a university version control system; student-controlled selection of repositories for analysis; LLM-based assessment of repository-level code-quality indicators; identification of weaker programming practices; personalized programming challenge generation; automated scoring with textual feedback; parameterized test generation; and storage of student activity, test history, challenge history, scores, and generated feedback.

The system was implemented as a functional web application and evaluated through prototype user testing with students at the Technical University of Košice. The evaluation focuses on usability, task completion, and perceived acceptance. It should not be interpreted as evidence that the system improves programming ability or produces learning gains.

2 Related Work

Programming education research has long examined the difficulties students face when moving from conceptual knowledge to practical code construction. Introductory programming requires syntax knowledge, problem decomposition, debugging, testing, and the ability to reason about program behavior [5]. Teaching reforms that increase practice and student engagement can improve the learning environment [23], but they do not by themselves solve the need for individualized feedback on students' own code. Automated assessment and feedback systems are therefore widely used in programming courses. Ala-Mutka [1] surveys automated assessment approaches for programming assignments, including static and dynamic assessment methods, while Keuning et al. [14] show that automated feedback can

take many forms, including test-based feedback, hints, style feedback, and repair suggestions. Recent work has also adapted automated assessment infrastructure for large programming courses [8] and benchmarked AI models for grading computer science assignments across multiple domains [11]. The present work follows this line of research but focuses on using repository-level evidence to select a personalized practice task rather than only grading a single submitted exercise.

Adaptive learning systems and intelligent tutoring systems provide a broader foundation for personalization. Brusilovsky and Peylo [4] describe adaptive and intelligent web-based educational systems that rely on learner information to personalize content and navigation. VanLehn [21] discusses the effectiveness of tutoring systems and highlights the importance of step-level support and feedback. The prototype described in this paper is related to these systems, but it does not claim to implement a psychometrically validated learner model. Its current student model is an operational profile used to store and display activity, tests, challenges, scores, and feedback. Learning analytics and educational data mining in programming provide methods for analyzing programming behavior and student artifacts. Ihantola et al. [12] review the use of programming process data in computing education, while Espana-Boquera et al. [7] show how data collected from programming environments can support analysis of learning processes. Static code analysis has also been studied as a basis for programming assessment and feedback. Nutbrown and Higgins [17] show that static analysis can support automated assessment, but also warn that simplistic implementations can diverge from human assessment. Static code analysis has also been studied as a basis for personalized learning analytics in computer science education [10]. Related source-code analysis work includes comparison of code-similarity tools on large sets of student projects [9], which is relevant to future duplicate detection and repository-level analytics. Such work motivates the use of programming artifacts as a data source, but the present prototype currently analyzes repository snapshots rather than fine-grained event logs or commit histories.

Generative AI has recently become a major topic in programming education. Studies of AI code-generation tools show both potential benefits and risks for novice learners, including effects on task completion, prompting behavior, over-reliance, and the need for guardrails [6, 13, 15]. Experience reports from CS1 and CS50 describe course-level integration of LLM-based tools for code explanation, testing support, style feedback, and guided assistance [20, 16]. Other work discusses AI-powered programming education, changing instructional strategies, and student use of ChatGPT during programming tasks [18, 2]. The system presented here uses LLMs not as an unrestricted code-generation assistant, but as a component for source-code analysis, challenge generation, and textual feedback within a constrained workflow. Unlike LLM assistants embedded directly into the same programming task, this system uses previously submitted repository artifacts to derive the target of subsequent practice. Against this background, the contribution is not a claim of general educational effectiveness, but a prototype architecture that connects version-control integration, repository-level analysis, personalized task generation, and activity history in one system.

3 Methodology

This section describes the design and implementation of the proposed personalized learning system. It clarifies the data processed by the prototype, the code-quality indicators used for analysis, the adaptive logic used to generate programming challenges, and the way student activity is stored. The description is intentionally limited to the behavior implemented in the prototype.

3.1 System Overview

The proposed system is a web-based platform for programming education. It combines instructor-managed learning materials and tests with student-initiated personalized programming challenges. The end-to-end workflow is as follows. First, the student connects the application account to the university version control system using an access token. Second, the application lists repositories available to the student, and the student selects which repositories should be included in the analysis. Third, the system retrieves and analyzes the whole current version of the selected repository or repositories, including source files and the repository structure relevant to the submitted coursework. Fourth, the LLM-based analysis identifies weaker aspects of the student's code from a set of code-quality indicators. Fifth, the system generates a personalized programming challenge targeted at one of the identified weaknesses. Finally, the student submits a solution, and the system returns an automated score and textual feedback.

The analysis is language-agnostic in principle because the LLM interprets source code as input text rather than relying on a compiler for one specific language. However, the prototype and evaluation context mainly targeted C and Java, since these languages are commonly used in programming courses at the Technical University of Košice. The current implementation does not compile, execute, or unit-test the repositories during the analysis phase. As a result, it can process incomplete or syntactically incorrect code as source-code input, but this does not guarantee that the LLM assessment of such code is correct.

Personalization can be initiated even from a single selected repository. A single repository can provide enough evidence for generating an initial personalized challenge, especially when it contains a complete course assignment. It should not be interpreted as sufficient for a robust long-term student model. More reliable long-term personalization would require additional data, repeated interactions, and validated modeling assumptions.

3.2 Operational Student Profile

The system stores student activity, completed tests, generated tests, programming challenge history, submitted solutions, achieved scores, and generated feedback. These data are displayed to the student and teacher so that both can review completed activity and previous results. In the current version, this profile is operational: it supports tracking and presentation of activity inside the system. It is not a psychometrically validated learner model, and the stored history is not yet used by a validated adaptive algorithm for continuous personalization.

3.3 Data Collection

The primary data source used in the personalized challenge workflow is the source code contained in repositories selected by the student from the university version control system. The student explicitly chooses which repositories are included. The system then analyzes the whole selected repository or repositories rather than a single isolated file.

To enable data access, the web application integrates with the university version control system using secure authentication based on access tokens. The application retrieves repository content while respecting the permissions of the connected account. The current implementation analyzes the final or current state of the selected repositories. It does not analyze commit history, commit messages, branching behavior, or the temporal evolution of the student's work.

Before LLM analysis, the prototype prepares a textual representation of the selected repository content. The intended input consists of source-code files and a lightweight repository structure relevant to coursework; for the evaluated context this mainly means

C and Java source files, such as `.c`, `.h`, and `.java` files. Binary files, images, archives, dependency directories, generated build outputs, and other clearly non-source artifacts are not intended to be analyzed by the LLM. Context-size management is limited in the current version. If selected repositories exceed the model context limit, the system does not yet implement a validated chunking, sampling, or aggregation strategy; the input must be reduced to content that fits the prompt, which may make the resulting analysis incomplete.

3.4 Source Code Analysis

The selected repositories are analyzed using code-quality indicators that reflect programming practices relevant to introductory and intermediate programming courses. The indicators are summarized in Table 1. The analysis is LLM-based: the model receives the source-code context and structured instructions asking it to assess these indicators and identify weaker areas. The current version does not define a formally validated weighting scheme for the indicators and does not define empirically validated thresholds for weak, medium, or strong performance levels. Weaknesses are therefore derived from qualitative LLM-based assessment rather than from fixed metric thresholds.

Table 1 Code-quality indicators used for LLM-based source-code analysis.

Indicator	Observed aspect	Pedagogical meaning
Readability	Naming, formatting, and expression clarity.	Maintainable code that communicates intent.
Abstraction level	Helper functions, procedures, classes, or routines that hide details.	Lower complexity in the main solution.
Modularity	Coherent components with limited responsibilities.	Maintainability and preparation for larger programs.
Code duplication	Copied logic and avoidable repetition.	Weak design habits and missed abstraction.
Input handling	Validation of user input, file input, and boundary values.	Robustness against invalid or unexpected data.
Edge-case handling	Empty inputs, limits, exceptional cases, and unusual states.	Robust boundary-condition reasoning.
Algorithmic complexity	Approximate efficiency of algorithms and data structures.	Solution efficiency and scalability.
Defensive programming	Error handling, precondition checks, and invalid-state protection.	Robust behavior under imperfect conditions.
Recurring code smells	Long functions, global mutable state, and hard-coded assumptions.	Design habits that can persist across assignments.

An example of the code analysis output is illustrated in Fig. 1. The figure presents evaluated indicators and highlights areas where the student solution exhibits lower quality. The numerical values shown in the interface are diagnostic, interface-level indicators generated from the LLM assessment. They help present the analysis to the user, but they are not validated measurement scores and should not be interpreted as psychometric or empirically calibrated code-quality metrics.

Because repository analysis is based on LLM interpretation, the system may produce inaccurate or incomplete assessments, especially when the submitted repository contains incomplete code, ambiguous solutions, generated code fragments, or cases that require execution-based verification. This limitation is important for both automated challenge generation and generated textual feedback.

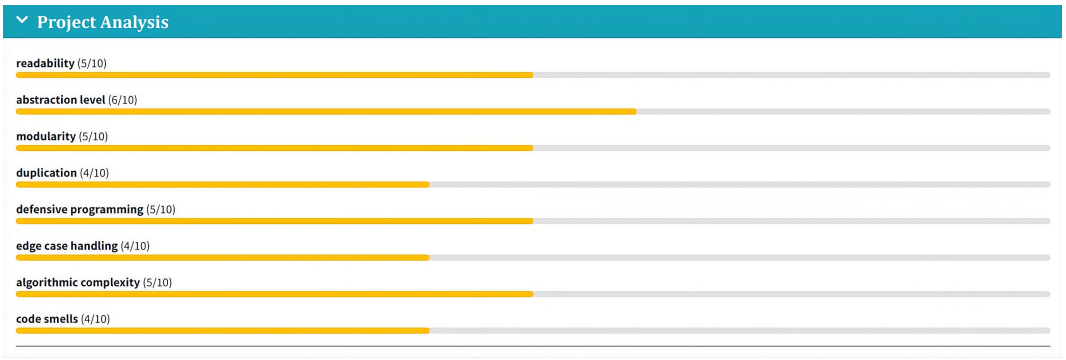

Figure 1 Interface-level example of LLM-generated diagnostic indicators for source-code analysis.

3.5 Adaptive Content Generation

The system supports two forms of generated content. First, it generates personalized programming challenges from source-code analysis. Second, it generates parameterized tests from user-defined parameters such as topic, difficulty level, and question type. The role of the language model in the system is illustrated in Fig. 2.

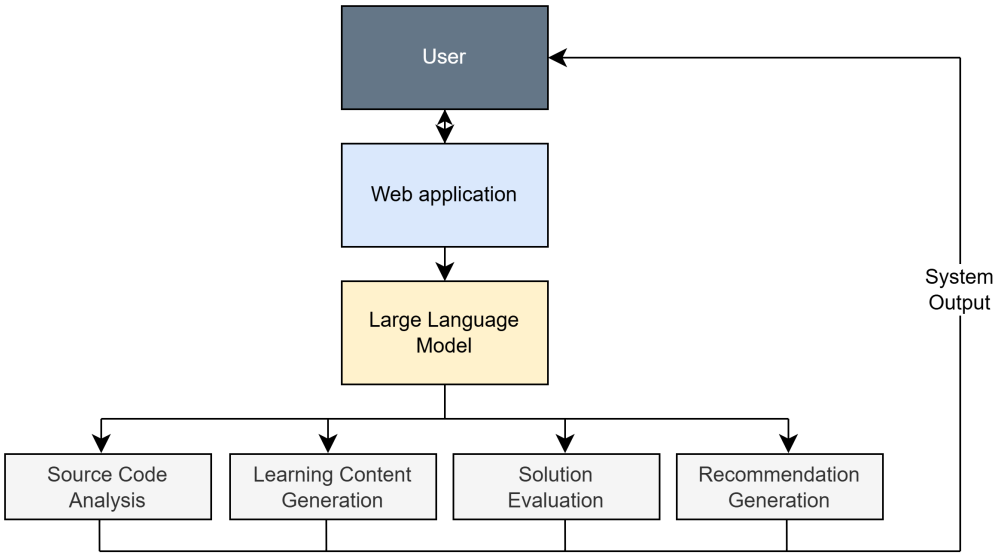

Figure 2 Conceptual model of the role of the language model in personalized content generation.

The adaptive logic is deliberately simple in the current prototype. The system asks the LLM to identify weaknesses from the selected code-quality indicators and then uses the identified weakness as the target for task generation. No fixed thresholds, statistical weights, or validated classification boundaries are used. For example, if the analysis repeatedly identifies missing input validation and insufficient edge-case handling, the generated challenge should require the student to solve a problem where invalid input, empty input, and boundary cases are part of the expected solution.

Generation is prompt-based. The LLM receives structured instructions containing the target topic or weakness, expected output format, challenge constraints, and evaluation requirements. When a machine-readable format is expected, such as JSON for generated

tests or challenge metadata, the system checks at least structural validity before storing or presenting the generated object. This validation checks whether the output follows the expected format; it is not a full factual or pedagogical validation of the generated content.

The manuscript remains model-agnostic because the evaluation focuses on the implemented workflow rather than benchmarking a specific LLM.

A compact version of the prompt structure is shown below. The actual prompt can include additional implementation-specific fields, but the main elements remain the target weakness, output constraints, and evaluation focus.

Role: Programming education assistant.
Input: Selected language or languages, repository-analysis summary, target weakness, course topic, and difficulty.
Task: Generate one programming challenge that trains the target weakness.
Constraints: Do not include a full solution; use the requested language context; include edge cases when relevant.
Output JSON: `title`, `problem_statement`, `input_output_format`, `constraints`, `example_tests`, `evaluation_rubric`, and `feedback_focus`.
Evaluation: Score a submitted solution against correctness, the target weakness, readability, robustness, and complexity.

Quality control of AI-generated content is still limited. During development and testing, one author manually inspected generated challenges and questions as an informal development-time check. This inspection helped detect malformed or unsuitable outputs, but it was not a formal expert validation study and did not involve multiple independent evaluators. The current implementation also does not include a robust automatic mechanism for detecting semantic duplicates among generated programming challenges.

3.6 Example Scenario

Consider a student who connects the system account to the university version control system and selects two repositories from previous C assignments: one focused on arrays and one focused on file processing. The system analyzes the current state of both repositories and finds that the code is mostly functional but contains long functions, repeated input-parsing logic, limited validation of file contents, and little handling of empty input files. These findings are interpreted as weaker modularity, duplication, input handling, and edge-case handling.

The generated personalized challenge asks the student to implement a small file-processing program that reads a list of integers, validates malformed lines, handles an empty file, separates parsing from computation, and reports summary statistics. The expected evaluation focuses not only on producing the correct result, but also on separating the input parser from the computation, avoiding duplicated validation logic, and handling boundary cases. Table 2 is an illustrative, edited example prepared to explain the workflow. It is not a logged output from the user evaluation and should not be read as empirical evidence about generation quality.

3.7 System Workflow

The complete workflow can be summarized as a sequence of student and system actions. The student registers or logs in, connects the application to the university version control system, and selects one or more repositories for analysis. The system retrieves the current repository

Table 2 Compact example of a generated personalized challenge.

Field	Example content
Identified weakness	Input validation, edge-case handling, duplicated parsing logic.
Generated challenge	Write a C program that reads integers from a text file, ignores malformed lines with a warning count, handles an empty file without crashing, and prints the minimum, maximum, average, and number of valid values.
Evaluation focus	Correct output, explicit handling of empty and malformed input, modular parsing and computation functions, limited duplication, and readable naming.
Feedback focus	Explain missing validation, duplicated checks, unclear decomposition, and boundary cases that were not handled.

content and performs LLM-based analysis of the selected code-quality indicators. The identified weakness is then used to generate a challenge and a corresponding evaluation rubric. The student solves the generated task and submits the solution through the application. Finally, the system applies automated evaluation and returns a score together with textual feedback.

The same platform also supports instructor-defined materials and tests. Teachers can create or manage tests, and students can complete predefined or parameterized generated tests. Histories of completed tests and programming challenges are stored and shown to both students and teachers. These histories support review and transparency, but the current paper does not claim that they are used by a validated continuous personalization algorithm.

4 Results

This section presents a prototype usability and functionality evaluation of the proposed system. The goal was to determine whether students could complete the main workflows of the implemented prototype and how they perceived its usability. The evaluation was not designed to measure learning gains, retention, or improvement in programming performance.

4.1 User Testing

The proposed system was evaluated through controlled user testing involving 11 students from the Technical University of Košice. The participants were divided into two groups based on their level of study: 6 students in the bachelor's degree program and 5 students in the master's degree program. The small number of participants limits the generalizability of the results.

The evaluation used predefined scenarios covering key system functionality: registration and login, test generation, integration with the university version control system, repository selection, and generation of personalized programming challenges. During testing, the time required to complete tasks, success rate, and user interaction issues were recorded. These measures are treated as usability indicators, not as evidence of educational effectiveness.

In the first scenario, focused on registration, login, and test generation, all participants successfully completed the tasks. The average completion time was 136.73 seconds, with only minor differences between undergraduate and graduate students. These results indicate that the tested workflow could be completed by users from different study levels.

The second scenario, involving integration with the university version control system, was successfully completed by 10 out of 11 participants, with an average completion time of 97 seconds. The main difficulty identified was related to the process of generating an access token. Based on this finding, improvements were made to the user interface by providing clearer guidance.

In the third scenario, focused on access to and initiation of personalized programming challenge generation, all participants successfully accessed and initiated personalized challenge generation. The average interaction time was 27 seconds. The results suggest that this functionality was straightforward to access and initiate in the tested setting, but they do not show that the generated challenge improved learning or that participants completed the programming task itself.

These findings indicate that participants were able to complete the main tested workflows in the evaluated setting. The consistency of results suggests that no major usability barriers were observed during the evaluation.

4.2 Usability Evaluation

The usability of the system was further evaluated using a questionnaire based on the System Usability Scale (SUS), a widely used standardized method for assessing the perceived usability of interactive systems [3]. The SUS questionnaire measures user satisfaction and perceived ease of use, not learning outcomes.

The system achieved a SUS score of **86.54**. According to common SUS interpretation guidelines, scores above 80 are often interpreted as positive perceived usability. In this small-sample evaluation, the score suggests that the system was positively perceived by the participating students from the perspective of user interaction.

The results also indicate a willingness among users to use the system regularly. As shown in Fig. 3, most participants expressed a positive attitude towards regular use of the system. The majority of responses fall into the highest agreement category, while the remaining responses are also positive.

This distribution of responses suggests that users perceived the system as useful and easy to use in the context of the tested scenarios. The absence of negative responses is consistent with the usability findings, although longer-term use would be needed to assess sustained adoption.

4.3 Summary of Results

Overall, the results indicate that the implemented prototype was functional and usable in the evaluated scenarios. Participants could connect to the version control system, generate tests, and access personalized challenge generation. Identified issues were mainly related to usability aspects, especially guidance for access-token creation. The evaluation does not provide evidence that the system improves students' programming performance, retention, or learning gains.

5 Discussion

The results should be interpreted as an initial prototype evaluation. The observed task completion rates, completion times, and SUS score indicate that the main workflows were accessible to the 11 participating students. These results are useful for assessing whether the system can be operated by students, but they are not sufficient for drawing conclusions about educational effectiveness.

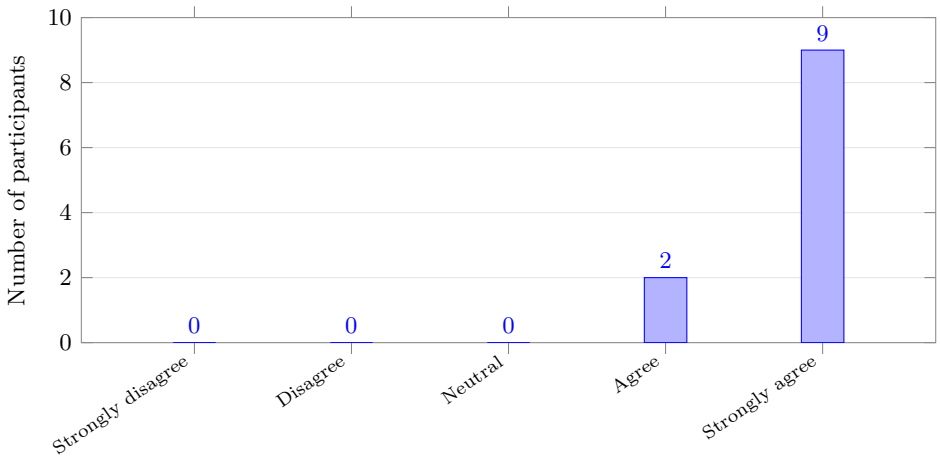

Figure 3 Distribution of responses regarding the willingness to use the system regularly.

The repository-based workflow provides a practical way to connect programming coursework with personalized practice. By allowing students to choose repositories and by analyzing the current state of those repositories, the system can generate an initial task that reflects aspects of the student's own code. This is a technically feasible basis for personalization, but further work is needed before it can be treated as a validated adaptive learning mechanism.

The main value of the prototype lies in making repository data actionable for formative programming practice, rather than using it only for assessment or retrospective analytics.

The use of LLMs also introduces risks. LLM-generated analysis and feedback may contain inaccuracies, especially for incomplete code, ambiguous solutions, or cases requiring execution-based verification. Generated challenges may be malformed, pedagogically weak, too similar to previous tasks, or insufficiently aligned with a course objective. The current system can enforce format-level constraints and was manually inspected by one author during development, but it has not yet undergone systematic expert evaluation with multiple evaluators.

5.1 Threats to Validity

The evaluation has several limitations. First, the sample size was small: only 11 students participated, which limits generalizability. Second, the evaluation was conducted in one university context, the Technical University of Košice, and the results may not transfer directly to other institutions, courses, programming languages, or repository practices. Third, testing was short-term and focused on predefined workflows. Fourth, there was no control group and no longitudinal follow-up. Fifth, the evaluation did not directly measure learning gains, retention, or improvement in programming performance. Sixth, the system has not yet been deployed at larger scale, and there are currently no deployment data showing improvement in students' programming performance, retention, or learning gains. Finally, detailed information about participants' prior programming experience and Git or version-control experience was not collected, which limits interpretation of the task-completion times.

There are also technical limitations. The current version does not define a formally validated weighting scheme for code-quality indicators and does not define empirically validated thresholds for weak, medium, or strong performance levels. Repository analysis

uses the current state of selected repositories rather than commit history. Repository preprocessing and context-limit handling are not yet robustly validated, so large repositories or repositories with many generated and non-source files may require manual or heuristic reduction before analysis. The system does not yet include robust semantic duplicate detection for generated programming challenges. Content quality control is limited to format-level checks, informal manual inspection during development, and ordinary testing of system workflows.

5.2 Future Work

Future work should focus on larger-scale deployment and controlled evaluation of learning outcomes. A stronger study design should include a control group, longitudinal follow-up, direct measurement of programming performance, and analysis of whether personalized challenges lead to measurable learning gains.

Further technical work is also needed. Priorities include systematic expert validation of generated content and feedback, duplicate detection for generated challenges, refinement of the operational student profile into a better supported student model, empirically grounded metric weighting, support for repository history and commit-level analysis, and stronger safeguards for AI-generated evaluation. Additional work should also examine how teachers can review, approve, or adjust generated tasks before students use them in graded or high-stakes contexts.

6 Conclusion

This paper presented the design and implementation of a web-based prototype for personalized programming practice. The concrete technical contribution is the integration of repository selection, LLM-based repository-level analysis, personalized challenge generation, automated feedback, and activity history into one operational workflow. The system integrates with a university version control system, lets students select repositories for analysis, applies LLM-based assessment of repository-level code-quality indicators, generates personalized programming challenges, and returns automated scores with textual feedback. It also supports parameterized test generation and stores student activity, test history, challenge history, scores, and generated feedback.

The prototype demonstrates the technical feasibility and initial usability of integrating repository-based code analysis with AI-generated personalized programming tasks. The evaluation with 11 students showed that participants could complete the main workflows and reported positive perceived usability, reflected in a SUS score of 86.54. These results should be interpreted as usability and functionality findings only.

The current work does not show that the system improves programming ability, retention, or learning outcomes. Its educational impact remains future work. Before such claims can be made, the system requires larger-scale deployment, controlled learning-effect evaluation, stronger validation of generated content and feedback, and improved safeguards around LLM-based analysis and assessment. Nevertheless, the prototype provides a concrete implementation path for connecting programming-course repositories with generated practice tasks and feedback.

References

1 Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005. doi:10.1080/08993400500150747.

2 Norbert Baláž, Jaroslav Porubän, Marek Horváth, and Tomáš Kormaník. Using chatgpt during implementation of programs in education. In *5th International Computer Programming Education Conference (ICPEC 2024)*, pages 18–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASIcs.ICPEC.2024.18.

3 John Brooke. Sus: A quick and dirty usability scale. In Patrick W. Jordan, Bruce Thomas, Bernard A. Weerdmeester, and Ian L. McClelland, editors, *Usability Evaluation in Industry*, pages 189–194. Taylor and Francis, 1996.

4 Peter Brusilovsky and Christoph Peylo. Adaptive and intelligent web-based educational systems. *International Journal of Artificial Intelligence in Education*, 13(2–4):159–172, 2003. doi:10.3233/IRG-2003-13(2-4)02.

5 Marilza Antunes de Lemos and Roseli de Deus Lopes. Challenges of programming apprentice. In *Proceeding of the International Conference on e-Education, Entertainment and e-Management*, pages 320–325. IEEE, 2011. doi:10.1109/iceeem.2011.6137816.

6 Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 1136–1142. Association for Computing Machinery, 2023. doi:10.1145/3545945.3569823.

7 Salvador Espana-Boquera, David Guerrero-Lopez, Alvaro Hermida-Perez, Josep Silva, and Jose V. Benlloch-Dualde. Analyzing the learning process (in programming) by using data collected from an online ide. In *2017 16th International Conference on Information Technology Based Higher Education and Training (ITHET)*, pages 1–4. IEEE, 2017. doi:10.1109/ithet.2017.8067822.

8 Marek Horváth, Tomáš Kormaník, and Jaroslav Porubän. Adaptation of automated assessment system for large programming courses. In *5th International Computer Programming Education Conference (ICPEC 2024)*, pages 4–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASIcs.ICPEC.2024.4.

9 Marek Horváth and Emília Pietriková. An experimental comparison of three code similarity tools on over 1,000 student projects. In *2024 IEEE 22nd World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 000423–000428. IEEE, 2024.

10 Marek Horváth, Emília Pietriková, and Filip Gurbál'. Personalized learning analytics through static code analysis in computer science education. *Acta Informatica Pragensia*, 2026(1):54–71, 2026.

11 Marek Horvath, Lukas Tomascik, Nikola Geciova, Norbert Adam, and Emilia Pietrikova. Benchmarking ai models for grading cs assignments across multiple domains. *Acta Polytechnica Hungarica*, 23(5):207–226, 2026.

12 Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Á. Rubio, Judy Sheard, Barbara Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE Conference on Working Group Reports*, pages 41–63. Association for Computing Machinery, 2015. doi:10.1145/2858796.2858798.

13 Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23. Association for Computing Machinery, 2023. doi:10.1145/3544548.3580919.

14 Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19(1):3:1–3:43, 2018. doi:10.1145/3231711.

15 Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery, 2023. doi:10.1145/3631802.3631830.

16 Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. Teaching cs50 with ai: Leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, pages 750–756, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3626252.3630938.

17 Stephen Nutbrown and Colin Higgins. Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education*, 26(2–3):104–128, 2016. doi:10.1080/08993408.2016.1179865.

18 Ivica Pesovski, Daniela Vorkel, and Vladimir Trajkovik. Ai-powered education: Rethinking the way programming is taught using ai tools and reversed bloom's taxonomy. In *2024 21st International Conference on Information Technology Based Higher Education and Training (ITHET)*, pages 1–6. IEEE, 2024. doi:10.1109/ithet61869.2024.10837604.

19 Igor Skoric, Boris Pein, and Tihomir Orehovacki. Selecting the most appropriate web ide for learning programming using ahp. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 877–882. IEEE, 2016. doi:10.1109/mipro.2016.7522263.

20 Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. Cs1-llm: Integrating llms into cs1 instruction. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, pages 297–303, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649217.3653584.

21 Kurt VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011. doi:10.1080/00461520.2011.611369.

22 Murad Zeer, Rawan Wael Siaj, Jiries Abu Ghannam, and Mohammad Kanan. Ethics of artificial intelligence in university education. In *2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI)*, pages 1–4. IEEE, 2023. doi:10.1109/eiceeai60672.2023.10590285.

23 He Zhi-guo and Zhou Chao-xuan. The reformation of teaching methods and curriculum system for software courses. In *2012 International Symposium on Information Technologies in Medicine and Education*, pages 1041–1045. IEEE, 2012. doi:10.1109/itime.2012.6291479.