Acta Informatica Pragensia
2026, Volume 15, Issue 1, 54–71
https://doi.org/10.18267/j.aip.283

PRAGUE UNIVERSITY OF ECONOMICS AND BUSINESS

Article Open Access

Personalized Learning Analytics Through Static Code Analysis in Computer Science Education

Marek Horváth, Emília Pietriková, Filip Gurbáľ

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovak Republic

Corresponding author: Marek Horváth (marek.horvath@tuke.sk)

Editorial Record

First submission received:
April 24, 2025

Revisions received:
June 22, 2025
July 22, 2025

Accepted for publication:
July 25, 2025

Academic Editor:
Bartosz Marcinkowski
University of Gdańsk, Poland

This article was accepted for publication by the Academic Editor upon evaluation of the reviewers' comments.

How to cite this article:
Horváth, M., Pietriková, E., & Gurbáľ, F. (2026). Personalized Learning Analytics Through Static Code Analysis in Computer Science Education. *Acta Informatica Pragensia*, 15(1), 54–71. https://doi.org/10.18267/j.aip.283

Abstract

Background: Learning programming is often difficult for beginners, primarily because of the challenge of providing timely and personalized feedback in large educational environments. While automated assessment systems have improved efficiency in grading and feedback, they typically focus on correctness and often lack personalized guidance concerning code quality, readability, and maintainability.

Objective: This study aims to investigate whether integrating static code analysis into automated assessment systems to provide personalized feedback can effectively enhance students code quality, learning process, and engagement in programming courses.

Methods: We designed a personalized feedback system integrated with static analysis tools (Cppcheck and Clang-format), deployed within an existing automated assessment platform used by undergraduate programming students. The system was evaluated in a controlled experiment involving 60 students randomly divided into control and treatment groups. The effectiveness of personalized feedback was measured through quantitative metrics (style violations, potential bugs, and design issues), qualitative surveys, and submission behaviours over multiple assignments.

Results: Results demonstrated that students receiving personalized feedback improved their code quality, reducing the number of style violations by 76%, potential bugs by 52%, and structural issues by 32% compared to the control group. Students also expressed higher satisfaction, increased motivation, and greater willingness to iteratively refine their code based on personalized feedback.

Conclusion: The integration of static code analysis for personalized feedback not only enhances code quality but also helps a deeper understanding of good programming practices among students. Future research should focus on making feedback systems more adaptive, incorporating intelligent tutoring techniques, and exploring long-term impacts on programming habits and skill retention.

Index Terms

Personalized feedback; Static code analysis; Student engagement; Code quality improvement; Learning analytics; Student-centered feedback; Clean code principles; Code quality metrics; Interactive feedback.

1 INTRODUCTION

Teaching programming effectively, especially in large classes, is naturally challenging because of the varying skill levels and different learning speeds among students. Beginners frequently require prompt, detailed, and individualized feedback to identify their mistakes clearly and continuously improve (Gallien and Oomen-Early, 2008). However, traditional methods relying on manual feedback from instructors or teaching assistants do not scale efficiently.

With growing class sizes, timely personalized feedback becomes difficult to achieve, often resulting in generic, shallow comments that do not address the specific issues students encounter. As a result, learners might become frustrated, lose motivation, or develop persistent misconceptions because they lack proper guidance.

To address this issue, educational institutions commonly employ automated assessment tools. These systems usually focus on verifying program correctness by running predefined test cases or performing basic style checks (Rocha et al., 2023). Although these tools greatly enhance feedback delivery speed, their generic and standardized nature often limits their educational effectiveness. They mainly inform students whether their solution meets functional requirements, often overlooking important aspects of programming such as readability, maintainability, and compliance with good coding practices. As a result, students may receive feedback that, although technically correct, does not effectively encourage deeper learning or meaningful improvement in their coding skills.

Static code analysis provides a promising approach for improving the quality and relevance of feedback in programming education. Also known as linting, static analysis tools inspect source code without executing it, identifying potential issues such as stylistic inconsistencies, poor coding practices, redundant or inefficient code, and possible bugs (Messer et al., 2024). In professional software development, static analysis is commonly used to ensure high code quality and consistency. Introducing these tools into educational settings holds great potential, as they can automatically detect numerous issues in student code submissions. Unlike feedback focused exclusively on correctness, static analysis provides students with practical suggestions for improving their coding style, structure, and overall code quality, leading to a more complete learning experience.

However, a critical challenge remains when using static analysis tools in educational settings. Since these tools are primarily developed for professional programmers, their feedback is often delivered as technical warnings that can easily overwhelm novice learners (Fehnker and Blok, 2017). Beginners may struggle with interpreting the feedback due to its technical wording, large number of warnings, or difficulty in determining which warnings to address first. The complexity of these messages may even lead students to ignore useful suggestions, reducing the overall effectiveness of the feedback.

To address these challenges, feedback personalization becomes essential. Personalizing static analysis feedback involves customizing the feedback specifically to each student's current skill level, understanding of coding concepts, and individual learning path. Instead of presenting students with overwhelming lists of technical warnings, personalized feedback selectively filters and prioritizes relevant issues, presents messages in instructional and supportive language, and highlights recurring problems that students struggle with the most. This tailored approach makes feedback clearer, more manageable, and directly useful, promoting a constructive learning environment.

This study explores the integration of personalized static analysis feedback into introductory programming courses. We aim to answer two primary research questions:

- **RQ1:** Does integrating personalized static analysis feedback into programming assignments improve students' code quality and learning outcomes compared to standard feedback methods?
- **RQ2:** How do students perceive and interact with personalized static analysis feedback in terms of usability, helpfulness, and motivational aspects?

To explore these questions, we developed a specialized feedback system integrated with widely used static analysis tools (*Cppcheck* and *Clang-format*). This system was implemented within an existing automated assessment environment utilized in undergraduate programming courses at the *Technical University in Košice*. We conducted a controlled experiment involving students randomly divided into control and experimental groups. Student submissions were evaluated quantitatively to measure improvements in code quality and qualitatively through surveys to gauge students' personal experiences and satisfaction with the feedback received.

In the following sections, we first discuss existing research related to automated feedback methods, particularly focusing on static code analysis and personalization strategies. Then, we describe the methodology employed, including details of our experimental setup and feedback system development. Afterward, we present the results obtained and interpret them within the broader context of existing literature, also addressing any limitations identified. Finally, we summarize our main contributions and propose potential directions for future research focused on personalized feedback in programming education.

2 RESEARCH BACKGROUND

The challenge of delivering effective feedback in programming education has been widely discussed in recent literature (Biňas and Pietriková, 2022). Automated assessment systems are now common in both large-scale university settings and online courses. These systems help reduce instructors' workload while providing timely feedback to students. A systematic review found that most tools emphasize correctness, typically through dynamic analysis like unit tests or static comparisons with a reference solution (Messer et al., 2024). Feedback in such systems is often limited to binary outcomes (Horváth and Gurbál, 2024), such as "passed" or "failed", or output differences. Although immediate, this type of feedback does not explain how students could improve their code in terms of structure, readability, or style. The same review emphasized that only a small proportion of tools consider deeper code quality factors like maintainability or documentation, and even fewer include static analysis metrics. Nevertheless, it confirmed that rapid feedback delivery supports iterative development, helping students improve over multiple submissions, which in turn contributes to higher satisfaction and more learning opportunities.

Static analysis in educational contexts has been explored by several researchers aiming to supplement correctness-based assessment with code quality evaluation (Truong et al., 2004). Study (Nutbrown and Higgins, 2016) evaluated the integration of static analysis into *Java* assignment grading and initially found that the alignment between rule-based static analysis and human grading was weak, suggesting that many automated warnings were not relevant or were misleading for novice code. However, after refining the selected rules and adjusting the way feedback was presented, the correlation with instructor assessments improved substantially. This study underlined the importance of calibration and thoughtful selection of rules when using static analysis in learning environments.

Another contribution (Fehnker and Blok, 2017) proposed a set of custom static rules added to the *PMD* tool (Copeland, 2005) specifically designed for beginners. These rules addressed frequent novice mistakes such as inconsistent naming, redundant code, or misuse of control structures. Their system was embedded in an *IDE* and provided live feedback while coding. Students responded well to these personalized suggestions, and the system successfully identified many issues relevant to their skill level. On the other hand, the study also highlighted that warnings from professional tools often contain information that is either too advanced or irrelevant to novices, which can distract or overwhelm learners. A separate study observed that students addressed simple issues like formatting or unused variables quickly but struggled more with abstract warnings such as overly complex methods or architectural suggestions. In several cases, students acknowledged the feedback but postponed resolving design-level problems due to time constraints or fear of breaking functionality. These findings collectively suggest that static analysis tools can assist in education, but must be filtered, explained, and contextualized properly.

The issue of feedback overload and clarity is echoed across multiple studies. Large volumes of warnings, especially without prioritization or explanation, can reduce student engagement. This has led to calls for educational static analysis systems that are simplified, filtered, and accompanied by tailored explanations. Several researchers point out that linters designed for industrial use may not translate well into classrooms unless adapted. Even basic formatting or naming violations can confuse students if not clearly connected to coding principles they are learning. Hence, tools must balance between guiding students and overwhelming them with details. Another important strand of related work is the personalization of feedback in computing education. General education literature has long recognized the benefits of personalized learning interventions. In the context of programming, intelligent tutoring systems attempt to offer this kind of adaptivity by analysing student performance over time and tailoring feedback accordingly. A recent mapping study concluded that personalization is one of the most underdeveloped aspects in automated feedback systems (Gallien and Oomen-Early, 2008). Most systems continue to present static, generic messages without accounting for student history, performance patterns, or learning progress. The review proposed several design directions for incorporating personal learning profiles into future systems, including rule prioritization, graduated hints, and differentiated language complexity.

The broader trend in computing education supports moving beyond binary correctness toward richer, formative feedback. Research shows that students benefit more from feedback that explains why something is problematic and how to address it, not just that it is wrong. However, building such systems is complex, particularly when balancing automation, precision, and accessibility. Combining static analysis with personalization is seen as a promising direction. Instead of overwhelming students with every warning, a system could prioritize messages based on past behaviour, hide repeated information, or escalate detail gradually. Some proposals include dynamic systems that adapt based on submission history, student profile, or demonstrated skill level.

Our contribution builds on these themes by proposing a static analysis feedback system that incorporates basic personalization. We adopt static tools not just for correctness but to evaluate code clarity and structure and filter the feedback to match the student's learning needs. By doing so, we attempt to address the gaps reported in prior work. Absence of adaptive guidance, the risk of feedback overload, and the lack of educational tailoring in static analysers. In the following section, we describe how our system was implemented, the configuration choices made for educational appropriateness, and how it was tested in a real classroom setting.

2.1 Maintaining Code Quality

Following up on the need for meaningful feedback in programming education, it is equally important to guide students in writing code that is not only functionally correct but also readable, maintainable, and consistent. As students' progress beyond solving simple problems, the clarity and structure of their solutions become increasingly important. In this context, teaching code quality forms a key part of programming education.

Article (Martin, 2008) emphasizes that good code should be easy to read, understand, and modify. According to him, software is not done until it is done right. Instilling this mindset in students early in their education encourages habits that go beyond getting the program to work; it teaches them to think critically about their code as a long-term artifact that might be read or reused by others later. These ideas help shift the focus from *"getting it done"* to *"doing it well"*, which is relevant in collaborative or long-term software projects. At the same time, Martin also warns against excessive commenting, arguing that clear structure and naming should make the code self-explanatory. In our approach, we encourage students to use comments meaningfully, but not as a substitute for clean and understandable code.

2.2 Static Analysis

One way to help students write better code is using static code analysis (Molnar et al., 2020). This approach allows tools to examine code without running it, using pre-defined rules to identify issues such as poor structure or violations of style guidelines. For students, these tools act as early support mechanisms, offering hints and corrections before the code is ever compiled or executed.

Static analysis is distinct from dynamic analysis, which requires program execution. While dynamic tools examine how code behaves at runtime, static tools analyse structure, logic, and syntax at rest. This makes static tools particularly useful during early stages of development or in educational settings where immediate feedback is valuable. The aim is not only to detect bugs or vulnerabilities but also to highlight opportunities for cleaner, more maintainable code (Cooper and Torczon, 2023).

These tools provide feedback that can help learners recognize and address issues that go beyond correctness, such as naming, code duplication (Bubenkova et al., 2025), complexity, or unused variables. They are often used in industry to enforce code quality standards and catch problems early. By incorporating them into programming education, students can adopt similar habits that professional developers rely on. Several studies (Jurado, 2021; Chen et al., 2022) have shown that using static analysis as a feedback tool for students can not only help them catch mistakes earlier but also improve their understanding of how good code is written. In this way, the analysis becomes a learning tool, not just a correctness checker.

2.3 Code Conventions

Alongside static analysis, consistent code conventions are essential in helping students learn to write clean, understandable code. These rules provide a shared style and vocabulary that make the code easier to read, review, and maintain. While style alone does not determine correctness, poor formatting or naming can make code harder to understand, which often leads to bugs or miscommunication. Author (McConnell, 2004) argues that consistent style is one of the cornerstones of good software construction. Teaching conventions early helps form habits that reduce friction in collaborative environments and align with professional development practices.

To illustrate this, **Table 1**. presents selected code snippets that contrast common mistakes with their improved counterparts. These examples are intentionally brief to emphasize clarity and stylistic consistency.

Table 1. *Selected examples of common code convention mistakes and their improved versions.*

Incorrect (non-compliant)	Correct (compliant)

`// Function naming and return`
`int a(int b) {`
`  return b*b;`
`}`
	`int computeSquare(int number) {`
	`  return number * number;`
	`}`

`// Formatting and logic clarity`
`if(x>0)y=10;else y=20;`
	`if (x > 0) {`
	`  y = 10;`
	`} else {`
	`  y = 20;`
	`}`

`//Inconsistent spacing and readability`
`for(int i=0;i<5;i++)printf("%d",i);`
	`for (int i = 0; i < 5; i++) {`
	`  printf("%d", i);`
	`}`

`// Vague comment`
`// area`
`double a = w * h;`
	`// Calculate rectangle area from width and height`
	`double area = width * height;`

As these examples suggest, even small improvements in naming or formatting can make a substantial difference in code clarity. Encouraging students to adopt such practices helps reduce misunderstandings and improves maintainability. These habits become especially valuable in collaborative or long-term projects where readability is crucial.

3 SYSTEM IMPLEMENTATION AND STUDY DESIGN

The study follows a quasi-experimental design, conducted in a real-world educational setting without random assignment. Its aim was to evaluate whether integrating personalized static code analysis into a programming course would influence student behavior and code quality. Student groups from different semesters were exposed to varying levels of automated and personalized feedback, which enabled comparative analysis of outcomes. Although no formal control group was defined, the study included multiple naturally occurring cohorts, allowing us to assess the impact of feedback interventions over time. To support this objective, we extended an existing submission and feedback system by embedding tools capable of analysing student source code from multiple perspectives. These tools included *Cppcheck* for static code analysis and *Clang-format* for style and formatting evaluation (Delev and Gjorgjevikj, 2017). The extended system generated tailored feedback that students received via email as they progressed through their assignments. Feedback was delivered during the process of completing the assignment, allowing for iterative improvements. This personalized feedback was constructed by interpreting raw outputs from the analysis tools and mapping them to readable explanations and actionable suggestions. In addition to identifying problems, the messages aimed to explain their rationale, impact, and possible solutions. Feedback generation was governed by several heuristics. The system prioritized serious issues over stylistic ones and reduced redundancy by downplaying repeated warnings. Feedback wording varied depending on prior student behavior. For instance, students who had previously addressed similar warnings received shorter suggestions, while others were provided more detailed guidance. The tone across all messages was constructive, in accordance with findings from (Fehnker and Blok, 2017) that overly technical or judgmental messages reduce effectiveness in novice-oriented environments. Examples of feedback transformations are illustrated in **Table 2**. These demonstrate how a raw warning from a tool like *Cppcheck* was reframed to offer pedagogically valuable guidance.

The evaluation was performed over two programming assignments during one academic semester. Students were divided into three target groups to ensure a broader perspective. The first group consisted of students currently enrolled in the course who received personalized feedback while working on assignments. The second group included students from previous semesters who had access to standard feedback but were later exposed to the updated version of the feedback system. The third group comprised students who had never received any such feedback during their coursework, including both recent graduates and current students with relevant programming experience. All groups worked within the same infrastructure, and students were allowed to make multiple submissions. The feedback for the active group was refreshed with every new submission, enabling them to address issues iteratively. To assess the impact, we collected several types of data:

- Submission logs and code snapshots to track resolution of detected issues.
- Final static analysis reports to evaluate remaining problems at the deadline.
- Assignment grades to determine if feedback affected performance.
- Survey responses from 171 participants across three groups.

Survey items were constructed to measure both objective outcomes and students' subjective perceptions. These included statements such as ``*did the feedback help you find and fix mistakes?*'' or ``*was the feedback helpful or confusing?*''. Thematic coding was applied to open-ended responses to identify commonly mentioned benefits and drawbacks. Following recommendations from (Rocha et al., 2023) and (Nutbrown and Higgins, 2016), the survey design and feedback mechanism emphasized clarity, accessibility, and individual relevance. This methodology enabled us to observe not only how the feedback affected immediate outcomes but also how students perceived its role in their learning.

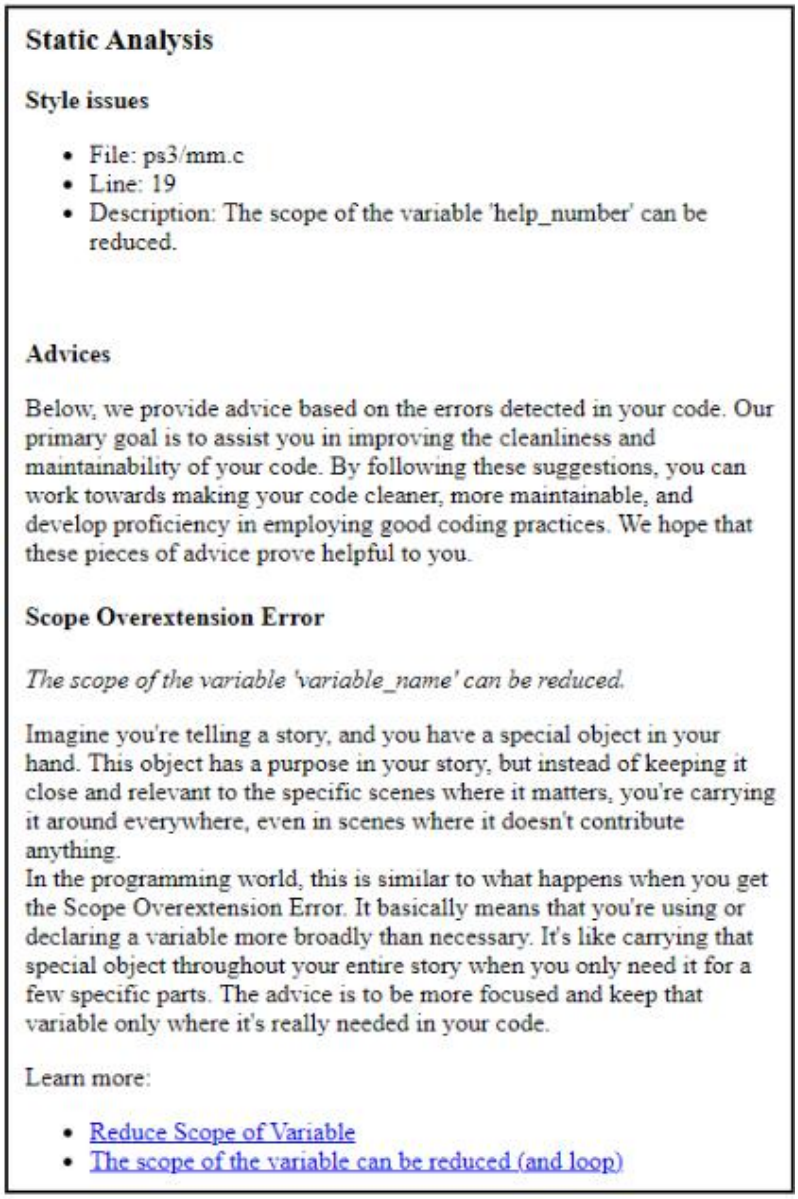
Table 2. *Comparison of Static Analyzer Messages and Personalized Feedback.*

Detected Issue	Typical Static Analyzer Message	Personalized Feedback to Student
Unused variable	Unused variable temp.	The variable temp is declared but never used. Consider removing it to clean up your code.
Naming convention not followed	Variable name *my_var* does not match naming standard.	Try renaming *my_var* to follow naming conventions (e.g., *myVar*). Consistent naming improves readability.
Method too long	Method *calculateResults* is excessively long.	Your method *calculateResults* could be split into smaller functions. This would improve clarity and make testing easier.
Potential null dereference	Null pointer risk (object data may be null).	data might be null here. Consider adding a check before accessing it to prevent possible crashes.
Magic number usage	Magic number used: 42.	Avoid using numbers like 42 directly in your code. Define it as a constant with a descriptive name to improve clarity.
Duplicated code block	Code block is repeated.	This section of code appears multiple times. Consider creating a function to eliminate redundancy and improve maintainability.
Unused include directive	Unused include: *math.h*.	The *math.h* header is not used in your program. You can safely remove it to reduce clutter.
Missing return statement	Control reaches end of non-void function.	This function is expected to return a value, but no return statement was found. Add a return to ensure the function behaves correctly.

3.1 Integration of Static Analysis with Cppcheck

Our automated evaluation system (Horváth et al., 2024) regularly tests programming assignments using unit tests and tracks various performance metrics. We extended this feedback system with support for static code analysis using *Cppcheck*, a tool designed to analyse *C* source code and detect potential issues without running it (Pereira and Vieira, 2020). The tool examines structure, logic, and syntax to identify possible bugs, warnings, or violations of coding conventions. After student code is submitted, it is processed by *Cppcheck* with parameters *--enable=all* to activate all checks and *--std=c11* to match the course's compilation settings. The output is stored in *JSON* format and parsed into structured statistics linked to the student's record. The issues are divided into three categories:

- Errors are serious issues that should be corrected to prevent potential runtime failures. These represent logical or syntactic mistakes that can result in undesired behavior.
- Warnings indicate constructs that may not break the code but suggest poor design or risky patterns. Students are encouraged to consider them carefully.
- Style issues relate to code formatting and readability. Although they do not affect execution, they influence maintainability and clarity.



Static Analysis

Style issues

- File: ps3/mm.c
- Line: 19
- Description: The scope of the variable 'help_number' can be reduced.

Advices

Below, we provide advice based on the errors detected in your code. Our primary goal is to assist you in improving the cleanliness and maintainability of your code. By following these suggestions, you can work towards making your code cleaner, more maintainable, and develop proficiency in employing good coding practices. We hope that these pieces of advice prove helpful to you.

Scope Overextension Error

The scope of the variable 'variable_name' can be reduced.

Imagine you're telling a story, and you have a special object in your hand. This object has a purpose in your story, but instead of keeping it close and relevant to the specific scenes where it matters, you're carrying it around everywhere, even in scenes where it doesn't contribute anything.
In the programming world, this is similar to what happens when you get the Scope Overextension Error. It basically means that you're using or declaring a variable more broadly than necessary. It's like carrying that special object throughout your entire story when you only need it for a few specific parts. The advice is to be more focused and keep that variable only where it's really needed in your code.

Learn more:

- Reduce Scope of Variable
- The scope of the variable can be reduced (and loop)

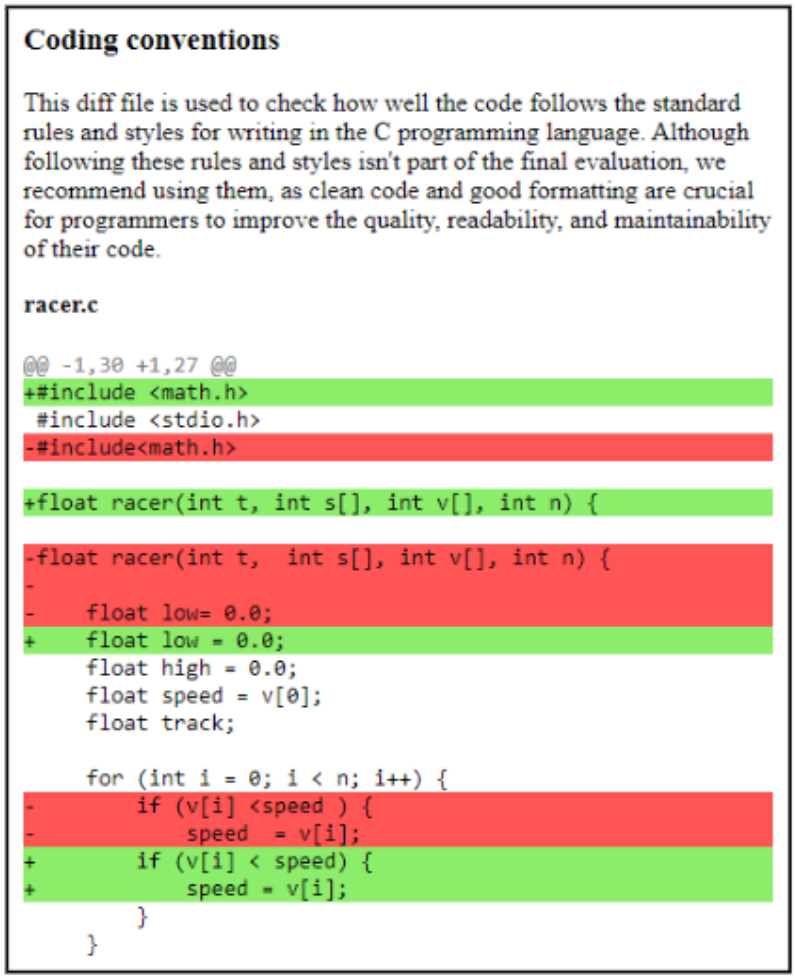
Figure 1. *Feedback example showing an overextended variable scope warning.*

The feedback students receive (**Figure 1**) is structured to reflect this categorization. Each type of message includes file name, line number, and a brief description. To increase usefulness, we added explanations for frequent errors. This includes summaries, fix suggestions, and in many cases, hyperlinks to external resources. Initially, we analysed code from 100 student repositories to determine which types of errors appeared most often. From this analysis, we selected the top 20 most frequent error patterns and developed advice for each of them. This advice includes simplified explanations and practical tips tailored to novice programmers. We noticed that simply listing errors in emails was often insufficient, some students ignored the messages or didn't understand how to act on them. Therefore, we modified the email content to include explanations next to each error. Each message now starts with the original diagnostic from *Cppcheck*, followed by a human-readable summary, an explanation of the issue, and a suggestion for improvement. Where relevant, the message includes a link to trusted documentation or tutorials.

By narrowing the focus to common and recurring issues, we kept the feedback manageable while maintaining relevance. Covering every possible *Cppcheck* output would have been unrealistic, so we emphasized those findings most useful to most students. This setup transformed error detection into a feedback loop with an educational function. The system not only detects issues but also guides students in understanding and resolving them, turning static code analysis into a practical tool for learning good coding habits.

3.2 Automated Formatting Feedback with Clang-Format

To help students write code that is not only functional but also clean and easy to read, we integrated *Clang-format* into our feedback workflow (Babati et al., 2017). While *Cppcheck* addresses deeper structural or logical problems, *clang-format* focuses on surface-level consistency, such as indentation, spacing, and bracket placement, that significantly affects how readable and maintainable the code is. The intention was not to enforce style for its own sake, but to show students how consistent formatting helps communicate ideas clearly and avoids unnecessary confusion. We prepared a custom *clang-format* configuration file tailored to the educational context. The rules we used covered the basic aspects of layout such as indentation width, brace styles, line length, and use of blank lines. The configuration aligns with common professional conventions but also avoids overly strict enforcement to reduce cognitive overhead for beginners.



Coding conventions

This diff file is used to check how well the code follows the standard rules and styles for writing in the C programming language. Although following these rules and styles isn't part of the final evaluation, we recommend using them, as clean code and good formatting are crucial for programmers to improve the quality, readability, and maintainability of their code.

racer.c

```
@@ -1,30 +1,27 @@
+#include <math.h>
 #include <stdio.h>
-#include<math.h>

+float racer(int t, int s[], int v[], int n) {

-float racer(int t,  int s[], int v[], int n) {
-
-    float low= 0.0;
+    float low = 0.0;
     float high = 0.0;
     float speed = v[0];
     float track;

     for (int i = 0; i < n; i++) {
-        if (v[i] <speed ) {
-            speed  = v[i];
+        if (v[i] < speed) {
+            speed = v[i];
         }
     }
```

Figure 2. *Formatting feedback example showing changes suggested by the system.*

The system was extended to run *clang-format* automatically during the feedback generation process. When a student submits code, the tool reformats it according to the defined rules and compares the result with the original submission. If there are differences, the system extracts the relevant changes and includes a brief message in the feedback. Rather than listing every line that was altered, we highlight only the relevant violations and summarize them in a way that points students to the underlying principle. To improve clarity, each style issue is accompanied

by a short explanation. A visual example of this kind of feedback can be seen in **Figure 2**, where an overly broad variable declaration is flagged and described using an analogy that makes the concept more relatable.

We found that simply informing students about formatting violations has limited value unless they understand why these rules matter. That's why each part of the feedback combines the technical detail with commentary written in accessible language. These messages aim to build intuition for clean code over time, helping students internalize good habits rather than memorize rules. This integration of *clang-format* turns formatting feedback into something more than a mechanical critique. It becomes an opportunity to practice discipline in presentation, to develop attention to detail, and to treat code as a form of communication. These skills are often underemphasized in early programming courses, yet they are essential for long-term development as a programmer.

3.3 Feedback system implementation

After incorporating new forms of feedback focused on static analysis and code formatting, we needed to evaluate how these changes affected students' perception of their own code quality. During the semester, students received multiple feedback emails while working on assignments, as well as a final one with summary statistics. Since the feedback was delivered throughout the assignment process, students had a chance to identify and fix issues while still working on their solutions. It is important to note that the intention of these improvements was not to improve final scores directly, but rather to help students recognize mistakes, refine their style, and develop awareness of code quality. In programming, clearer and more structured code often leads to fewer logical errors and easier maintenance. Still, the impact of such interventions is not always visible through traditional evaluation metrics.

To gain insight into how students perceived the usefulness of the new feedback, we conducted a questionnaire evaluation. We prepared a structured form designed to assess how feedback supported the development of readable and maintainable code. The form contained several groups of statements rated on a five-point scale, along with open questions. These invited students to share what they found helpful and suggest improvements. Questions were divided into four thematic parts such as overall experience, code analysis, formatting, and proposals for future development.

We also included a visual example of the feedback (**Figure 2**) to help respondents recall the format and nature of the messages they had received. This figure illustrates how code formatting suggestions are presented when comparing the student's version to the corrected one.

We distributed adapted versions of the questionnaire to three different groups:

1. group consisted of current students enrolled in the introductory programming course who received feedback during the semester. These respondents had the most direct exposure to the system in practice.
2. group included students from previous years who completed the same course before the feedback extensions were introduced. They were shown updated examples based on two older assignments, allowing them to reflect on the differences.
3. group consisted of students who had never received any form of feedback from the assistant. Many of them were already graduates working in the software industry, while others were still studying. Before filling out the questionnaire, they were asked to review feedback examples and reflect on their usefulness from a more experienced perspective.

By comparing responses from these three groups, we aimed to understand how different levels of experience and exposure to feedback affected students' perceptions. This helped us assess the broader impact of the changes beyond the course setting itself.

4 RESULTS

To evaluate the real impact of the personalized feedback mechanism integrated into the automated evaluation system, we analysed student responses from three target groups with different levels of exposure to the system. A total of 171 valid responses were collected, offering a detailed picture of how the feedback was received and how it influenced students’ perceptions, habits, and coding outcomes.

4.1 Feedback from First Respondent Group

The first group consisted of students who worked with the feedback in real time while solving assignments. According to the responses, 82.5% evaluated the feedback positively. Students appreciated the system's ability to highlight not only functional mistakes but also code quality issues. About 66% agreed that the feedback helped them write higher-quality code. The static code analysis component was singled out as useful by 73% of respondents, suggesting that students paid attention to these details and found them beneficial. Only 43% reported using coding conventions before the course. Nevertheless, about half of the group found the formatting comparisons useful, and 54% felt that this part of the feedback was clearly presented. This gap between previous habits and current appreciation suggests that students were exposed to new expectations and learned from them.

Open-ended responses reinforce these findings. Several students mentioned they had never considered formatting or naming conventions until the feedback pointed them out. One remarked, *"I always just checked if my program passed the tests, but this time I saw where my code could be better even if it worked."* Others mentioned that the emails encouraged them to revisit their code even after they had received a passing score. The addition of links to external resources was often cited as helpful, giving students the chance to read more about specific problems if they were interested. Some also described how their workflow changed over time. After the second or third feedback round, students began checking for certain recurring issues in advance. As one noted, *"By the third assignment I was already writing code with those suggestions in mind."* This illustrates a transfer effect, feedback on one assignment influenced how students approached the next one.

To explore how students perceived the role of feedback in their learning process, three specific statements were included in the questionnaire:

- **Q1**: *"I think that this kind of feedback is a great idea."*
- **Q2**: *"I think that Oracle feedback would help me to improve my results and working skills."*
- **Q3**: *"Oracle feedback would motivate me to get better."*

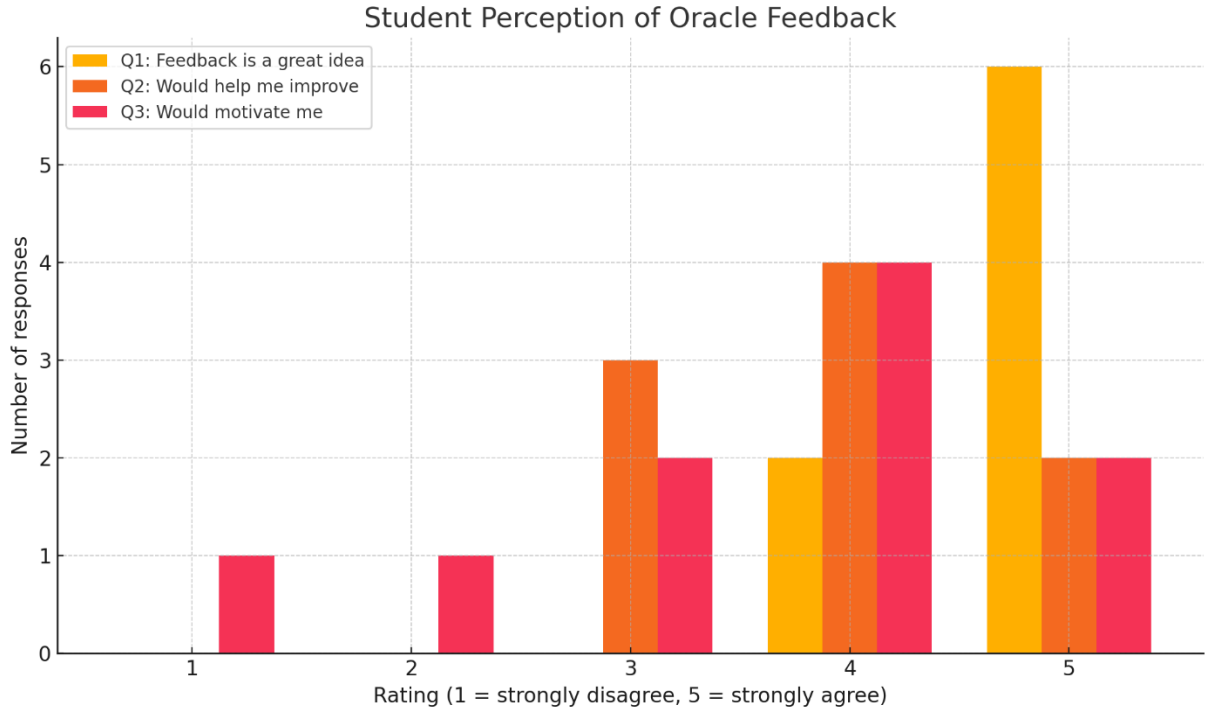


Figure 3. *Responses to three questionnaire items (Q1–Q3) about perceived usefulness and motivational impact of the feedback, measured on a five-point Likert scale.*

These questions aimed to assess the general acceptance of feedback (Q1), its perceived impact on skill improvement (Q2), and its motivational potential (Q3). The summary of responses is presented in **Figure 3**. The data show that most students responded with values of 4 or 5 on the Likert scale, suggesting strong agreement with all three statements. While Q1 reflects general appreciation, Q2 and Q3 indicate that students also found the feedback

beneficial for their learning and self-improvement. These results further support the idea that timely and relevant automated feedback can positively influence both the quality of students' code and their motivation throughout the course.

4.2 Feedback from Second Respondent Group

Students in the second group, who had completed the same course earlier, were sent updated examples of feedback based on their past assignments. Nearly all respondents supported adding the feedback system permanently to the course. One-third said it would have improved their results, and almost 78% believed the static analysis section would have helped them write better code. Only 45% indicated they would act on every suggestion, yet 88% found the advice helpful. Several noted that the tone of the messages was encouraging rather than punitive. One student wrote, *"It told me what to improve, not just what I did wrong."* Suggestions for improvement included adding more context to the reported errors, such as showing a few lines of code before and after the issue. Some also recommended highlighting the specific change in formatting with colours or better contrast. Formatting suggestions received mixed reviews. 67% found the comparison to formatted code helpful, but only 44% thought it was clearly presented. Several students commented that they initially found the visual formatting diff difficult to read but appreciated it more after understanding how it worked.

Multiple responses pointed out that feedback on quality is often missing from automated systems. One respondent wrote, *"When I was taking the course, all that mattered was passing the tests. I would have appreciated this kind of feedback."* Such statements reinforce the notion that correctness and quality must be addressed together in early programming education.

Student perception also confirmed that static code analysis was seen as a valuable addition. Many students agreed that Oracle should be used every year during *ZAP* and *PVJC* courses (Q1), and more than half reported that static code analysis was useful (Q2). This aligns with the observation that such feedback helped them not only identify issues but also understand the reasoning behind corrections. Moreover, several students stated that feedback based on static analysis helps them write code of better quality (Q3). These reactions support the idea that early exposure to code quality principles can influence long-term coding habits. **Figure 4** summarizes the distribution of responses to these questions, showing generally strong agreement across all three.

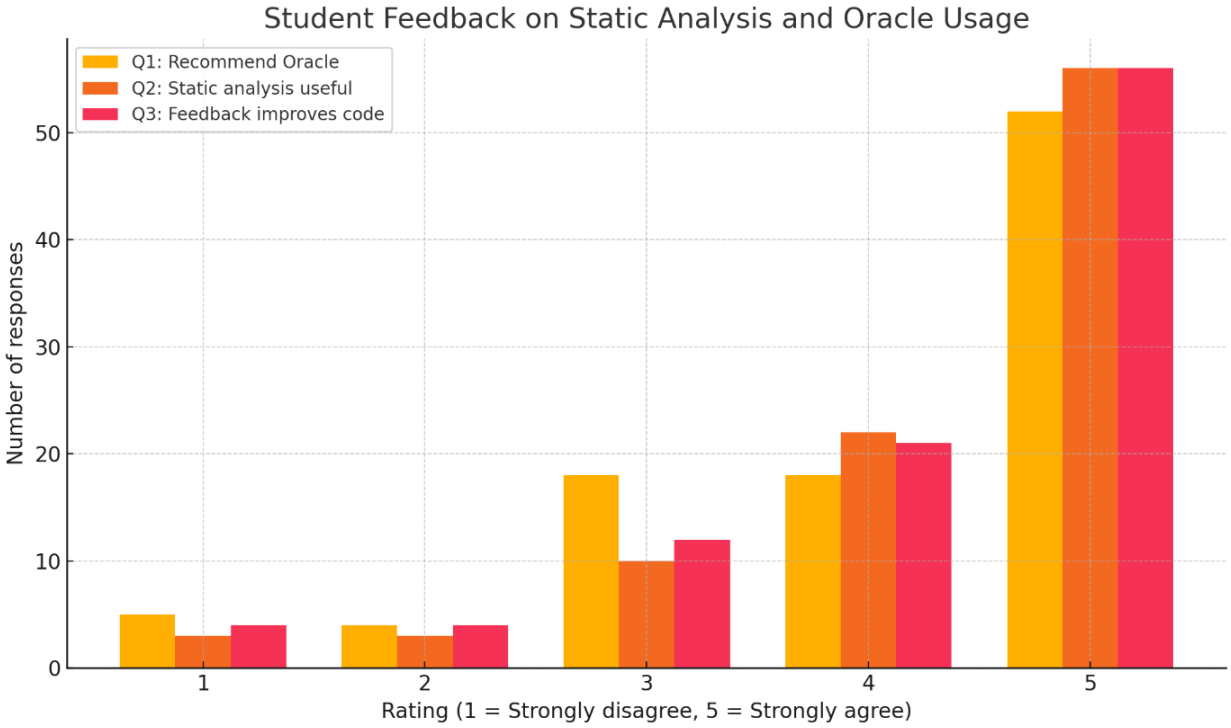


Figure 4. *Distribution of student responses to three evaluation items concerning the impact of Oracle feedback and static analysis.*

4.3 Feedback from Third Respondent Group

The third group, composed of students with no prior experience with the feedback system, reviewed examples before completing the form. A strong 95% supported the idea of integrating it into the course. Around 84% believed the static analysis messages could help students write cleaner code, and most said they would act on the suggestions provided. In fact, most respondents agreed with the statement *"I would try to fix all the issues mentioned in the emails so there are none left"* (Q2), which indicates a readiness to actively engage with the feedback. The section dealing with formatting feedback was also well received. 79% found the comparisons helpful, and 84% said that following code conventions had benefited them professionally. Many respondents also indicated that *they "found the static code analysis part useful"* (Q3), reinforcing the value of static techniques even for those who had not previously encountered them. Some noted that such habits are usually only acquired later, during internships or jobs, and should ideally be introduced earlier. Furthermore, a large portion of students believed that this improvement *"would help them write code of better quality"* (Q1), which supports the idea that educational tools like these can foster professional-level awareness early in the learning process.

Suggestions from this group focused on presentation and usability. Several proposed an interactive version of the feedback, where one could click on the message and see examples or links in a side panel. Others wanted better formatting visibility, *"If I could see the old and new version side by side, it would be much easier to understand."* These observations reveal a general openness to using such a tool, paired with expectations shaped by experience with professional environments. **Figure 5** summarizes the results of all three items.

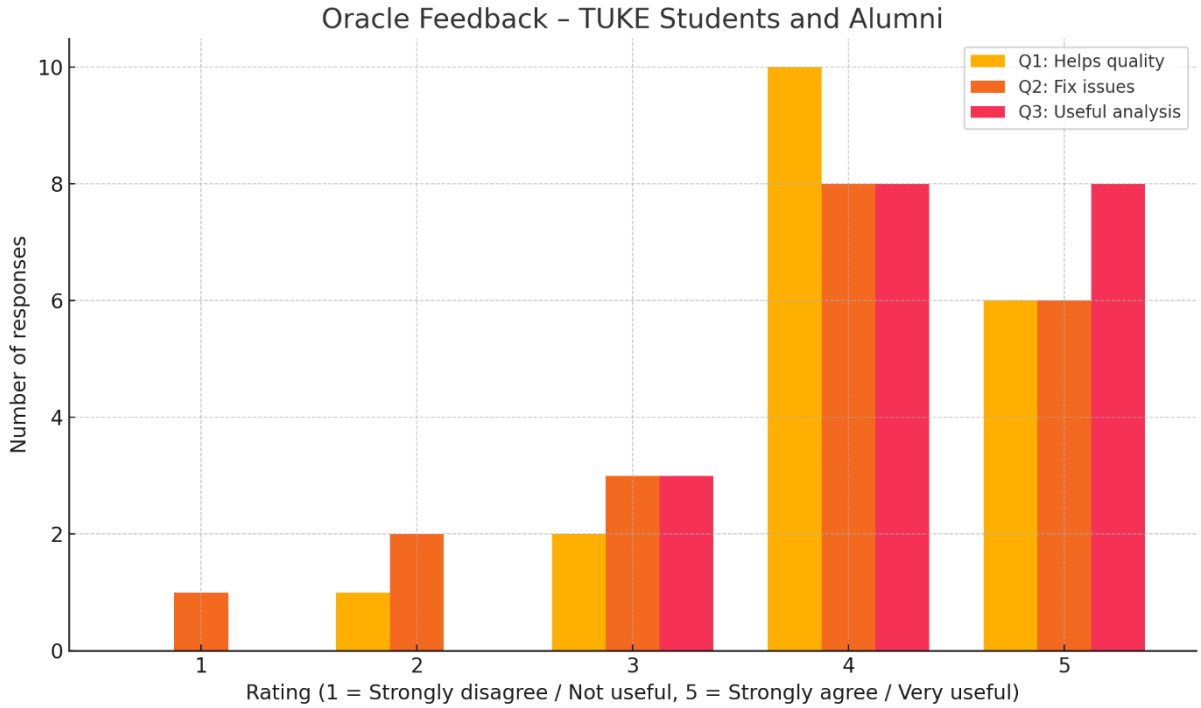


Figure 5. *Distribution of responses from TUKE students and alumni.*

4.4 Aggregated Results and Visual Summary

Across all groups, the feedback system was positively received. About 85% of respondents considered it useful and worth using regularly. The static analysis segment was mentioned by 74% of all students as helpful for learning how to improve their code. Feedback was not just accepted; it was engaged with. Students used it to discover overlooked mistakes, improve readability, and internalize conventions that are otherwise treated as secondary.

Open-ended responses emphasized the value of error explanations and actionable advice. Many appreciated that the emails did not simply say *"this is wrong"* but went further, offering concrete suggestions. Students used phrases like *"I didn't know about that rule before, but now I do"* and *"This taught me something I hadn't heard in the lectures."* Such responses show that the feedback supported learning outside of formal instruction. This is also reflected in the overall distribution of responses shown in **Figure 6**.

Formatting suggestions were rated slightly lower in clarity, though still seen as valuable. Several students said they had to read the diff output multiple times to understand what had changed, but even those who found it less clear admitted that it pushed them to reflect on their own style. Suggested improvements included better code highlighting, integration with code editors, and showing formatted output in a split view.

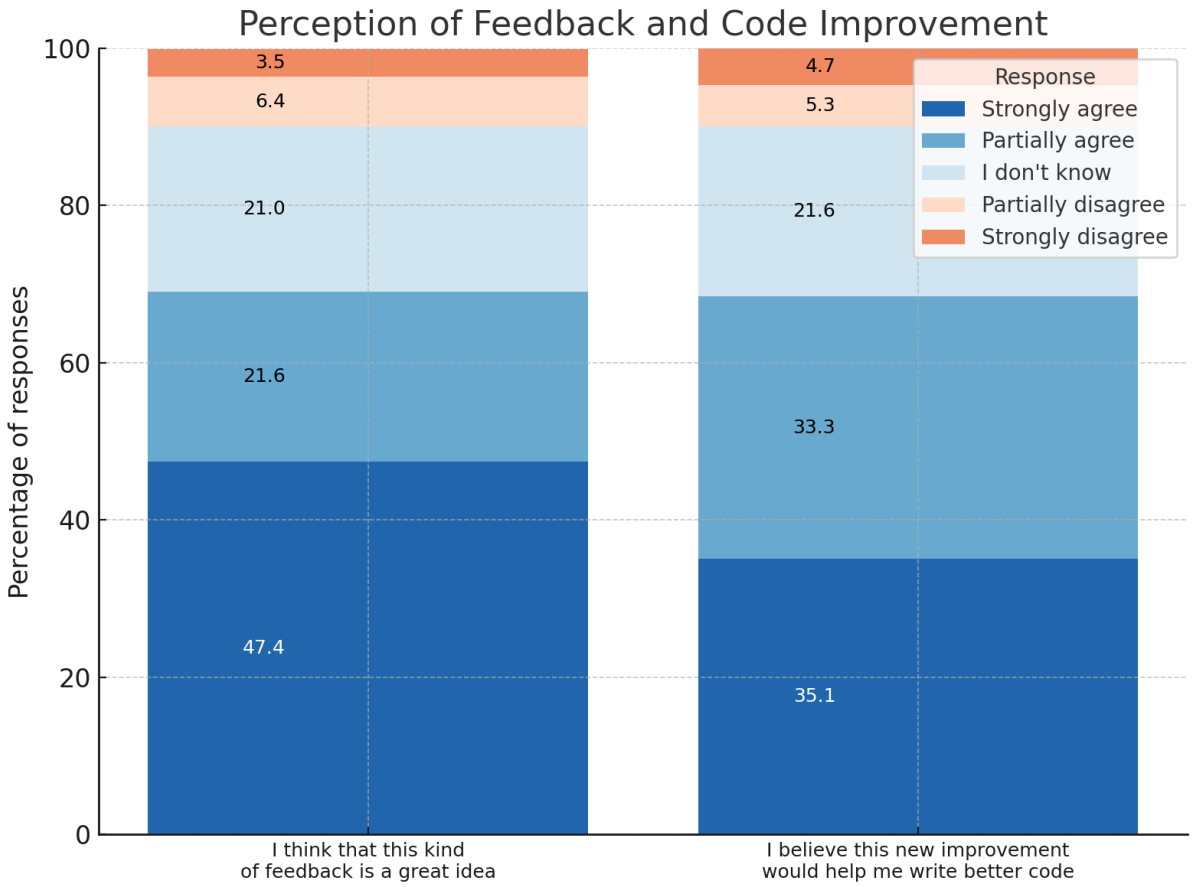


Figure 6. *Summary of Student Perceptions Regarding Feedback and Code Improvement.*

We also observed differences in how students approached the feedback. Some saw it as guidance and fixed their code in multiple iterations, while others chose to address only the most obvious or required changes. Still, there was a general trend of increased awareness. One student described the process *"It was a lot at first glance, but I tackled the easiest fixes first and then the code got much cleaner step by step."*

The system's tone and timing were also key factors. Students commented positively on the feedback arriving during the assignment window, giving them time to respond. Several mentioned that they viewed the assistant as a companion in the coding process. One described it as *"a second pair of eyes that helped me catch things I missed."*

In summary, the collected data suggests that personalized feedback based on static analysis and formatting is both effective and appreciated. It addresses a known gap in programming education, guidance on code quality, and does so in a way that students find understandable and actionable. Although there is room to improve how the feedback is displayed, especially in terms of formatting clarity, students across all backgrounds recognized its value and encouraged its continued use.

4.5 Code Quality Improvement

To provide additional insight into student progress beyond survey responses and experimental tasks, we also analyzed performance data across several distinct problem sets used in multiple academic years. These tasks vary in complexity and are designed to assess not only correctness but also aspects of code structure, readability, and coding conventions. We compared average scores between submissions where students received personalized static analysis feedback and those evaluated under standard conditions without such feedback.

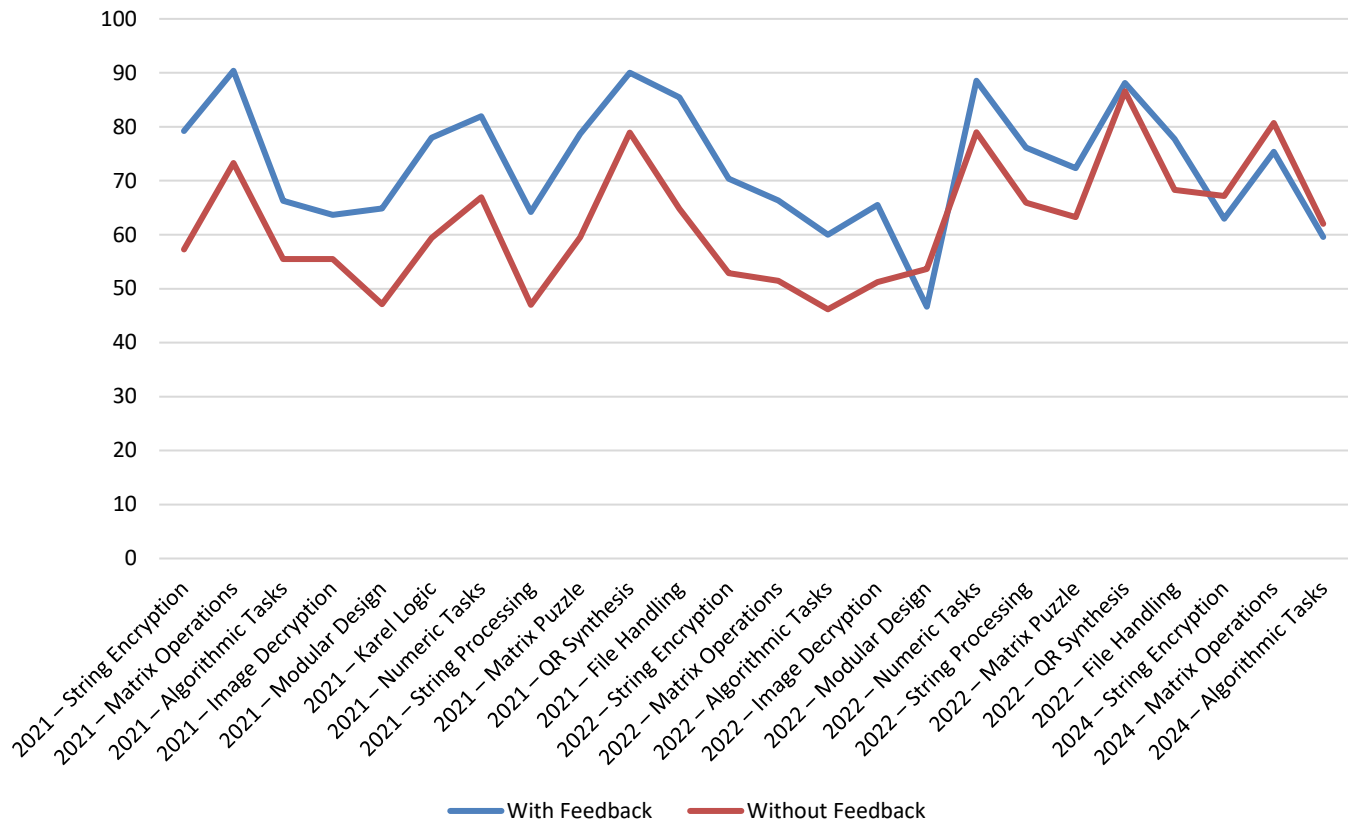


Figure 7. *Average assignment scores across selected programming tasks with and without personalized feedback.*

As shown in **Figure 7**, the results indicate that students exposed to personalized feedback consistently achieved higher scores across most tasks. This effect was especially notable in problems where structural clarity and naming conventions played a larger role. While causality cannot be definitively claimed, the repeated pattern across different cohorts and assignments supports the assumption that targeted feedback may encourage more careful coding practices and lead to improved outcomes over time. These results align with the original intention of our system, to raise awareness of code quality and not just correctness. Some problem sets, particularly in 2024, show smaller or even inverse differences, which may reflect varying task designs or external factors. Observed trend is consistent with the hypothesis that guided feedback helps reinforce clean coding habits and contributes positively to student development.

5 DISCUSSION

The findings of this study suggest that integrating static code analysis into personalized feedback can support students in improving code quality, particularly in areas that are often underemphasized, such as formatting and clarity. While correctness has long been a central metric in programming education, our results show that students are willing and able to respond to qualitative insights when those are presented in a way that is understandable and relevant to their learning stage. In the target group, students corrected a higher number of style related issues and showed greater attention to readability and structure. They did this without compromising correctness. Rather than focusing only on passing automated tests, students reflected on feedback and made iterative improvements. This was supported by submission behavior, where feedback recipients more often resubmitted their work and addressed elements that did not directly affect their grade. Several students also remarked that the messages were *"not overwhelming"* and helped them understand what good code should look like, not just how to get it working.

These findings correspond with prior work that highlights the value of targeted formative feedback in programming courses. Nutbrown and Higgins (2016) showed that static analysis can surface issues that human graders care about, but which are difficult to capture using test-based assessment. Our study extends that idea by showing that giving static analysis feedback directly to students can prompt proactive correction and reflection. The feedback system

was designed to present only the most relevant information in a friendly tone. This avoided excessive output, which in earlier studies has been shown to confuse or discourage learners. One student even described it as *"like having a mentor that watches over your code"*. The comparison between issue types also reveals a pattern in what students prioritize. Style issues were addressed more frequently, while design-related suggestions, such as refactoring complex functions, remained largely unresolved. This reflects a realistic strategy given the time constraints of assignments. Students fixed what they could, starting with the easier parts, while postponing more structural changes.

To summarize key aspects of the proposed solution, its advantages and limitations, Table 3 provides overview of current observations and potential directions. While the observations presented in this analysis indicate promising possibilities, further work is needed to evaluate long term effects, generalizability to different student, and the impact on deeper programming comprehension.

Table 3. *SWOT analysis of the personalized static analysis feedback system.*

Strengths	Encourages active learning and reflection, improves code quality beyond correctness, scalable integration into automated grading systems, minimal instructor intervention required, aligns with professional coding habits.
Weaknesses	Limited adaptation to individual skill levels, predefined rule set may not cover complex cases, occasional superficial fixes by students without full understanding, dependent on proper rule selection and configuration.
Opportunities	Integration with adaptive feedback mechanisms, application in other courses and languages, combination with AI-based coding assistants for advanced explanation generation, incorporation of pattern recognition, contextual understanding, self-learning and efficiency into educational workflows.
Threats	Risk of students developing dependency on external AI tools, potential shift towards passive coding behavior when fully automated tools dominate, balancing automation and skill development remains critical.

These insights suggest that introducing feedback earlier in the workflow, or embedding it directly into the development environment, might result in better outcomes. If students encounter suggestions while still writing their code, rather than after submission, they may be more inclined to make meaningful improvements. While modern *IDEs* and AI tools such as *GitHub Copilot* provide comprehensive real time suggestions directly during coding, their use in introductory courses raises certain pedagogical concerns. Fully automated suggestions may lead novice programmers to over rely on such systems without adequately developing their own code reasoning skills. Our approach, in contrast, intentionally delivers controlled feedback post-submission, encouraging students to reflect on their mistakes and actively engage in code revisions. At our institution, introductory programming courses are intentionally designed to expose students to minimal tooling like basic editors such as Vim, which ensures that students first master fundamental programming concepts before transitioning to more advanced integrated environments.

The role of personalization was central to this approach and appeared to enhance its effectiveness. Students appreciated the supportive tone and the tailored nature of the feedback, which was based on their recurring mistakes or ignored issues from past submissions. Rather than providing broad recommendations, the system generated specific messages that aligned with each student's needs. This likely helped them stay focused and less overwhelmed, especially compared to generic output from traditional static analysers. While personalization in this study was based on simple heuristics, such as frequency of issues or repetition, more advanced modelling could make it even more impactful. A system that tracks long-term patterns and adapts suggestions accordingly could address not only what students got wrong but also what they continue to struggle with over time. Some students indicated that they remembered the feedback and used the suggestions in later tasks. Although we did not formally measure long-term transfer, these observations point toward potential lasting benefits.

Static analysis combined with personalized delivery shows potential to enhance programming education in a way that supports learning without adding pressure. Students benefit from understanding not just whether their code works, but how it can be improved.

5.1 Institutional and Educational Implications

While the primary focus of this system was to enhance student learning outcomes, its implementation has also demonstrated several positive effects for educators and the institution. For instructors, the automated feedback system reduces the repetitive and time consuming task of manually reviewing common code quality issues such as formatting inconsistencies, naming conventions, or redundant structures. This allows teaching staff to dedicate more time to more advanced discussions with students, focusing on logic, algorithms, and conceptual problem solving rather than technical corrections that can be easily automated. Aggregated feedback data collected over multiple assignments provides instructors with a broader overview of common mistakes across the class. This insight can be used to adjust lecture content, design targeted tutorials, or clarify topics that students systematically struggle with. The system also provides benefits at the institutional level. Introducing standardized feedback on code quality helps maintain consistency across different instructors, semesters, and parallel course sections. As programming courses are often taught by multiple instructors with slightly different styles and expectations, a unified feedback system supports harmonization of course standards and learning objectives. Over time, this may contribute to more consistent development of coding habits across the student population and may reduce discrepancies in grading related to subjective style differences. From a broader perspective, adopting such systems can also serve as a step towards improving internal quality assurance processes within the institution by documenting recurring problem areas and monitoring gradual improvements in student code quality.

5.2 Study Limitations

There are, however, several limitations that should be acknowledged. The experiment involved a single course with a moderate sample size and was conducted over only two assignments. Broader testing across different institutions, programming languages, or student populations would be necessary to confirm generalizability. Also, the feedback configuration was designed for our course context, which may differ from others. Instructors elsewhere would need to adapt rule sets and message content based on local expectations. The system supports such adjustments through a configurable file, but this requires effort and familiarity with the tool. Another issue is the lack of direct evidence on whether students fully understood the advice. In some cases, we observed superficial fixes. One student removed an unused variable warning by simply printing the variable, which eliminated the warning but did not improve the program. This kind of workaround highlights a known challenge in automated feedback, it can lead to shallow compliance if not supported by instruction. A possible remedy is to occasionally review selected feedback items in lectures or tutorials, helping students see the rationale behind certain suggestions. Despite these challenges, most students reported that the feedback helped them understand programming more deeply and write better code overall. Many even requested that such tools be made part of all future programming courses. From a teaching perspective, the collected data could also serve to inform instructors about common mistakes. If a large portion of students repeatedly makes similar errors, those could be addressed during class sessions.

6 CONCLUSION

This study explored the integration of personalized static code analysis into an automated feedback system used in introductory programming education. By combining tools such as *Cppcheck* and *Clang-format* with a feedback generation layer that contextualizes technical messages in student-friendly language, we created a system capable of identifying common problems and guiding students toward better practices. The evidence gathered from both quantitative performance data and qualitative student feedback suggests that learners were not only able to improve their code iteratively.

The impact of the intervention was not limited to immediate improvements in code submissions. Many students explicitly stated that the personalized suggestions helped them recognize issues they previously ignored and encouraged them to reflect more deeply on their work. Iterative behavior, such as making additional submissions to improve code clarity even after achieving correctness, highlights that the system influenced their learning strategies.

In addition to the subjective feedback collected through surveys, we also observed a measurable improvement in students final assignment outcomes across multiple academic years. These assignments include criteria related to code clarity and maintainability, not just functional correctness. The upward trend in results following the introduction of personalized static feedback supports the assumption that such interventions can positively influence student performance.

What distinguished our feedback from more conventional static analysis outputs was the manner of delivery. By prioritizing messages based on severity, tailoring tone to student behavior, and including explanations or examples, we avoided the overload commonly associated with linters in education. The tone of the feedback was described as helpful and clear, with several students mentioning they felt more supported in the process.

Our results suggest that embedding static code analysis within student workflows can significantly enrich programming education, particularly when presented in a pedagogically mindful manner. Rather than simply relying on static tools to flag issues, systems should be designed to guide students through the reasoning process behind improvements. These findings open several opportunities for future research and enhancement:

- Develop adaptive feedback mechanisms that learn from student history and adjust both message priority and complexity over time.
- Incorporate the system into additional programming languages and course levels to evaluate generalizability.
- Improve integration with development environments to offer feedback during active coding, rather than post-submission.
- Explore the combination of static analysis with AI-based tools to generate dynamic explanations or enable follow-up questions.
- Conduct longitudinal studies to assess how early exposure to feedback impacts coding practices in more advanced courses.
- Investigate how concepts addressed by modern AI-based coding assistants such as pattern recognition, contextual understanding, self-learning, and efficiency can be incorporated into programming education while preserving student engagement with core problem solving skills.

By continuing to refine and scale these systems, educators can support student learning at both technical and conceptual levels. Personalized static analysis feedback has the potential to bridge the gap between automated testing and meaningful mentorship, making quality instruction more accessible and scalable in programming education.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding: This work was supported by project APVV-23-0408 “Evolving Architectural Knowledge in the Edge-to-Cloud Continuum”.

Conflict of Interests: The authors declare no conflict of interest.

Author Contributions: M.H.: Conceptualization, Formal Analysis, Methodology, Investigation, Validation, Visualization, Writing – original draft, Writing – review & editing. E.P.: Funding Acquisition, Project Administration, Resources, Supervision. F.G.: Data Curation, Software.

Statement on the Use of Artificial Intelligence Tools: The authors declare that no generative artificial intelligence tools were used for the creation of textual or visual content in this article. Language models (such as ChatGPT) were employed solely for minor editorial purposes, such as improving grammar, style, and clarity of sentences originally written by the authors. All conceptual, methodological, and analytical content was developed entirely by the authors.

Data Availability: The data that support the findings of this study are available from https://doi.org/10.5281/zenodo.16939667.

REFERENCES

Babati, B., Horváth, G., Májer, V., & Pataki, N. (2017). Static analysis toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics*, (pp. 23–29). Eszterházy Károly Catholic University. https://doi.org/10.14794/ICAI.10.2017.23

Biňas, M., & Pietriková, E. (2022). Impact of virtual assistant on programming novices' performance, behavior and motivation. *Acta Electrotechnica Et Informatica*, *22*(1), 30–36. https://doi.org/10.2478/aei-2022-0005

Bubenkova, L., Pietrikova, E., & Horvath, M. (2025). Code Reuse and Good Clones in Programming Education. In *2025 IEEE 23rd World Symposium on Applied Machine Intelligence and Informatics*, (pp. 401–406). IEEE. https://doi.org/10.1109/sami63904.2025.10883291

Chen, H., Nguyen, B., & Dow, C. (2022). Code-quality evaluation scheme for assessment of student contributions to programming projects. *Journal of Systems and Software*, *188*, 111273. https://doi.org/10.1016/j.jss.2022.111273

Copeland, T. (2005). *PMD Applied*. Centennial Books.

Cooper, K. D., & Torczon, L. (2023). Chapter 9: Data-Flow Analysis. In *Engineering a Compiler*, (3rd ed., pp. 449–516). Morgan Kaufmann.

Delev, T., & Gjorgjevikj, D. (2017). Static analysis of source code written by novice programmers. In *2022 IEEE Global Engineering Education Conference*, (pp. 825–830). IEEE. https://doi.org/10.1109/educon.2017.7942942

Fehnker, A., & Blok, T. (2017). Automated Program Analysis for Novice Programmers. In *Third International Conference on Higher Education Advances*, (pp. 1–8). Polytechnic University of Valencia. https://doi.org/10.4995/head17.2017.5533

Fosdick, L. D., & Osterweil, L. J. (1976). Data flow analysis in software reliability. *ACM Computing Surveys*, *8*(3), 305–330. https://doi.org/10.1145/356674.356676

Gallien, T. & Oomen-Early, J. (2008). Personalized Versus Collective Instructor Feedback in the Online Courseroom: Does Type of Feedback Affect Student Satisfaction, Academic Performance and Perceived Connectedness With the Instructor?. *International Journal on E-Learning*, 7(3), 463–476.

Horváth, M., & Gurbál, F. (2024). Code Clones: A Novel Approach to Detecting Plagiarism in binary decomposition of C Programs. *Acta Electrotechnica Et Informatica,* 24(2), 13–18. https://doi.org/10.2478/aei-2024-0006

Horváth, M., Kormaník, T., & Porubän, J. (2024). Adaptation of Automated Assessment System for Large Programming Courses. In *5th International Computer Programming Education Conference,* (pp. 4:1–4:11). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.ICPEC.2024.4

Jurado, F. (2021). Teacher Assistance with Static Code Analysis in Programming Practicals and Project Assignments. In *2021 International Symposium on Computers in Education (SIIE).* IEEE. https://doi.org/10.1109/siie53363.2021.9583635

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024). Automated Grading and Feedback Tools for Programming Education: A Systematic review. *ACM Transactions on Computing Education*, *24*(1), 1–43. https://doi.org/10.1145/3636515

Molnar, A., Motogna, S., & Vlad, C. (2020). Using static analysis tools to assist student project evaluation. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, (pp. 7–12). ACM. https://doi.org/10.1145/3412453.3423195

Neamtiu, I., Foster, J. S., & Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, *30*(4), 1–5. https://doi.org/10.1145/1082983.1083143

Novak, M., & Binas, M. (2011). Automated testing of case studies in programming courses. In *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, (pp. 157–162). IEEE. https://doi.org/10.1109/iceta.2011.6112606

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

Pereira, J. D., & Vieira, M. (2020). On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *2020 16th European Dependable Computing Conference*, (pp. 97–102). IEEE. https://doi.org/10.1109/edcc51268.2020.00025

Rocha, H. J. B., De Azevedo Restelli Tedesco, P. C., & De Barros Costa, E. (2023). On the use of feedback in learning computer programming by novices: a systematic literature mapping. *Informatics in Education*. https://doi.org/10.15388/infedu.2023.09

Truong, N., Roe, P., & Bancroft, P. (2004). Static Analysis of Students' Java Programs, In *ACE '04: Proceedings of the Sixth Australasian Conference on Computing Education*, (pp. 317–325). ACM.

Acta Informatica Pragensia is published by the Prague University of Economics and Business, Czech Republic | eISSN: 1805-4951