A Cross-Course Methodology for Studying Unproductive Behaviors in Programming Education

Ján Skalka[1[0000-0003-2211-2794]] Matúš Valko[1[0009-0007-4305-6943]] Cyril Klimeš[2[0000-0001-9545-5195]]

[1] Faculty of Natural Sciences and Informatics, Constantine the Philosopher University in Nitra, Slovakia
[1] Faculty of Business and Economics, Mendel University in Brno, Czech Republic
jskalka@ukf.sk

Abstract. Automated programming environments collect detailed process data that can reveal unproductive learning behavior not visible in final scores. This paper presents a cross-course methodology for analyzing such behavior through two task-sensitive indicator families: rapid guessing in closed-ended programming-related items and code-submission anomalies in programming assignments. The study was conducted on data from 297 students who participated in four programming courses, comprising 603,089 attempt records, 379 closed-ended tasks, and 591 programming tasks. Rapid guessing was identified using response-time-and-accuracy methods, with CUMP-DOWN as the primary method and RTRA-DOWN for verification. Students were clustered by reading speed and first-attempt success to support cluster-aware RGB analysis. Code-submission anomalies were analyzed through the first correct program version, using task-relative indicators based on cumulative time to correctness and comment density. Results showed that RGB was concentrated mainly in the fast-reading + low-success profile. Code-submission anomalies were rarer and more task-dependent, with the combined fast-and-comment-rich indicator providing the most cautious signal. The combined analysis showed moderate but positive overlap between both indicator families, supporting a differentiated framework rather than a single general risk score.

Keywords: rapid guessing; programming education; learning analytics.

1 Introduction

Digital programming environments and automated assessment systems allow educators to examine not only whether students solve a task correctly, but also how they interact with the learning material before producing a response or submitting a solution [1], [2]. These systems collect rich process data, including code submissions, timestamps, and assessment results, which are valuable for learning analytics, performance prediction, and pedagogical intervention [3], [4], [5]. At the same time, the increasing automation of programming education also introduces new challenges [6], [7]. When students work in digital environments, unproductive or harmful behaviors may

become easier to perform, repeat, or hide [8], [9]. Students can answer closed-ended items with minimal effort or use generative AI tools to produce code without sufficient understanding. These behaviors are not always visible in final scores alone, but they may leave traces in response times, submission histories, code evolution, and content of submitted programs [3], [10], [11]. This complexity creates a need for a differentiated analytical framework that can connect different forms of behavioral evidence to the task contexts in which they arise.

In automated programming education, harmful or unproductive behavior does not appear in one uniform form [12], [13]. It depends on the task type, the course level, and the type of interaction the system records [7]. A closed-ended knowledge-check item, an introductory programming exercise, and an advanced programming assignment each produce different evidence about student behavior [14], [15]. Therefore, the same analytical logic cannot be applied equally to all task types.

Although programming education primarily aims to develop students' ability to write, test, and reason about programs, many learning activities do not require immediate code production [16], [17]. This statement is especially true in microlearning and formative assessment, where closed-ended programming-related items can help students acquire or verify conceptual understanding. Such items may ask students to predict the output of a short code fragment, identify a correct syntactic form, choose an appropriate programming construct, or recognize the effect of a small algorithmic step [18]. Therefore, multiple-choice questions with a single correct answer are used not as substitutes for programming assignments, but as complementary activities that capture specific aspects of programming understanding [19], [20].

In this task context, the most relevant behavioral evidence is the relationship between response time and correctness [21]. The recorded response time allows analysis of whether students engage with the question long enough to plausibly read, understand, and solve it [22], [23]. When answers are submitted extremely quickly, especially in repeated patterns, this may indicate **rapid guessing behavior** (RGB) [8], [13]. In response-time research, student responses are often interpreted in terms of the distinction between solution behavior and rapid-guessing behavior. Solution behavior denotes responses produced after sufficient time for reading, cognitive processing, and item solving [9]. Rapid-guessing behavior denotes responses submitted below a plausible time threshold, where the observed answer is more likely to reflect minimal effort than actual knowledge [19], [24].

Programming assignments with automated verification provide a different type of behavioral evidence [25], [26]. Automated assessment systems are widely used in programming education because they allow students to practice repeatedly, receive immediate feedback, and solve tasks independently outside direct teacher supervision [1], [27], [28]. This scalability is pedagogically valuable, but it also creates conditions in which students may focus on obtaining a correct answer rather than engaging in the reasoning process through which the answer should be constructed [1]. In such settings, unproductive behavior may appear not as an incorrect final solution, but as a weakened or bypassed process of solution development [7], [29], [30].

In programming tasks, students are expected to construct code gradually, test or revise it, and move through a visible process of developing a solution [31], [32].

Automated systems can record submission histories, timestamps, code length, assessment results, and changes between consecutive versions [1], [33]. These traces make it possible to examine whether a solution emerges incrementally or whether it appears through abrupt code growth with little visible intermediate development. In the current educational context, such patterns are particularly relevant for analyzing over-reliance on GenAI tools, where a complete or nearly complete solution may be inserted without the gradual construction process expected from a beginner. Such submissions may also contain code-level traces that are atypical for introductory work, including unusually extensive comments, explanatory annotations, or a mismatch between the sophistication of the submitted code and the visible process through which it appeared in the system [11], [34].

Therefore, code-submission process indicators do not depend on the final code alone [11]. Their value lies in relating the submitted code to the process by which it entered the system [31], [35]. Large jumps in code length, few intermediate revisions, short development trajectories, unusually comment-rich code, or a mismatch between the complexity of the final solution and the visible process history can indicate that the learning process was reduced or bypassed, even when the submitted program is formally correct. Comment-related features are therefore best treated as supplementary indicators and interpreted together with process-level evidence rather than as standalone signals.

The present study organizes behavioral evidence into two indicator families:

- rapid guessing indicators in multiple-choice items, based on response-time and correctness patterns;
- code-submission anomaly indicators in programming assignments, based on time to first correct submission and comment-related code features.

Together, these families form a task-sensitive framework for analyzing harmful or unproductive behavior in programming education.

The core methodological aim is to examine how these indicators can be operationalized and compared across courses and task types. The framework links each behavior to the type of evidence that can most clearly reveal it, enabling the study of indicator transferability across programming languages, course levels, and task formats while preserving the specific nature of each educational activity.

2 Methodology

The study is designed as a descriptive multi-course case study. It aims to examine how indicators of harmful or unproductive learning behavior can be operationalized and compared across programming courses, programming languages, and task types.

The methodology follows a task-sensitive design. Each indicator family is applied only to the type of educational activity that produces suitable behavioral evidence. Rapid guessing indicators are applied to closed-ended items. Code-submission anomaly indicators are applied to programming assignments with recorded verification attempts and sufficient task-level baseline data.

All course data are organized into a common event-based structure. Each event is linked to a student, course, task, timestamp, task type, submitted response or code, and assessment result.

2.1 Rapid guessing

Rapid guessing is analyzed in closed-ended programming-related items with one correct answer, using response time, correctness, student identifier, item identifier, and item length. The analysis distinguishes solution behavior from rapid-guessing behavior: solution behavior assumes sufficient time for reading and cognitive processing, while rapid guessing refers to responses submitted too quickly to indicate meaningful engagement with the item.

RGB is identified using response-time-and-accuracy methods derived from previous microlearning work [16], [29]. For each item, first responses are divided into one-second response-time intervals. In each interval, the observed accuracy is compared with the chance level determined by the number of answer options. For example, an item with four options has a chance level of 25%. Intervals in which accuracy remains close to or below the chance level indicate likely guessing, while the point where accuracy rises above chance is used to estimate the response-time threshold separating rapid guessing from solution behavior.

To account for differences in student reading behavior, reading speed is estimated separately from textual microactivities. Time-on-task for these items is normalized to 100 characters, and the resulting course-level median reading speed is used along with first-attempt performance to group students before RGB threshold estimation. Students are then grouped according to speed- and performance-related characteristics, allowing RGB thresholds to be estimated within more homogeneous student groups rather than only on the aggregated dataset.

The study uses two top-down methods, RTRA-DOWN and CUMP-DOWN [29]. The primary identification method in this study is CUMP-DOWN, which determines the RGB threshold cumulatively from longer to shorter response-time intervals. RTRA-DOWN is used as a verification method to assess the stability of the obtained classifications. The resulting indicators are computed at the student–item level and aggregated within each course. The main outputs are the number of responses classified as RGB and the RGB ratio, defined as the proportion of valid first responses identified as rapid guessing.

2.2 Code-submission anomaly

Code-submission anomaly indicators are analyzed in programming assignments with recorded verification attempts and sufficient numbers of correct solutions for task-level comparison. The analysis focuses on the first correct program version submitted by each student for each task. This level of analysis allows the submitted program to be interpreted relative to the behavior of other students solving the same task.

The cumulative time to the first correct version is computed from recorded attempt times. If the recorded task activity time decreases between consecutive attempts of the

same student on the same task, the decrease is treated as a session reset, and the previous segment is added to the cumulative solution time. For each first correct program, the total number of source-code characters without whitespace and the number of comment characters without whitespace are computed. The comment ratio is then calculated from these values.

Because tasks differ in length and difficulty, the indicators are interpreted relative to the same task. For each course-task pair, task-level baselines are computed from all students who correctly solved the task. The 10th percentile of cumulative time defines unusually fast first correct submissions, and the 90th percentile of comment ratio defines comment-rich submissions. Their combination identifies the first correct programs that are both unusually fast and unusually comment-rich.

For each course, the main outputs are the counts and proportions of unusually fast first-correct submissions, comment-rich submissions, and combined fast-and-comment-rich submissions.

3 Case study

The case study was conducted on a cross-course dataset, restricted to 297 students who participated in all four analyzed programming courses from the microlearning Priscilla system [5]. The dataset contained 603,089 attempt records across 970 course-task instances. The analyzed tasks included 379 closed-ended single-answer tasks with 112,640 attempt records and 591 programming tasks with 490,449 attempt records. In addition to attempting data, the study used log-based exposure records to estimate time spent on learning objects. After filtering to the same student population, the log dataset contained 465,753 valid exposure records across 2,546 course-task instances.

From this overall dataset (Table 1), separate analytical subsets were derived. Rapid guessing analysis used first responses to closed-ended tasks. Reading-speed estimation used log-based exposure times across textual learning items, combined with cleaned task content length. Programming-task attempts were retained for code-submission anomaly analysis.

Table 1. Dataset description.

Course	Respondents	Attempt records	Closed-ended tasks	Closed-ended attempts	Programming tasks	Programming attempts
Java I	297	165,309	71	20,135	225	145,174
Java II	297	96,615	177	36,689	69	59,926
Python I	297	215,693	91	39,198	220	176,495
Python II	297	125,472	40	16,618	77	108,854
Total	**297**	**603,089**	**379**	**112,640**	**591**	**490,449**

3.1 Rapid guessing

Rapid guessing analysis was performed on closed-ended single-answer tasks. For each student, course, and task, only the first valid response was retained, because later attempts may already be influenced by feedback or prior exposure to the item.

Each retained response contained the student identifier, course identifier, task identifier, response time, score, maximum score, and timestamp. Correctness was derived from the relation between score and maximum score.

Reading-speed estimation was based on textual microactivities, i.e., short content-oriented learning items that provide micro-information students are expected to read before answering related tasks. The HTML content of these items was cleaned before length calculation: HTML tags were removed, HTML entities were decoded, and embedded images were replaced with short text to avoid distorting the character count. For each valid exposure record, time-on-task was normalized to 100 characters. The student's course-level reading speed was then defined as the median of these normalized values across textual items. This procedure produced one reading-speed value for each student in each course. In the *Python II* course, textual microactivities were largely replaced by interactive, web-based activities with code execution. Since time-on-task in these activities reflects both reading and code experimentation, the reading-speed profile from the corresponding *Python I* course in the same semester was used as a semester-matched proxy.

Student clustering was performed only on user-course records that satisfied the minimum data requirements for both reading-speed estimation and first-attempt performance estimation. Candidate K-Means solutions with $k = 3$, $k = 4$, and $k = 5$ were evaluated separately for each course. The four-cluster solution was retained because it matched the interpretation used in previous RGB work, combining reading speed and performance into four profiles: fast/high-success, fast/low-success, slow/high-success, and slow/low-success [29]. Although $k = 3$ achieved slightly better separation in some courses, $k = 4$ showed sufficient stability, with mean ARI values between *0.880* and *1.000*, and produced acceptable cluster sizes across all courses.

The cluster-aware RGB analysis showed that rapid guessing was not evenly distributed across student profiles (Table 2). Across the clustered courses, the highest RGB ratios were consistently found in the **fast reading + low success** group. Using CUMP-DOWN, this group reached 43.6% in course *Java I*, 56.5% in course *Java II*, 24.9% in course *Python I*, and 51.3% in course *Python II*. The same group also had the highest rapid incorrect ratio, indicating that many of these fast responses were not only quick but also incorrect.

Table 2. RGB ratios and rapid incorrect responses for the highest-risk students across courses.

Course	Cluster	RGB ratio CUMP-DOWN	RGB ratio RTRA-DOWN	Rapid incorrect ratio
Java I	fast reading + low success	43.6%	36.6%	33.7%
Java II	fast reading + low success	56.5%	48.8%	43.9%
Python I	fast reading + low success	24.9%	14.4%	17.3%
Python II	fast reading + low success	51.3%	45.6%	41.5%

When aggregated across clustered courses, the fast-reading + low-success profile remained dominant. It had an overall CUMP-DOWN RGB ratio of 45.1%, compared with 15.5% for slow reading + low success, 8.0% for slow reading + high success, and 7.6% for fast reading + high success. RTRA-DOWN produced lower values but retained the same ordering of student profiles.

3.2 Code-submission anomaly indicators

The second indicator family focused on programming submissions. For each student, course, and programming task, the first correct program version was selected. The time to the first correct version was computed from the recorded attempt times.

For each first correct program, two code-level measures were computed: the total number of source-code characters without whitespace and the number of comment characters without whitespace. The comment ratio was then calculated from these values. Because tasks differ in length and complexity, all indicators were interpreted relative to the same task. For each course-task pair, task-level baselines were computed from all students who solved the task correctly.

Three task-relative indicators were then derived: unusually fast first-correct solution, comment-rich first-correct solution, and their combination. The combined indicator was treated as the most informative signal because it captures both the compressed solution time and the unusually high comment density in the first correct version. These indicators are interpreted as code-submission anomaly signals, not as direct evidence of GenAI use (Table 3).

Table 3. Code-submission anomaly indicators by course.

Course	Student-task cases	Fast first correct	Comment-rich	Fast + comment-rich	Students with fast + comment-rich
Java I	39,041	10.5%	12.0%	1.6%	50.0%
Java II	10,675	10.6%	15.2%	1.8%	25.6%
Python I	37,827	10.4%	13.8%	2.1%	69.4%
Python II	16,933	10.5%	7.3%	1.2%	22.8%
Total	**104,476**	**10.5%**	**12.2%**	**1.7%**	**85.2%**

The fast-first-correct indicator was close to 10% in all courses, reflecting its task-relative percentile definition. Comment-rich submissions varied more strongly across courses, ranging from 7.3% in *Python II* to 15.2% in *Java II*. The combined fast-and-comment-rich indicator occurred in 1.2% to 2.1% of analyzed student-task cases. Across all four courses, 253 of 297 students had at least one such case, while 151 students showed the combined signal in at least two courses, and 69 students in at least three courses.

The code-submission anomaly indicators were computable across all programming courses, but their interpretation is more context-dependent than RGB. Timing alone is insufficient, as it is defined as a task-relative percentile. Comment density alone is also insufficient because commenting style varies across tasks and courses. The combined

indicator provides a more cautious signal by identifying first correct solutions that are both unusually fast and unusually comment-rich

3.3 Combined indicator comparison

The final step combined the RGB and code-submission anomaly profiles at the student–course level. The RGB branch is represented by the CUMP-DOWN RGB ratio, and the code-submission branch by the fast-and-comment-rich first correct solution ratio.

RGB was more frequent than the combined code-submission anomaly in all courses (Table 4). The overall RGB ratio ranged from 5.4% in course *Python I* to 18.0% in course *Java II.* The fast-and-comment-rich indicator ranged from 1.2% to 2.1% of the analyzed programming-task cases.

Table 4. Combined behavioral indicators by course.

Course	RGB ratio	Fast + comment-rich ratio	Users with code anomaly
Java I	13.2%	1.6%	49.8%
Java II	18.0%	1.8%	24.9%
Python I	5.4%	2.1%	69.4%
Python II	15.0%	1.2%	22.5%

The association between the indicators was positive in all courses. Spearman correlations between RGB ratio and fast-and-comment-rich ratio ranged from 0.183 to 0.270.

The overlap analysis showed that 8.8% to 9.9% of students were simultaneously in the high-RGB and high-code-anomaly groups within a course (Table 5).

Table 5. Overlap between high-RGB and high-code-anomaly groups.

Course	High RGB + high code anomaly	High RGB only	High code anomaly only
Java I	9.4%	15.5%	15.5%
Java II	8.8%	15.8%	16.2%
Python I	9.8%	15.5%	15.8%
Python II	9.9%	17.1%	12.6%

Across courses, 79 of 297 students appeared in both high groups in at least one course. Repeated overlap was less frequent: 25 students appeared in both high groups in at least 2 courses, and 8 in at least 3.

4 Discussion and conclusion

The results support a task-sensitive approach to analyzing unproductive behavior in programming education. The two indicator families captured different but complementary traces of student behavior. Rapid guessing was linked to closed-ended tasks, where response time, correctness, and chance level provide direct evidence for identifying likely low-effort responses. Code-submission anomalies were more context-dependent

because they were based on task-relative timing and comment density in the first correct program version.

In the RGB branch, the clearest pattern was found in the fast reading + low success group. This group consistently showed the highest RGB ratios and rapid incorrect ratios. The results confirm that fast response time alone is not sufficient for RGB interpretation; it becomes informative when combined with weak first-attempt performance. The comparison between CUMP-DOWN and RTRA-DOWN showed the same tendency, with CUMP-DOWN producing more sensitive estimates and RTRA-DOWN more conservative ones.

The code-submission branch required cautious interpretation. Fast first correct submissions alone mainly identify the fastest solutions within each task, whereas comment density depends on language, task design, and course conventions. The combined fast-and-comment-rich indicator therefore provided the most useful signal. It may indicate compressed or externally supported solution development, but it does not identify the source of the submitted code.

The combined analysis showed positive but moderate relationships between RGB and code-submission anomalies. Some students showed both patterns, including repeated overlap across courses, while others were high in only one dimension. This result confirms that the indicators are related but not interchangeable. A single universal risk score would mask these differences; instead, indicators should be interpreted in the context of the task that produced them.

The practical value of the framework lies in course-level diagnostics. RGB indicators can reveal low-effort answering in closed-ended activities, while code-submission indicators can highlight first-correct solutions that deviate from typical task behavior. These signals can support targeted review, task redesign, and further investigation, but they should not be treated as standalone evidence of misconduct.

The main limitation is the dependence on available process data. Reading-speed estimation required suitable textual microactivities, and one course required a semester-matched proxy for reading speed. Code-submission indicators were based on verification attempts and first correct submissions, not on continuous editing traces. Future work should extend the framework with richer code-evolution data and more detailed analysis of submission histories.

Overall, the study shows that unproductive learning behavior can be operationalized and compared across programming courses, but only through differentiated indicators tied to specific task types. RGB was transferable across courses when response time, correctness, and task chance level were available. Code-submission anomalies were more task-dependent, with the combined fast-and-comment-rich indicator providing the most cautious signal. Together, the findings support a practical framework for analyzing behavioral risks in automated programming education without reducing diverse behaviors to a single general metric.

Acknowledgments

This work was supported by the *Science Grant Agency of the Ministry of Education of the Slovak Republic* under Grant 1/0667/25, by the *Nitra Cybersecurity Competence Center* under project

No. 17R05-04-V01-00003, funded by *the Recovery and Resilience Plan of the Slovak Republic*, within the call 17R05-04-V01 – *Establishment of Cybersecurity Competence Centers*, and by the *European Commission under the ERASMUS+ Programme, KA220* under Grant 2024-1-SK01-KA220-HED-000249044.

Disclosure of Interests

The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Paiva, J.C., Leal, J.P., Figueira, Á.: Automated Assessment in Computer Science Education: A State-of-the-Art Review, (2022). https://doi.org/10.1145/3513140.
2. Omer, U., Tehseen, R., Farooq, M.S., Abid, A.: Learning analytics in programming courses: Review and implications. Education and Information Technologies. 28, (2023). https://doi.org/10.1007/s10639-023-11611-0.
3. Edwards, J., Hart, K., Shrestha, R.: Review of CSEDM Data and Introduction of Two Public CS1 Keystroke Datasets. Journal of Educational Data Mining. 15, (2023). https://doi.org/10.5281/zenodo.7646659.
4. Price, T.W., Hovemeyer, D., Rivers, K., Gao, G., Bart, A.C., Kazerouni, A.M., Becker, B.A., Petersen, A., Gusukuma, L., Edwards, S.H., Babcock, D.: ProgSnap2: A Flexible Format for Programming Process Data. In: Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE (2020). https://doi.org/10.1145/3341525.3387373.
5. Skalka, J., Drlik, M.: Priscilla - Proposal of System Architecture for Programming Learning and Teaching Environment. IEEE International Conference on Application of Information and Communication Technologies. (2018).
6. Tabanao, E.S., Rodrigo, M.M.T., Jadud, M.C.: Predicting at-risk novice Java programmers through the analysis of online protocols. In: ICER'11 - Proceedings of the ACM SIGCSE 2011 International Computing Education Research Workshop (2011). https://doi.org/10.1145/2016911.2016930.
7. Tie, J., Yao, B., Li, T., Fang, H., Ahmed, S.I., Wang, D., Zhou, S.: 'Should I Give Up Now?' Investigating LLM Pitfalls in Software Engineering. ACM Trans. Softw. Eng. Methodol. (2026). https://doi.org/10.1145/3801972.
8. Wang, B., Huggins-Manley, C., Kuang, H., Xiong, J.: Enhancing Effort-Moderated Item Response Theory Models by Evaluating a Two-Step Estimation Method and Multidimensional Variations on the Model. Educational and Psychological Measurement. 85, (2025). https://doi.org/10.1177/00131644241280727.
9. Skalka, J., Valko, M.: Comparison of Methods for Identifying Rapid Guessing Behavior in Microlearning Courses. In: E-Learning and Enhancing Soft Skills: Contemporary Models of Education in the Era of Artificial Intelligence (2025). https://doi.org/10.1007/978-3-031-82243-8_4.
10. Ivanova, M.G., Eklöf, H., Michaelides, M.P.: Comparison of Threshold Identification Methods for Response Time Effort on Computerized Test Items. Journal of Applied

Testing Technology. 24, (2023).
11. Shrestha, R., Leinonen, J., Hellas, A., Ihantola, P., Edwards, J.: Codeprocess charts: Visualizing the process of writing code. In: ACM International Conference Proceeding Series (2022). https://doi.org/10.1145/3511861.3511867.
12. Svetina Valdivia, D., Rutkowski, L., Rutkowski, D., Canbolat, Y., Underhill, S.: Test engagement and rapid guessing: Evidence from a large-scale state assessment. Frontiers in Education. 8, 1–10 (2023). https://doi.org/10.3389/feduc.2023.1127644.
13. Michaelides, M.P., Ivanova, M.: Response time as an indicator of test-taking effort in PISA: country and item-type dif-ferences. Psychological Test and Assessment Modeling. 64, (2022).
14. Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: How Programmers Interact with Code-Generating Models. Proceedings of the ACM on Programming Languages. 7, (2023). https://doi.org/10.1145/3586030.
15. Nascimento, N., Alencar, P., Cowan, D.: Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency using ChatGPT. Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering. (2023).
16. Skalka, J., Valko, M., Drlík, M.: RAPID GUESSING BEHAVIOUR IDENTIFICATION IN PROGRAMMING COURSES. In: Proceedings of the International Conferences on Mobile Learning 2025 and Educational Technologies 2025 (2025).
17. Gorham, T., Majumdar, R., Ogata, H.: Analyzing learner profiles in a microlearning app for training language learning peer feedback skills. Journal of Computers in Education. 10, 549–574 (2023). https://doi.org/10.1007/s40692-023-00264-0.
18. Sankaranarayanan, R.: Microlearning as a solution to challenges in teaching and learning introductory programming courses. In: Global Perspectives on Micro-Learning and Micro-Credentials in Higher Education. pp. 257–275 (2024). https://doi.org/10.4018/979-8-3693-0343-6.ch015.
19. Welling, J., Gnambs, T., Carstensen, C.H.: Identifying Disengaged Responding in Multiple-Choice Items: Extending a Latent Class Item Response Model With Novel Process Data Indicators. Educational and Psychological Measurement. 84, (2024). https://doi.org/10.1177/00131644231169211.
20. Skalka, J., Drlik, M.: Development of Automatic Source Code Evaluation Tests Using Grey-Box Methods: A Programming Education Case Study. IEEE Access. 11, (2023). https://doi.org/10.1109/ACCESS.2023.3317694.
21. Sideridis, G., Alghamdi, M.: Identifying quality responses using an analysis of response times: the RTcutoff function in R. Frontiers in Psychology. 15, (2024). https://doi.org/10.3389/fpsyg.2024.1479249.
22. Wise, S.L., Kuhfeld, M.R.: A cessation of measurement: Identifying test taker disengagement using response time. In: Integrating Timing Considerations to Improve Testing Practices. pp. 150–164 (2020). https://doi.org/10.4324/9781351064781-11.
23. Rios, J.A., Deng, J.: Does the choice of response time threshold procedure substantially affect inferences concerning the identification and exclusion of rapid guessing responses? A meta-analysis. Large-Scale Assessments in Education. 9, 1–25 (2021). https://doi.org/10.1186/s40536-021-00110-8.

24. Wise, S.L., Kong, X.: Response time effort: A new measure of examinee motivation in computer-based tests, (2005). https://doi.org/10.1207/s15324818ame1802_2.
25. Skalka, J., Benko, L., Drlik, M., Munk, M., Svec, P.: Guidance for Introductory Programming Courses Creation Using Microlearning and Automated Assessment. In: Microlearning (2022). https://doi.org/10.1007/978-3-031-13359-6_3.
26. Winder, J., Francis, E., Staley, B., Edwards, J.: Incremental Development and CS1 Student Outcomes and Behaviors. In: ACM International Conference Proceeding Series (2024). https://doi.org/10.1145/3636243.3636253.
27. Skalka, J.: Microlearning and Automated Assessment -- A Framework Implementation of Dissimilar Elements to Achieve Better Educational Outcomes. In: Smyrnova-Trybulska, E., Kommers, P., Drlík, M., and Skalka, J. (eds.) Microlearning: New Approaches To A More Effective Higher Education. pp. 1–26. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13359-6_1.
28. Pecuchova, J., Benko, Ľ., Drlik, M.: Automated Grading of Open-Ended Questions in Higher Education Using GenAI Models. International Journal of Artificial Intelligence in Education. 35, (2025). https://doi.org/10.1007/s40593-025-00517-2.
29. Skalka, J., Valko, M.: Rapid Guessing Behavior Detection in Microlearning: Insights into Student Performance, Engagement, and Response Accuracy. IEEE Access. 12, (2024).
30. Halvoník, D., Kapusta, J., Munk, M.: Improve estimated time-on-task calculation in a Virtual Learning Environment. Interactive Learning Environments. 31, 2914–2929 (2023). https://doi.org/10.1080/10494820.2021.1913609.
31. Shah, A., Granado, M., Sharma, M., Driscoll, J., Porter, L., Griswold, W.G., Soosai Raj, A.G.: Understanding and Measuring Incremental Development in CS1. In: SIGCSE 2023 - Proceedings of the 54th ACM Technical Symposium on Computer Science Education (2023). https://doi.org/10.1145/3545945.3569880.
32. Domino, M., Shaffer, C.A.: Using a wider digital ecosystem to improve self-regulated learning. Frontiers in Education. 10, (2025). https://doi.org/10.3389/feduc.2025.1487344.
33. Kazerouni, A.M., Edwards, S.H., Hall, T.S., Shaffer, C.A.: DevEventTracker: Tracking development events to assess incremental development and procrastination. In: Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE (2017). https://doi.org/10.1145/3059009.3059050.
34. Karnalim, O., Toba, H., Johan, M.C.: Detecting AI assisted submissions in introductory programming via code anomaly. Education and Information Technologies. 29, (2024). https://doi.org/10.1007/s10639-024-12520-6.
35. Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., Koller, D.: Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. Journal of the Learning Sciences. 23, (2014). https://doi.org/10.1080/10508406.2014.954750.