Automated Grading and Feedback Tools for Programming Education: A Systematic Review

MARCUS MESSER, NEIL C. C. BROWN, and MICHAEL KÖLLING, King's College London, UK
MIAOJING SHI, Tongji University, China

We conducted a systematic literature review on automated grading and feedback tools for programming education. We analysed 121 research papers from 2017 to 2021 inclusive and categorised them based on skills assessed, approach, language paradigm, degree of automation, and evaluation techniques. Most papers assess the correctness of assignments in object-oriented languages. Typically, these tools use a dynamic technique, primarily unit testing, to provide grades and feedback to the students or static analysis techniques to compare a submission with a reference solution or with a set of correct student submissions. However, these techniques' feedback is often limited to whether the unit tests have passed or failed, the expected and actual output, or how they differ from the reference solution. Furthermore, few tools assess the maintainability, readability, or documentation of the source code, with most using static analysis techniques, such as code quality metrics, in conjunction with grading correctness. Additionally, we found that most tools offered fully automated assessment to allow for near-instantaneous feedback and multiple resubmissions, which can increase student satisfaction and provide them with more opportunities to succeed. In terms of techniques used to evaluate the tools' performance, most papers primarily use student surveys or compare the automatic assessment tools to grades or feedback provided by human graders. However, because the evaluation dataset is frequently unavailable, it is more difficult to reproduce results and compare tools to a collection of common assignments.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Applied computing** → *Computer-assisted instruction*; • **Social and professional topics** → **Student assessment**; *Software engineering education;*

Additional Key Words and Phrases: Automated grading, feedback, assessment, computer science education, systematic literature review, automatic assessment tools

ACM Reference format:
Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (February 2024), 43 pages.
https://doi.org/10.1145/3636515

Authors' addresses: M. Messer, N. C. C. Brown, and M. Kölling, Department of Informatics, King's College London Strand London WC2R 2LS United Kingdom; e-mails: {marcus.messer, neil.c.c.brown, michael.kolling}@kcl.ac.uk; M. Shi, College of Electronic and Information Engineering, Tongji University, 1239 Siping Road, 200092, Shanghai, China; e-mail: mshi@tongji.edu.cn.

1946-6226/2024/02-ART10 $15.00
https://doi.org/10.1145/3636515

1 INTRODUCTION

Most computer science courses have grown significantly over the years, leading to more assignments to grade [39]. The time window for evaluating assignments is typically short as prompt and timely feedback increases student satisfaction [11], resulting in the danger of inconsistent grading and low feedback quality.

One method for providing a grade and feedback in good time is to use multiple human graders. This approach, however, increases the chance of variation in grading accuracy, consistency, and feedback quality [5]. As a result, automatic grading tools have become widely popular, as they are able to assign grades and generate feedback consistently for large cohorts. **Automatic Assessment Tools** (**AATs**) may be used by instructors either to fully automate the marking and feedback process, or to indicate potential issues while manually assessing the submissions.

There is typically a relationship between grading and feedback when assessing student submissions for a given assignment. Formative assessment focuses on providing feedback to teachers and students to help students learn more effectively while providing an ongoing source of information about student misunderstanding [17]. Whereas summative assessments typically intend to capture what a student has learned and judge performance against some standards and are almost always graded [17]. While some AATs focus on providing only feedback for formative assessments, typically, many provide a grade and feedback for summative assessments. In this article, we will refer to providing a grade and/or feedback as an assessment unless we explicitly discuss grading or feedback exclusively.

Using AATs to grade programming assignments originated in the 1960s [27]. Traditionally, AATs have focused on grading program correctness using unit tests and pattern matching. They typically struggle, however, to grade design quality aspects and to identify student misunderstandings. Only recently have researchers begun to address these areas as well [70, 71].

As part of implementing a unit test-based AAT, instructors must provide a comprehensive test suite. It typically takes considerable effort and requires students to follow a specific structure when implementing their solutions, such as replicating the exact output being tested or using predefined class and function names. The considerable effort and specific structure often lead to instructors designing short and well-defined coursework, as it is easier to implement a comprehensive test suite for such assignments. However, some instructors prefer to incorporate large-scale project-based assignments in their courses, which are typically less well-defined to allow students to control the direction of the assignment and use their creativity, both of which can motivate students to perform well in their assignments. These open-ended assignments are usually nearly impossible to automatically assess using unit test-based approaches, as the structure and functionality of the student assignments can change, making it difficult to implement unit tests.

We conducted a systematic literature review to investigate recent research into automated grading and feedback tools. Our review offers new insights into the current state of auto-grading tools and the associated research by contributing the following:

— Categorised AATs based on core programming skills [46].
— A summary of the state of the art.
— Detailed statistics of grading and feedback techniques, language paradigms graded, and evaluation techniques.
— An in-depth discussion of the gaps and limitations of current research.

We utilised the programming skills and **machine learning** (**ML**) papers from the results of this systematic literature review to further investigate ML-based AATs by performing a meta-analysis focusing on ML-based AATs. In the meta-analysis [46], the ML papers from the current review were used as initial papers for a backward snowball search to find other ML-based AATs. After

conducting the snowball search and finalising our included papers, we categorised the ML-based AATs using the core skills discussed in Section 2, and the techniques and evaluation criteria utilised.

The format of the article is as follows: Sections 2 and 3 introduce our framework for categorising AATs. Section 4 presents existing reviews and discusses how our own work relates to and expands on this prior work. Section 5 details our methodology, inclusion and exclusion criteria and introduces the research questions addressed by this work. Section 6 presents an analysis of the selected sources and presents the results, and Section 7 discusses our findings. Finally, Section 8 summarises our conclusions and presents recommendations.

2 PROGRAMMING SKILLS

We distinguish four core criteria for assessing programming assignments based on our experience: correctness, maintainability, readability, and documentation. These criteria are manually or automatically graded by evaluating a student's source code statically or dynamically and have been active research areas within computer science education.

There are multiple facets of correctness research, including students' conceptions and perspectives of correctness [37, 56, 66], how students fix their code to improve correctness [3, 12], and assessment of correctness [22, 160, 172]. Furthermore, there has been a multitude of research into code quality, which includes readability and maintainability [8, 30, 65]. The research has included designing rubrics for grading code quality [67] and investigating code quality issues within student programs [32]. Finally, code documentation is considered one of the best practices in software development [13, 34, 58] and is widely researched within the software engineering domain, including code summarisation [25, 42], how documentation is essential to software maintenance [13] and how to prioritize documentation effort [43]. We separated code assessment into these four skills to provide a comprehensive overview of which skills are automatically graded.

The four core criteria provide our frame of reference for investigating research into AATs for evaluating student submissions:

Correctness – Evaluate whether a student has understood and implemented the tasks in a manner that conforms to the coursework specification. There are two commonly evaluated areas of correctness. The most common is correctness of functionality: testing whether a student has adequately implemented the assignment's essential features, whether correctly calculating the Fibonacci sequence, implementing `FizzBuzz`, or creating a text-based adventure game. The other is the correctness of methodology, verifying that a student has used a specific language feature, such as recursion to calculate the Fibonacci sequence, a for-loop and modulo to implement `FizzBuzz`, or polymorphism and inheritance to create an object-oriented game.

Maintainability – Investigates how well a student has implemented maintainable or elegant code. This could include how well a student has used functions to reduce duplicated code, whether the method used to solve a problem is simplistic or overly complicated or if the student has used polymorphism and inheritance to reduce class coupling.

Readability – Analyses whether a student's submission is easy to understand. While maintainable code contributes to the readability of the code, other aspects of source code can signify if the code is readable. These aspects include: following code style guidelines, meaningful naming of classes, functions and variables, replacing magic numbers with constants, and using whitespace to separate code blocks.

Documentation – Inspects the existence and quality of a student's documentation, including inline comments and docstrings. While some believe code should be self-documenting, including inline comments to explain functionality and implementation is common practice. However, these are often useless [59] or poor quality [68]. Additionally, developers must include documentation

in the form of docstrings to explain the purpose and the interactions for a specific class or function. These are typically used when other developers interact with or maintain the function [68].

3 CATEGORIES OF AATS

AATs have numerous advantages for both educators and students. They typically reduce the time it takes to assess each assignment and can be used to enhance the traditional assessment workflow and establish a real-time feedback system. The process of automatically assessing source code can be done in various ways, such as unit testing and comparing source code to an instructor's model solution or other students' correct submissions. Depending on the assignment's intended learning outcomes, an AAT may employ multiple methods to grade various core skills.

In 2005, Ala-Mutka published a "A Survey of Automated Approaches for Programming Assignments", where they categorised automatic assessment of different features into two main categories: dynamic analysis and static analysis [1].

3.1 Dynamic Analysis AATs

Dynamic analysis evaluates a running program's attributes by determining which properties will hold for one or more executions [6]. Typical methods employed by dynamic analysis AATs include using a suite of unit tests to grade the correctness of a student's submission by comparing either the printed output or return values of individual methods.

Additionally, dynamic analysis can be used to evaluate the student's ability to write efficient code or to write complete test suites [1]. To create a comprehensive test suite, instructors typically need to be skilled at creating unit tests using complex language features such as reflection and invest significant time developing and evaluating the tests. Test suites are usually given to students in one of three ways: complete access before the final deadline [98, 99, 132], no access before the final deadline [83, 96, 116, 144, 155], or access to a subset of the test suite before the final deadline [139, 159, 188]. How the test suites are presented to students depends on the assessment approach. Typically, if the assignment is formative, students will be given complete access throughout; if the assignment is summative, the students will either have no access or only access to a subset of tests. Limiting students' access to the complete set of tests or using hidden tests are typically used to limit the students' ability to game the system by hardcoding return values.

3.2 Static Analysis AATs

Static analysis tools assess software without running it or taking inputs into account [4].

3.2.1 Static Analysis Tools. Many industry-standard static analysis tools have been implemented into AATs, including linters, such as pylint[1] and cpplint,[2] CheckStyle,[3] FindBugs[4] [4], and PMD.[5] These static analysis tools typically focus on identifying or partially identifying issues, and checking maintainability, readability, and the existence of documentation.

3.2.2 Software Metrics. Software metrics are a technique for assessing code most commonly used in commercial settings that have been effectively integrated into AATs [1]. Metrics are typically used to evaluate maintainability; Halstead and McCabe created such complexity measures. Halstead [24]'s metrics include program length, comprehension difficulty, and programming effort,

[1]pylint – a Python Linter: https://pypi.org/project/pylint/
[2]cpplint – A C++ Linter: https://github.com/cpplint/cpplint
[3]CheckStyle – A style guide enforcement utility: https://checkstyle.sourceforge.io/
[4]FindBugs has since been abandoned, and replaced with SpotBugs: https://spotbugs.github.io/
[5]PMD – A static analyser focusing on finding programming flaws: https://pmd.github.io/

whereas McCabe [44]'s Cyclomatic Complexity utilises graph theory to evaluate the program's complexity based on the control flow graph. Other than complexity, metrics can be used to evaluate object-oriented design principles, such as class coupling and depth of inheritance [9].

3.2.3 Comparative AATs. Other than adapting industry tools, static analysis has been used to compare a student's submission with a model solution or to a set of applicable solutions [1], typically focusing on grading correctness. These tools use various methods to evaluate the similarity between the student's submission and the model solution(s). Source code can be written differently but implement the same functionality. To improve accuracy, AATs can convert the source code to an **abstract syntax tree** (**AST**) [72] or a control flow graph [61] to abstract away from the syntactic representation. More comprehensive AATs have multiple methods for matching or partially matching student solutions to the model solution(s). Virtual Teaching Assistant [10] incorporates four patterns to facilitate grading partial correctness: location-free, where the order of output tokens or characters or specific structure is not a determining factor in the awarded grade and location-specific, where the order of the output or the specific structure does impact the awarded grade.

Comparative approaches can also be used to generate feedback for the student. Paaßen et al. [51] use edit-based approaches on student trace data to generate next-step hints for block-based programming languages.

3.3 Machine Learning AATs

Since Ala-Mutka [1] published their review in 2005, research into applying ML to auto-grading has increased. ML AATs use various techniques to grade or provide feedback on the correctness and maintainability of a student's submission, typically utilising dynamic or static approaches. Walker and Russell [71] trained a feed-forward neural network to grade the design quality and provide personalised feedback. They converted the source code to an AST and then converted it to a feature vector as the input to the regression model that predicts the score.

To grade correctness, Dong et al. [18] implemented two ML AATs into an online judge. The first model was trained using historical training data to predict what causes failed test results. The second model implemented was a knowledge-tracing model used to predict the probability of a student passing a new problem based on previous knowledge of a programming concept.

However, the approaches of both Walker and Russell, and Dong et al. require a large ground truth dataset to train their models. A zero-shot learning approach can resolve the need for large training data. Efremov et al. [20] implemented such an approach to provide next-step hints to students where no prior historical data for a task exists. They implemented a **Long Short-Term Memory** (**LSTM**) neural network to provide a vector representation that can be used for ML of a block-based language, followed by a reinforcement learning approach for the hint policy.

3.4 Degree of Automation

Ala-Mutka [1] also defined varying degrees of automated the assessment of programming assignments. Fully automated assessment is typically used for smaller assignments where unit testing or another form of automatic grading can easily be designed and implemented, such as assignments focusing on programming language basics [1]. Large courses use fully automated grading to reduce the workload on the instructors. However, fully automated grading tools have difficulty grading large-scale assignments, graphical user interfaces, maintainability, readability, and documentation, which are typically manually graded.

It is common to use a semi-automated approach, a mix of manual and automated assessment, to minimise the instructors' workload on these large-scale assignments. Automating certain aspects

of the assessment process allows the instructor more time to grade and give feedback on areas that cannot be easily automated, including code design [1].

4 RELATED WORK

Multiple studies have reviewed the literature and existing AATs for programming assignments. These survey papers focused on literature that discussed grading assignments or feedback given to students on their work.

4.1 Grading

Most recently, Paiva et al. [53] reviewed which computer science domains were automatically assessed. They reviewed proposed testing techniques, how secure code execution is, feedback generation techniques, and how practical the techniques and tools are.

Paiva et al.'s review showed that most automated assessment research focused on multiple programming domains, including visual programming, web development, parallelism, and concurrency. Additionally, they found that the tools and techniques for assessing assignments are in one or more categories: functionality, code quality, software metrics, test development, or plagiarism. When evaluating feedback, they used Keuning et al. [33]'s feedback categories (discussed in Section 4.2). A novel feedback approach they found was using automated program repair to fix software bugs.

Additionally, Paiva et al. classified each tool into web-based platforms, Moodle plug-ins, web services, cloud-based services, toolkits, and Java libraries and analysed the application of the found techniques to the tools. Finally, they discussed the previous and future trends of the key topics; the most notable previous key topics include tool development, static analysis, and feedback.

We previously introduced Ala-Mutka [1]'s survey on automated approaches for programming assignments, where they surveyed the literature to define different types of automated assessment approaches. They discussed the benefits and drawbacks of automated assessment and encouraged careful assignment design to provide students with several practical programming tasks.

Furthermore, Ullah et al. [69] expanded upon Ala-Mutka's work by introducing a hybrid category consisting of AATs that are both dynamic and static approaches and investigating automated assessment tools released before the paper's publication in 2018. They introduce a taxonomy of approaches for both static and dynamic approaches and discuss the approach, supported languages, advantages, and limitations of existing tools.

Aldriye et al. [2] analysed a small set of existing grading systems in multiple areas, including usability, understandability of system feedback, and the advantages compared to other tools. Similarly, Nayak et al. [48] discussed the implementation and functionality of numerous automated assessment tools, and Lajis et al. [40] reviewed AATs which used semantic similarity to a model solution to grade submissions.

Douce et al. [19] conducted a literature review of the most influential AATs from the earliest example in 1960 by Hollingsworth [27], until the paper's publication in 2005, and discuss the change of AATs throughout time, from early assessment systems, to tool-oriented systems, finally to web-based tools. Whereas, Ihantola et al. [28] investigated the approaches tools from 2006 to 2010 utilised both from a pedagogical and technical point of view, including programming languages assessed, how instructors define tests and if any tools specialised in specific areas such as assessment of GUIs.

Souza et al. [64] conducted a systematic literature review of AATs between 1985 and 2013, focusing on key characteristics of AATs, including supported programming language, user interfaces, and types of verification. They categorise the tools by degree of automation, if the tool is instructor or student-centred, and if the tools are specialised, such as contents, testing, or quizzes.

4.2 Feedback

Keuning et al. [33] conducted a systematic literature review of automated programming feedback. They aimed to review the nature of feedback generated, the techniques used, the tools' adaptability and the quality and effectiveness of the feedback.

To categorise the feedback types, Keuning et al. used and built on Narciss's [47] feedback components; these include: knowledge of performance for a set of tasks, knowledge of result/response, knowledge of correct results, knowledge about task constraints, and four others. They found that the most common feedback types were knowledge about mistakes and how to proceed.

To evaluate how teachers can adapt the tools, Keuning et al. categorised the tools by the different types of input teachers can provide, including model solutions, test data, and solution templates. Finally, they investigated how authors evaluated the quality and effectiveness of the feedback or tool and found that some completed most evaluations for technical analysis, such as comparing generated grades or feedback with an existing dataset of graded work. Other evaluation techniques include anecdotal evidence, surveys, and learning outcome evaluations.

4.3 Novelty of This Review

Multiple reviews are small non-systematic reviews that hand-picked tools to assess and summarise [2, 19, 40, 48, 69]. Our review differs from these by systematically searching the literature and extracting more detail about each system.

While Keuning et al. [33] investigated feedback generation, which overlaps with half of our review, their review only considered papers up to 2015 inclusive. Thus, none of their included papers overlap with ours. Nevertheless, their results form a valid distinct comparison with ours. Similarly, Ihantola et al. [28] and Souza et al. [64] investigated automated grading systems between 2006 and 2010 and is now outdated, given the recent growth in automated assessment systems.

Paiva et al. [53] carried out a state-of-the-art review in 2022 concurrently with this work. Several of their research questions (such as code execution security) have no overlap with this review. They did investigate which aspects of programs are assessed, including quality and the techniques used to generate feedback. However, they did not cover the evaluation of these tools in detail, in contrast to our detailed examination of evaluation versus human graders. Their review focuses more on the technical aspects of the automated graders. In contrast, ours takes a more pedagogical approach to consider the automated graders' place within education, focusing on how well they grade, what they grade, and what they should be grading.

To summarise, our review introduces a new method of categorising AATs by the core skills graded, whether that be correctness, maintainability, readability, or documentation, and builds on Ala-Mutka [1]'s categories for approaches to assessing programming assignments, by introducing a category for ML AATs, which is a sub-category of both static and dynamic approaches. We also include an analysis of the evaluation techniques used and the data availability and an analysis of the techniques used to grade or give feedback on the source code. In Section 7.8, we further compare the results of our literature review to previous literature reviews.

5 METHODOLOGY

A systematic literature review is a method for locating, assessing, and interpreting all accessible data on a specific subject. They are frequently used to summarise existing evidence, identify gaps, and set the stage for new research endeavours [35]. Conducting a systematic review involves developing a review protocol, identifying research, selecting primary studies, study quality assessment, data extraction, and data synthesis.

This section discusses our review protocol, defining our research questions, inclusion and exclusion criteria, and search and screening processes. We used Rayyan [50] to aid our screening process,

using their in-built tools for automated and manual de-duplication and keyword highlighting derived from our search criteria. We did not use their paper clustering feature to aid the screening process. Section 6 discusses the outcomes of selecting primary studies, quality assessment, data extraction, and data synthesis.

5.1 Preregistration

We preregistered our study [45] with the Open Science Foundation. In the preregistration, we provided a complete description of our planned methodology, including our research questions, search terms, screening, data extraction, and data synthesis.

5.2 Research Questions

To guide our review, we used the following research questions:

- **RQ1** Which are the most common techniques for programming automated assessment tools?
- **RQ2** Which programming languages do programming automated assessment tools target?
- **RQ3** Which critical programming skills, such as correctness, maintainability, readability and documentation, are typically assessed by programming AATs?
- **RQ4** How well do the automated grading techniques perform compared to a human grader?
- **RQ5** Is the feedback generated by these techniques comparable to human graders?
- **RQ6** What are the most common methods for providing feedback on a student's programming assignment, and what areas do they address?

5.3 Search Strings

We defined two search strings, one for automated grading (Listing 1) and the other for automated feedback (Listing 2). These were defined by extracting keywords from our research questions. After finding the keywords, we extended our search string to include the keywords' synonyms. To find the variants of a keyword, we stemmed the keywords and added a wildcard character. For example, "grading" becomes "grad*". However, the stemming method yielded numerous irrelevant results (e.g., gradient) due to the common nature of our keywords. Therefore, we decided to use specific variants, such as "grade", "grading", and "grader".

We decided to include a negation statement in our search strings to reduce the number of outside sources, which eliminates sources that focus on robotics, source code vulnerability, or information and communication technology, as these are out of the scope of this review.

Our search strings have minor differences to maximise the likelihood of our benchmark papers being located in our database search. In Listing 2, "student code" and "novice programm*" were included to allow the search to return the benchmark papers [54, 55, 63]. While the benchmark papers chosen for Listing 1 did not include the terms "student code" or "novice programming" in their title or abstracts.

```
(programming OR source code) AND
(grade OR grading OR grader OR
    mark OR marks OR marking) AND
(assignment OR exercise OR assessment OR course) AND
NOT(robot* OR vulnerability OR ICT)
```

Listing 1. Grading Search String.

```
(programming OR source code OR student code) AND
(feedback OR hint) AND
(assignment OR exercise OR
    submission OR novice programm*) AND
NOT(robot* OR vulnerability OR ICT)
```

Listing 2. Feedback Search String.

Table 1. Benchmark Results

Database	Grading Search String		Feedback Search String	
	Found	Total	Found	Total
ACM DL	[19, 27, 29, 54]	4	[16, 33, 63]	3
IEEEXplore	-	0	-	0
Scopus	[19, 27]	2	[16, 23, 33, 55, 63, 73]	6
Total	**4/5**		**6/7**	

Table 2. Inclusion & Exclusion Criteria

Inclusion	Exclusion
The paper is a primary source.	The paper is not written in English.
The paper focuses on auto-grading or feedback on source code/programming assignments.	The paper is not a peer-reviewed paper.
The tool supports a textual programming language.	The paper is not accessible via university subscriptions.
The paper was published in 2017 to 2021 inclusive.	Posters and papers shorter than four pages.
	Tools that use visual/block-based programming languages.

5.4 Search Process

To locate a set of relevant primary studies, we used our search strings in the **ACM Digital Library** (**ACM DL**), IEEEXplore, and Scopus. We chose these databases since ACM or IEEE publishes most Computer Science Education resources. We used Scopus, the world's largest abstract and citation database for peer-reviewed literature, to find additional sources outside of ACM and IEEE.

Initially, we investigated other databases, specifically Google Scholar, ScienceDirect, and SpringerLink. Google Scholar and SpringerLink returned almost 400,000 results, and ScienceDirect did not support the number of boolean terms in our search strings. We could not minimise the number of results by restricting our search strings because "feedback", "mark", "assessment", and "code" are terms in many other domains, including medical and genetics publications, and these databases do not allow for filtering by domain. Thus we did not use these databases.

We implemented a benchmark using papers we found during our initial reading to validate our search strings by confirming that the search results contain known papers that we identified during our initial reading. For automated grading, we used five papers [19, 27, 29, 54, 57], and for automated feedback, we used seven papers [16, 23, 33, 54, 55, 63, 73]. The results from our benchmark can be found in Table 1 and were performed without applying our inclusion/exclusion criteria (Table 2). As shown in Table 1, IEEEXplore does not contain any of our benchmark sources, as IEEE did not publish any of our benchmark sources. Even though there are no benchmark

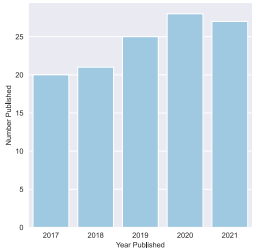

Fig. 1. The number of papers published per year.

sources, our search strings should still produce relevant results in IEEEXplore, as the search strings provide relevant sources from ACM and Scopus.

5.5 Inclusion/Exclusion Criteria

We employed the inclusion and exclusion criteria listed in Table 2 to determine whether a primary study is eligible for our review. We decided only to include sources supporting textual languages, such as Java, Python, and assembly. We only included sources from 2017 to 2021 inclusive to ensure we focused on the most up-to-date research. We omitted sources we could not access or those not written in English because we could not extract the required information. Similarly, we rejected posters and brief articles since they are unlikely to have sufficient detail. Finally, we removed non-peer-reviewed studies because the quality and veracity of these sources could not be verified.

5.6 Screening Process

We used two screeners to conduct our screening process, with any conflicts between decisions being discussed and resolved at regular intervals. We included or excluded studies based on their title and abstract for our first screening stage. In the second stage, we screened the introduction and conclusion. Finally, we screened the full-text sources and extracted data.

We changed our screening approach slightly (for the better) from our proposed approach in our preregistration [45]. We decided that both screeners would review all sources during all stages of the review, as this would increase the reliability of the results.

6 RESULTS

Figure 1 shows the general trend of automatic assessment-related papers over time. The increasing trend of research into automated grading correlates to the increasing class sizes.

Figure 2 shows an overview of the number of included and excluded papers in each stage of the review process. Our search for primary sources resulted in 1490 papers from the three databases, which after automated and manual deduplication, resulted in 1088 papers to screen by title and abstract. We excluded 820 publications based on our exclusion criteria during the title and abstract screening stage. Most of the exclusions were due to the paper's lack of focus on grading or feedback on programming assignments.

To conduct our introduction and conclusion screening, we sought the full text of 268 papers, 8 of which we could not retrieve due to a lack of access to the publication. Of the 260 papers we retrieved, 112 were excluded, and most were excluded for not focusing on grading or feedback, paper length, or focusing on visual programming languages.

In the full-text stage of our screening, we reviewed 147 articles. Out of the 147, 26 were excluded, most of which we excluded as they worked towards a grading or feedback tool rather than discussing a completed tool, leaving 121 papers to extract data from. The final included papers can be found in the supplementary material grouped by research question.

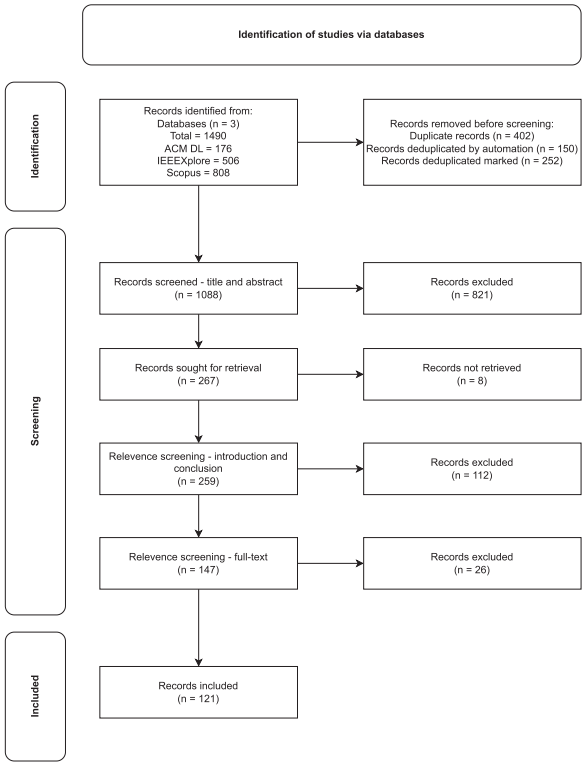

Fig. 2. PRISMA [52] diagram showing the inclusion and exclusion decisions at each stage.

We opted to annotate and analyse the papers themselves instead of extracting each tool from the articles, as the aim of this literature review is to provide an overview of current state-of-the-art research. Very few papers discussed multiple tools, and for these papers, we annotated based on all the tools discussed, which were typically within the same domain as each other. Furthermore, few tools were discussed in multiple papers, with only Antlr [78, 131, 153, 180, 189], VPL [90, 91, 125], Coderunner [101, 109, 190] and Travis [88, 121, 123] appearing in three or more papers, with Antlr and Travis primarily being underlying technologies for AATs. While there is some duplication in terms of papers discussing multiple tools and tools being discussed in multiple papers, this duplication does not overly skew the results of the literature review.

6.1 Skills Evaluated and Utilised Techniques (RQ1, RQ3, RQ6)

In this section, we present our results for the techniques utilised to automatically assess student assignments. We categorise the tools on the previously defined core programming skills as correctness, maintainability, readability, and documentation (Section 2), and the categories of AATs, including the degree of automation (Section 3).

Figure 3 shows that most of the tools focus on assessing correctness, followed by tools that grade correctness and readability or correctness and maintainability. Minimal research within our time frame has focused on grading documentation or exclusively maintainability or readability.

Figure 4 shows the combination of skills graded for each category of AAT and year. Most tools use a dynamic or static approach to assess correctness, and typically static analysis is used to grade readability and maintainability. Few papers investigate how ML can be used to assess any skill.

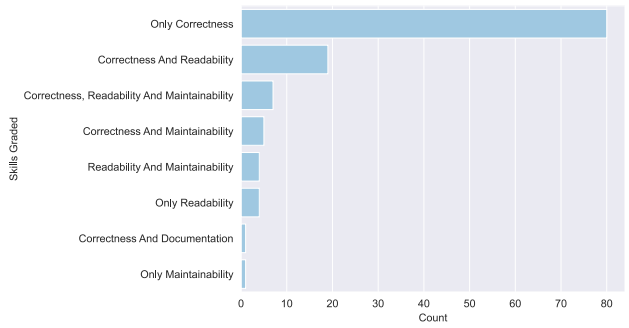

Fig. 3. The count of skills assessed, including tools that assess more than one skill.

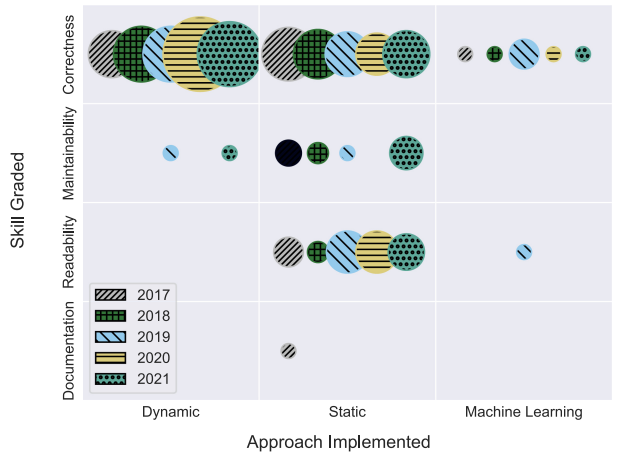

Fig. 4. The relative total count of tools categorised by year, assessed skill, and combination of approaches. The size of each bubble is the relative total count of tools categorised, with the larger the bubble the larger the number of tools categorised in research papers published in a particular year.

To grade and give feedback on the core skills, different techniques were used depending on the category of the AAT. Figure 5 shows the count of techniques used by category, as defined in Section 3. It shows that most dynamic AATs use unit testing, and most other techniques use static approaches. However, few used techniques such as ML or analysing the graphical output [155].

Figure 6 shows the count of the degree of automation, with most tools implementing fully automated graders. For a few tools, the publication was unclear on their automation approach.

A total of 81% of tools utilise a fully automated approach to assessment, while others have opted for a semi-automated approach (14%). Most AATs are implemented using a fully automated approach to allow for near-instantaneous feedback and multiple resubmissions of assignments without increasing the grading workload. In contrast, semi-automated approaches are typically used to verify grades given by an AAT or to aid a manual grading or feedback.

Most fully AATs offer bespoke solutions or implement existing grading tools, such as Virtual Programming Lab [14], to grade primarily grade correctness, typically by implementing some form of unit tests. Some tools have adapted continuous integration and delivery by using industry tools

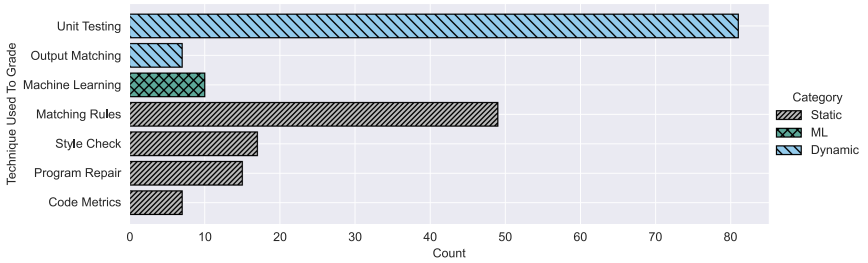

Fig. 5. The count of methods classified by the approach utilised to grade and generate feedback, as defined in Section 3. "Matching Rules" includes techniques such as comparing submissions with model solutions, previous student solutions, and use of domain-specific languages. Results with less than five occurrences are omitted.

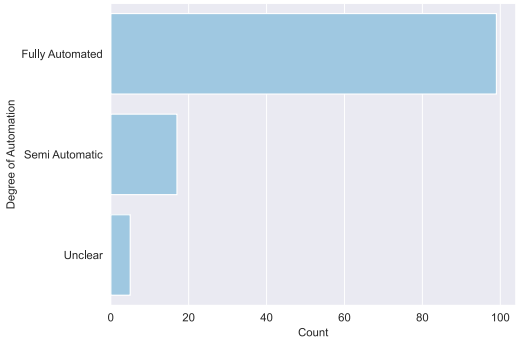

Fig. 6. The count of tools that are fully automated or semi-automated.

such as GitHub Actions[6] and TravisCI[7] to conduct unit tests when students push their code to a repository. Students typically receive the test results as feedback directly from the chosen tool.

Semi-automated assessment can be implemented using a variety of techniques. Andujar et al. [77]'s tool has instructor interactions through their grading workflow at multiple stages to grade computer graphics assignments. Instructors are expected to provide reference solutions, tests, read reports on outliers and highlight source code to produce a rubric, and finally grade the assignments based on the previous test results, extracted rubrics and highlighted code. Whereas in Insa et al. [125]'s grader, the instructors' role is to manually grade any assignments that cannot be automatically graded and correct any issues with the automatically generated test cases.

6.1.1 Correctness. A total of 66% of the research included in this review only focused on assessing correctness. Approximately 36% of AATs focused entirely on correctness and used only dynamic analysis in unit testing. The others used static analysis (17%), a combination of dynamic and static analysis (21%), ML in combination with static, dynamic analysis, or as a standalone model (6%). Most static analysis tools implement some form of comparison to a model solution or a set of existing solutions to grade or give feedback on a task. Whereas most dynamic and static analysis tools combine unit testing and comparison to other solutions.

Polito et al. [164] developed a gamified web-based automated grader for C programs. They utilise unit tests for both calculating the grade and providing specific feedback. Additionally, they award

[6]GitHub Actions – A commercially available CI/CD tools: https://github.com/features/actions
[7]TravisCI – Another CI/CD tool: https://www.travis-ci.com/

a certain amount of experience points and a badge for completing the assignment to a certain level. To evaluate their tool, they conducted a student survey and concluded that it was helpful and that users appreciated the gamification system.

Similarly, Cai and Tsai [88] developed a unit test-based tool for automatically grading Android applications. They employ an industry-standard continuous integration tool to automatically run the test suite when students push to their version control repository. The students are awarded specific points for each test case passed, with the sum being their total grade. The authors conducted a student survey and found that most spoke highly of the system, but some students found that the grading logic is not flexible enough, a known issue with unit test-based AATs.

While many automated assessment tools utilise dynamic approaches, some use a static approach. Delgado-Pérez and Medina-Bulo [105] developed a grader that uses an instructor-provided reference solution and verifications to grade a submission's correctness. The verifications allow instructors to adjust the assignment difficulty and associate the verifications with specific language elements. When evaluating their tool, they found that students' success rate improved, using the framework took less effort from the instructors, and they had a favourable reception from the students.

Dynamic and static approaches are most commonly used to automate the assessment of programming assignments. However, some tools have opted to use ML to assess the correctness of students' programming assignments. Verma et al. [184] trained a Support Vector Machine to grade assignments based on their structural similarity. They first convert the submissions to their ASTs and then replace all the identifiers with a standard character. Using these ASTs, they can calculate the structural similarity and produce a final score.

Ahmed et al. [74] and Souza et al. [177] adapted natural language processing techniques to assess source code. They normalised the source by removing comments and normalising the names and string literals. Ahmed et al. encode the source code as a binary vector, while Souza et al. trained a skip-gram model, which is a neural network that predicts a word using the context words around the missing word, to produce their vector encoding. Finally, they both train neural networks for a final task, Ahmed et al. used a dense neural network to predict how to repair an incorrect solution, and Souza et al. used a convolutional neural network to evaluate the quality of the assignments based on their underlying semantic structure.

6.1.2 Maintainability. While most AATs focus on correctness, only one tool in our review assesses exclusively maintainability. Cordova et al. [100] investigated conceptual feedback vs traditional feedback using a tool called Testing Tutor. Testing Tutor students learn how to create higher-quality test suites and improve their testing abilities. The ability to create a high-quality test suite allows the students to develop higher-quality code by finding bugs in their implementation. Additionally, creating a suite of test cases allows for future regression testing, tests you can run every time the program changes. These are a vital part of creating maintainable software [31].

Testing Tutor utilises a reference solution to detect missing test cases or fundamental concepts, such as testing boundary conditions or data integrity. This article gives feedback in either a detailed coverage report or conceptual feedback detailing a core concept that has been missed in the testing.

6.1.3 Readability. Four tools focus on assessing only readability and use a static analysis approach for their grading or feedback [127, 131, 146, 157].

Karnalim and Simon [131] implemented a tool to promote learning code quality using automated feedback. Their tool recommends improvements in a student's code and comments by providing suggestions that the student can implement.

They check that the naming has a consistent style, all names are written in `camelCase` or `snake_case`, check for misspelt subwords and validate if the name contains meaningful subwords.

Meaningful subwords should only contain letters and not contain any stop words. They implement similar suggestions for comments by checking that comments do not have any misspelt words, meaningless words, or comments that are too short. To evaluate the tool's effectiveness, they conducted two experiments. The first was conducted using a control group and an intervention group, and the second focused on a year one and a year two programming course. They conclude that while the research is incomplete, the tool can be helpful, as students do not satisfy all the code quality requirements due to human error.

Similarly, Liu and Petersen [146] extended an existing Python static analysis by developing custom checks and feedback for novice programmers. They provide a textual description of the error, provide an example, and explain why the error is problematic. To evaluate their tool, they compared two years of an introductory Python course, with one year using the tool and the previous year without. The year that utilised the authors' tool significantly reduced the number of repeated errors per submission and the number of submissions required to pass the exercise.

6.1.4 Documentation. No papers focus entirely on documentation. However, Gerdes [116] introduced AppGrader, an automated grader to grade Visual Basic applications, which graded correctness alongside documentation. They used static analysis to check if best practices have been followed and if comments exist for each subroutine or function. However, the tool did not analyse the quality of the documentation. In their bench tests, they assessed the tool against two scenarios, a typical homework assignment for an introductory programming course and the other was the source code for the tool itself, to find the overall execution times. The average overall execution time was 8.54 seconds for the typical home assignments, allowing this tool to be used as near-instantaneous feedback during development.

6.1.5 Combination Graders. While some AATs only focus on grading one skill, others aim to grade a combination of skills with varying approaches. The most common is to assess readability alongside correctness (15%). These tools primarily use a dynamic approach to assess correctness and a static approach to assess readability.

Some tools assess maintainability in addition to correctness and readability (6%) or just in addition to correctness (4%). These typically use either dynamic approaches in the form of mutation tests to assess the quality of the student-created test suites or static approaches such as metrics to calculate the complexity of the code. Few tools focus exclusively on assessing maintainability and readability (3%). These AATs use a static approach to assess both these skills.

Day et al. [104] introduced Annete, an intelligent tutor for Eclipse, to give feedback on correctness and readability. They use a neural network with a supervised learning algorithm to determine if a student needs assistance with their code and determines what feedback should be shown. Annete can give multiple types of feedback, including feedback about how to proceed, such as using language structures that should not be used in a particular assignment and practical support, such as giving positive feedback when they have passed a test case.

Similarly, Ureel II and Wallace [181] developed a tool that provides feedback on maintainability and readability during development and then used unit tests to grade correctness after the final submission. For the feedback on maintainability and readability, they utilised static analysis to find *antipatterns*, including commonly occurring practices that reflect misunderstanding, poor design choices, and code style violations in students' submissions.

While most tools provide exclusively textual feedback, Edmison and Edwards [111] have developed a tool that combines spectrum-based fault localisation with visualisation to provide feedback on maintainability and uses unit testing to grade and provide feedback on correctness. They visualise the spectrum analysis as a heatmap, with more suspicious code producing a higher score. To evaluate their tool, they conducted a user study over two semesters. One was a control group that

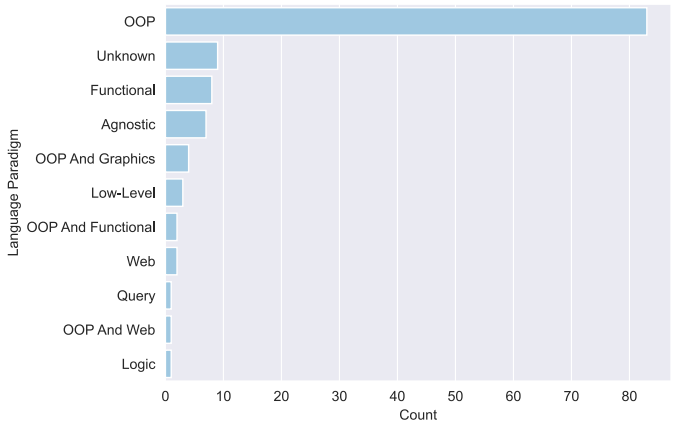

Fig. 7. The count of the language paradigms assessed, including tools that grade more than one language paradigm. "OOP and Graphics" are tools that assess the development of shaders and graphics using C/C++, and "Agnostic" includes tools that specify that any language can utilise the tool.

received only textual feedback, and the other semester provided both textual feedback and heat map visualisation. They found that having access to heat maps made it easier for students to make more incremental progress towards maximising their scores.

6.2 Languages Evaluated (RQ2)

As AATs can focus on grading any number of languages, we grouped the languages into their primary paradigm. Figure 7 shows the count of language paradigm among the tools. Most tools assess OOP languages, followed by functional languages and tools designed to grade any language.[8] Some papers did not clearly specify which language the tool assesses; these have been annotated as unknown.

6.2.1 Object-Oriented, Functional and Logical Programming Languages. Most AATs grade object-oriented languages (69%), such as Java and Python. Due to object-oriented languages' prominence in education and industry, object-oriented graders are the most developed, making practical teaching tools for the programming approaches we used [38].

Chow et al. [95] developed a tool to generate personalised hints for Python. Their tool first finds successful submissions that have failed the same test at some point and then computes an edit path from the student's incorrect submission to the successful submissions. They then translate the edit path to natural language hints. To evaluate their tool, they used a set of historical submissions and measured the percentage of submissions for which the tool can generate hints. They found that their tool could generate a personalised hint for most submissions while only using data from as little as 10 previous submissions.

While Chow et al. focused on generating personalised hints for Python assignments, other authors focused on automated grading of Java. Rump et al. [169] developed a tool to aid teaching assistants when manually reviewing submissions by generating a report assessing how well the student performed against a set of learning objectives. Similarly, Dil and Osunde [109]'s tool aimed to aid graders with grading the correctness of methodology; explicitly do students' submissions contain the required methods or fields rather than the correctness of the functionality.

[8]Non-OO or Imperative languages did not solely appear in the literature, and were paired with OO languages. Languages that can be used in an OO or non-OO way are included in the OO category.

Liu et al. [148] utilised formal semantics to develop a real-time Python AAT. They use a single reference solution to find differences in the output and execution trace of a student's submission. The student's submission is graded as incorrect if a difference is found. They evaluated their tool against a set of benchmarks of existing submissions graded by test suites and discovered that it revealed no false negatives, but a test suite did since it was missing some test cases.

Similarly, Marin et al. [153] developed a tool that uses semantic analysis, a knowledge base of programming patterns and the instructor's input to correlate the patterns with detailed natural language feedback to provide personalised feedback for Java assignments. They compared their work against state-of-the-art techniques, including Liu et al.'s tool, and concluded that their approach is based on understanding the semantics of the submissions and the original intentions of students when dealing with an assignment.

In addition to assessing object-oriented languages, some AATs assess functional languages, typically Haskell or OCaml [76, 83, 89, 113, 115, 120, 141, 147, 159, 176]. Canou et al. [89] developed an online IDE and AAT for OCaml while designing a **massive open online course** (**MOOC**). The IDE provided syntax and type error feedback as annotations and graded student submissions using unit tests.

While Canou et al. developed an IDE as part of a MOOC, Lee et al. [141] developed a program repair-based feedback tool for OCaml. They utilised test cases and correct reference solutions to find the fault and repair logical errors in students' submissions. They conducted a benchmark and student survey to evaluate the tool's efficiency and helpfulness. They found that the tool is powerful and capable of fixing logical errors in student submissions and that the students found it helpful.

Song et al. [176] developed an alternative program repair-based feedback tool for OCaml, utilising multiple partially matching solutions and test cases to generate a fix for logical errors. They compared their tool to Lee et al.'s tool and found that their approach was more effective at repairing submissions. Additionally, they conducted a user study, and students agreed that their tool was helpful.

Other tools have generated "am I on the right track" feedback for other functional languages. Feldman et al. [113] conducted a pilot study of a tool for Scheme that transformed a student's partial submission into a final program with the same functionality as a desired correct solution to determine if the student is on the right track.

Similarly, Gerdes et al. [115] developed a tool for Haskell, which used programming strategies derived from instructor-annotated model solutions to determine if a student's submission is equivalent to a model solution. They deployed their tool into their course, and most interactions with the tool were classified as correct or incorrect. Furthermore, they conducted a student survey to evaluate the perceived usefulness of the tool. They found that students were taking larger steps than the tool could handle, that the tool was sufficient, and that there was room for improvement.

Only one paper grades a logic language: Lazar et al. [138] has developed a tool to assess Prolog clauses using static analysis. To grade and give feedback on Prolog clauses, the authors convert the clauses to ASTs and extract patterns that encode relations between nodes and the program's syntax tree. These AST patterns are then used to predict program correctness and generate hints based on missing or incorrect patterns. They evaluated their approach on past student assignments and found that the tool helps classify Prolog programs and can be used to provide valuable hints for incorrect submissions. However, more work must be done to make the hints more understandable by annotating natural language explanations of their patterns and derived rules.

6.2.2 Specific Language Domains. While some AATs focus on the three major language paradigms, other AATs focus on grading more specific areas, such as web-based languages

[157, 170, 193], graphics development [77, 78, 150, 190], kernel development and assembly languages [103, 149, 161], or query languages [185].

Nguyen et al. [157] introduced a tool to grade the quality of web-based team projects and measure students' contributions. The authors utilise continuous integration tools to analyze the version control logs to determine the students' contribution and use existing static analysis tools, including SonarQube and StyleLint, to evaluate the quality of the source code. While applying this tool to a course in their department, they found a weak association between lines of code modified and the final grade. Students with better grades fixed more errors but also introduced more errors.

To assess computer graphics courses, Maicus et al. [150] and Wünsche et al. [190] developed AATs to grade OpenGL. Both solutions compared the students' output to a reference solution by comparing the difference in pixels, with Wünsche et al. also grading based on parameters passed to OpenGL and algorithm results, such as outputs of parametric equations. Maicus et al. implemented "visual unit testing", allowing instructors to script keyboard and mouse input and provide detailed feedback through screenshots and videos of the execution. They both conducted student surveys to evaluate their tools' usefulness for learning computer graphics and reported that the students found the tools helpful when learning to program computer graphics.

To assess operating system kernels Park and Kim [161] developed a cloud-based Linux kernel practice environment and judgement system. This uses a dynamic approach to test the students' attempts at programming an operating system kernel by comparing their kernel outputs to the teacher's reference solution. The authors evaluated the parallelisation of their tool by measuring the execution time when running tests and comparing these results to a baseline serial execution, with their tool taking less time to grade longer scripts. Finally, they concluded that their tool works well in the real world and reduces the effort to verify a student's work.

While Park and Kim focused on assessing kernel-level assignments, Damas et al. [103] and Liu et al. [149] developed tools to assess assembly code assignments. Both tools utilised a dynamic approach in the form of test cases to grade and give feedback on their assignments. Additionally, Liu et al. [149] also implemented an IDE plugin to give continuous feedback to students, including errors that the code cannot assemble, warnings to indicate potential bugs and information based on the analysis of the code. To evaluate their tools, they surveyed students and found that their tools were beneficial and contributed positively to the student's learning of assembly languages.

To grade SQL statements, Wang et al. [185] investigate combining dynamic and static analysis. The dynamic analysis compares the output of the students' statements with the expected value. The static analysis compares the syntax similarity using an AST and the textual similarity of the statements themselves. To evaluate their approach, they compared their hybrid approach to a dynamic analysis that executes and compares the results with the expected results, a syntax-based approach that calculates the syntax similarity between a submission and a reference solution using the AST and a text-based approach that calculates the textual similarity between the student's statement and a reference statement. They found that the existing grading approaches could not yield satisfactory results and that the hybrid approach successfully identifies various correct statements submitted by students and grades other statements according to their similarities.

6.3 Techniques Used to Evaluate the Tools (RQ4, RQ5)

Experiments were conducted using different techniques to evaluate the tool's quality. Figure 8 shows the count of techniques used in the papers, with student surveys and tools being compared to manual grading being the most common. Most of these experiments were conducted on course-specific assignments (66%). As such Figure 9 shows that most of the data is not available to validate the results or for future research. While most tools are evaluated in some form, most do not provide the dataset in which the evaluation occurs (84%); this might be due to most tools being evaluated

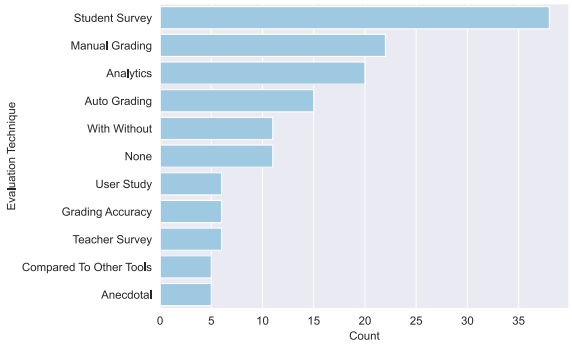

Fig. 8. The count of papers implementing these techniques to evaluate the automated grades. Results with under five occurrences are omitted.

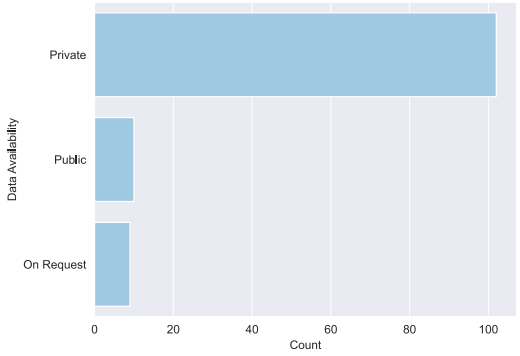

Fig. 9. The count of data availability used in the evaluation.

against course-specific assignments (66%) or exams (10%), which typically are not be distributed publicly.

6.3.1 Approaches. 90% of the papers include some form of evaluation, with most evaluations conducted by the tool developer. The primary method of evaluation is to ask for student feedback on a tool in the form of surveys (24%), as they are the primary beneficiaries of AATs. Their insight provides valuable feedback on how well a tool helps them learn how to program.

Another typical evaluation approach compares the tool to either manual grading (22%) or other automated tools (9%). The evaluation is typically used to check the tool's grade accuracy compared to manual grading. Other aspects, such as using an automated tool to improve students' grades or allow them to receive feedback faster, are also evaluated. Though most of the included articles perform some form of evaluation, 10% of them do not; this could be due to the tools still being in development and not at a stage where a full evaluation would be beneficial. However, even an initial validation that the tool is performing as expected at the early stages of the project could be beneficial to confirm that the final tool will aid students' learning.

6.3.2 Performance. As an exploratory analysis, we annotated a third of all papers that conducted some form of evaluation and how well they performed. We found that most tools perform well in their evaluations, though some authors report mixed results, especially if using multiple evaluation approaches. While the tool developers conducted most evaluations, some tools were

evaluated by third parties, typically in papers evaluating multiple tools or papers reporting authors' experiences using existing tools. Reporting only positive results makes it difficult to evaluate different tools reliably and can be challenging for instructors to select the most effective tool for their course, especially with most AATs implementing the same methodology.

Some authors have produced evaluation or experience reports discussing one or more tools to provide instructors with an external evaluation of some AATs. Reis et al. [167] evaluated two AATs, one that gave personalised hints using program repair and a program visualisation tool, to determine if students can solve problems faster, understand problem-solving and fix bugs easier, compared to a test suite. In their experiment, they had three test conditions; in the base condition, students only had access to the test suite, and in the other two, they had access to one of the tools and the test suite. The students were asked to complete a set of problems using one of the three conditions randomly assigned to the problem; after reaching a correct solution, they were presented with a post-test. This post-test was used to evaluate the student's understanding of how to solve the problem and consisted of students being presented with four different solutions for a problem and being asked to identify which solutions were correct or incorrect without being able to execute the program. They found that the program repair tool greatly cuts student effort, with fewer attempts, and students using the visualisation tool showed lower post-test performance.

Similarly, Bey et al. [84] evaluated two AATs, one implemented using unit tests and the other utilising reference solutions. They focused on applying these AATs in the context of a MOOC. They found that the reference solution-based AAT typically performs as well as the unit test-based approach. However, the reference solution-based tool awards lower grades to correct solutions that are rarely implemented.

While some evaluation papers compare existing tools, Clegg et al. [97] investigate how different test suites can provide different grades and how the properties of the unit tests impact the awarded grades. To answer their research questions, they extended an existing set of programming assignments with artificial faulty versions and a sample of test suites from a larger pool. They generated the grades by calculating the percentage of passed tests and compared the different test suites. The authors concluded that the grades vary significantly across different test suites and that code coverage, the percentage of the source code executed while running a test suite, affects generated grades the most.

6.4 Performance Against Human Graders (RQ4, RQ5)

Only 22 tools are evaluated against human assessors, with 16 focused on grading and 6 focused on feedback. While some of the automated graders provide feedback to students, typically in the form of unit test results, similarity to model solutions or predefined human feedback, human assessors have not evaluated the generated feedback. To investigate how well AATs perform when compared to humans, we further annotated papers that conducted an evaluation that included some comparison to the assessment provided by a human.

Figure 10 shows the count of tools by different evaluation techniques that involved comparing the results from the AAT with a human and the authors' sentiment of the performance of the AAT. Most are evaluated against the accuracy of the AAT compared to human-provided grades [79, 108, 165, 171, 180, 184, 185, 189], with most reporting positive results.

Wang et al. [185]'s tool uses a hybrid approach of reference solutions to automate the grading of SQL statements. It is evaluated against a benchmark of human-graded submissions and three other state-of-the-art approaches, including dynamic, syntax-based, and text-based analysis. The proposed approach performs better than other state-of-the-art approaches with a clear advantage, with a mean average error of 8.37, compared to 26.01 for static analysis, 29.89 for text-based analysis, and 21.66 for dynamic analysis.

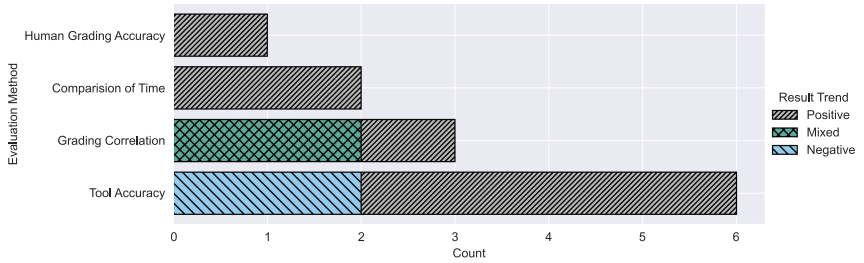

Fig. 10. The count of automated graders categorised by the evaluation technique when compared to human graders. The resulting trend is the authors' sentiment of the performance when compared to humans. In the case of human grading accuracy, the AAT was taken as the ground truth and the human graders were found to be more error-prone [126].

While Arifi et al. [79] primarily discusses automated feedback in the form of code repair suggestions, the evaluation compared to a human is focused solely on the effect the code repair has on the automated grading and not how the feedback compares to a human assessor. The evaluation shows that their code repair tool increases the grade precision, measured by comparing the submission's auto-graded result before and after code repair to manually graded scores, of their unit test-based automated grade, by 4%, from 81% to 85%. This shows that their tool improves the precision of the automated grader by repairing uncompilable code that can then be automatically graded.

Most tools that evaluate the accuracy of automated graders take the human-provided grades as ground truth. Insa and Silva [126] introduce JavAssess, a framework that automatically inspects, tests, marks, and corrects Java source code. To evaluate the performance of JavAssess to human graders, they compare the accuracy of manually and automatically graded exams. However, they treat the automatically provided grades as the ground truth and conduct an ANOVA test to validate the influence of human graders on marking errors. They concluded that human graders influence the mark when marking, with the probability value associated with the F value $Pr(> F) = 0.0164$ below the significance level of 0.05.

While most papers evaluate the tools' accuracy, some investigate the correlation between human-provided grades and automated tools [94, 98, 144, 160, 187]. Chou and Chen [94] utilise unit testing and pattern rules to automate the grading of six programming assignments within a single undergraduate course. They compared the human grades to the tool's grades to evaluate their tool. They found that the tool's grades were significantly positively correlated with human grading in all six assignments ($p < 0.001$), with on average 70% of the tool's grades being the same as the human graders.

Liebenberg and Pieterse [144] provide a detailed statistical analysis comparing manual and automated assessment of programming assignments and report on a lecturer's experience integrating automated assessment into their module. To evaluate their chosen AAT, they used 77 exams that were manually and automatically graded and were analysed using a paired T-test, and the correlation between manual and automated assessment was measured with Pearson's correlation coefficient.

Finally, the students were classified into three categories (failing, passing, and passing with distinction) based on their manual assessment marks, and a T-test was then performed on the categorical data. The T-test resulted in a "medium practically visible difference found" and can be attributed to the difference in granularity between the marks given by manual assessment and those given by unit tests and the correlation proved to be a significant relationship ($r = 0.789, p < 0.001$).

Table 3. The Automated Feedback Tools are Categorised by the Evaluation Technique when Compared to Human Graders

		Evaluation Technique				
		Accuracy of Categorisation	**Error Detection**	**Expert Agreement**	**Grading Correlation**	**Time to Bug Fix**
Result Trend	**Positive**	[119]	[182]	[169]		
	Mixed			[113]	[143]	[74]

The resulting trend is the authors' sentiment of the performance when compared to humans.

The t-test results on the categorical data show a very large, practically significant difference for failing students ($d = 1.303, p < 0.001$) and students who passed ($d = 0.151, p = 0.409$), suggesting that automatic assessment is not reliable. However, they found that automated assessment can be more trustworthy for higher achieving students, with the small effect size ($d = 0.151, p = 0.409$).

Another factor in automating grading is reducing the time instructors take to grade. Two papers investigate the effect of the time taken to grade assignments, with and without their tool [125, 155]. Insa et al. [125] introduce a semi-automated approach to grading Java assignments by automatically generating unit tests from an instructor's solution and only presenting the graders with submissions that cannot be graded using the unit tests. To evaluate how well their tool expedites the grading process, they asked graders to grade six Java exams manually and then, six months later, asked the same six examiners to mark the exams using the tool. While instructors invested an additional hour, on average, to prepare the exams, the automated tool reduced the average time taken to grade each submission from 6 minutes to 2.5 minutes. The total average time invested, including preparation, decreased by 25.2 hours, from 53.5 to 22.3 hours.

Only six papers evaluate the feedback by comparing the automatically generated feedback to human-generated feedback [74, 113, 119, 143, 169, 182]. Table 3 shows the count of tools by different evaluation techniques that involved comparing the results from the AAT with a human and the authors' sentiment of the performance of the AAT.

Ahmed et al. [74] use code repair to provide targeted examples for compilation errors. They evaluate their AAT by comparing the time taken to repair compilation errors with and without the automated code repair feedback, with both groups having access to human **teaching assistants** (**TAs**). They found that the tool helped resolve errors 25% faster on average, and the large-scale controlled nature of the empirical evaluation implies that the tools are comparable to human TAs.

Feldman et al. [113] utilises instructor-provided solutions to generate "am I on the right track" feedback. As part of a pilot study, the tool's feedback was shown to a subset of experienced TAs, and all TAs were asked to replicate what they would do when interacting with incorrect student code, and the TAs were asked to provide feedback on the tool. The TAs in the study provided mixed feedback and suggested areas of improvement, with one TA saying that the fully automated feedback features caused them to have a lack of control over the process. Other TAs saw the potential for the tool and how "am I on the right track" style feedback can aid struggling students when they do not have access to teaching staff.

Vallejos et al. [182] present a virtual TA to help teachers detect object-oriented errors in students' source code by converting source code to Prolog and inferring errors from instructor-provided rules. To validate the corrections made by their tool, they used the virtual TA to check students' coursework previously checked by human graders, to check if the tool overlooks corrections made by the instructors and to reduce the instructors' workload. The authors observe

that the tool detected 125 additional errors, totalling 196 instead of 71 object-oriented errors. However, they also found three types of object-oriented errors that the tool cannot detect, including non-static methods that do not use any field or method of the object, non-abstracted fields in sister classes and using Java Collections methods without implementing the `equals` methods. They repeated this study with additional rules to handle these missing errors, and the tool found 29 of these specific three errors overlooked by the instructor.

7 DISCUSSION

7.1 Why the Focus on Correctness Assessment? (RQ3)

There are many reasons why researchers choose to focus on assessing correctness over other important skills. Assessing the correctness of functionality teaches students how to analyse and implement features based on written requirements. Understanding how the requirements can be translated to the desired features is a crucial skill and one that is often used in the industry. Additionally, assessing the correctness of methodology allows instructors to verify that students understand particular programming concepts, such as conditionals, iteration, and recursion, or other course material, such as particular algorithms they have been asked to implement.

Primarily, assessing correctness uses a dynamic approach in the form of unit tests, which have several benefits and limitations. Unit tests allow for quickly assessing large quantities of assignments and enable students to receive near-instantaneous feedback from the test suite results. Implementing a test suite to assess an assignment requires specific unit testing knowledge, and more complex implementations require reflection knowledge. However, many unit test-based tools aid instructors in creating their test suites by providing a framework to implement them or simplifying unit testing into a set of input-output tests.

While these AATs offer near-instantaneous feedback, they are often simplistic, typically displaying if the test passed or failed, and if it failed, the difference between the expected and actual outputs or any errors. This limited feedback only informs the student of an issue in their code, either that an error is produced or that their output does not match the desired output, and does not help them resolve their issues. Providing this form of limited feedback, where the student knows the error and potential location but does not provide any hints on how to fix their issues, is similar to the limited and cryptic feedback of compiler messages, which often results in increased frustration and hampers progress [7].

Furthermore, most of these AATs cannot award partial grades for incomplete or uncompilable programs or distinguish between qualitatively different incorrect solutions, resulting in students receiving a zero grade. In contrast, if a human graded them manually, they would typically receive partial marks for source code that implements a subset of the features or has minor logical or syntax issues. Some AATs resolve this by implementing code repair to fix the broken code before running the test suite, thus providing partial grades for syntactically incorrect submissions, typically by deducting marks from the unit tests results of the repaired code [79, 85, 125, 172, 183, 186].

Another limitation of using unit testing to assess correctness is that unit tests typically require students to follow a strict pre-defined structure for their code, often with class and function names specified. Furthermore, the popularity of unit test-based AATs can influence instructors when designing their assignments. Suppose the instructors have large class sizes or want to give students near-instantaneous feedback, in that case, they often use a unit test-based AAT, typically leading to small-scale closed-ended assignments. These small-scale closed-ended assignments and the strict structure can limit students' opportunities to be creative. Additionally, this strict structure and small scale of the assignments can limit the student's ability to learn how to write maintainable and readable code by limiting their opportunities to learn how to name classes and functions and how to design object-oriented solutions.

Instructors can design open-ended larger-scale coursework to enable students to take responsibility for their assignments and use their creativity to implement the required features. Students who assume control over their learning experience gain knowledge more effectively than those who do not [36]. Furthermore, allowing students to take ownership of their projects and use their creativity can increase their motivation and learning opportunities [62].

However, these are often impractical to assess at scale, as assessing open-ended assignments is time-consuming and challenging to automate with traditional auto-grading methods. A common approach to assessing open-ended assignments is to use multiple human graders [5]. However, using multiple graders can lead to grade and feedback consistency issues, as readability, maintainability, and documentation are often subjective. While multiple graders reduce the workload for a single grader, the overall assessment process is still time-consuming. Additionally, they cannot offer the near-instantaneous feedback that fully-automated assessment approaches can offer. The impracticality of open-ended assignments often leads to instructors designing their assignments to work with traditional auto-grading approaches, limiting the opportunities for students to take control of their learning.

7.2 Why Assess Maintainability, Readability and Documentation? (RQ3)

While most tools focus on assessing correctness, few focus on assessing code quality aspects, such as maintainability, readability, or documentation. The automated assessment of these skills is typically minimal and is often focused on static analysis to determine the quality of the source code.

While evaluating these skills using static analysis can provide a good indicator, most static analysis tools and metrics are designed to evaluate professional code. Some metrics that evaluate these areas may not be suitable for all types of novice programming assignments, especially short-form assignments typically used with unit testing. This could be due to these assignments providing the overall code design, limiting the ability to use maintainability metrics such as Depth of Inheritance Tree or Coupling Between Object Classes [9].

In the case of documentation, the static analysis tools are typically limited to detecting the presence of comments and that they follow the correct format. However, recent research has started to investigate how to implement metrics commonly used in prose that can be applied to evaluating documentation. Eleyan et al. [21] use the Flesch reading ease score and Flesch-Kincaid grade level to evaluate how understandable comments and docstrings are and how long they take to read.

Most AATs that assess these skills using static analysis typically output the result, either as a number or an indication that something is missing. They do not typically tailor the output of these professional software engineering tools to novice programmers by adapting the result to something a novice programmer can use to improve. The limited information provided to the student can be confusing, limit progress, and frustrate students [15].

While assessing correctness evaluates students' ability to write working code, assessing maintainability, readability, and documentation evaluate their ability to write good code. Typically, source code that is maintainable, readable, and well-documented is easier to adapt, especially when working in teams or for future development. Furthermore, evaluating students' adherence to consistent code style is critical in teaching students to write readable and well-documented code [26].

7.3 Degree of Automation (RQ1)

There are advantages and disadvantages to both fully automated and semi-automated assessment. Fully-automated assessment allows for near-instantaneous feedback but typically limits the scope of the assignment. Providing near-instantaneous feedback can encourage students to submit early [41] and address underlying misconceptions [23].

Most fully-automated assessment tools use unit testing to assess student solutions, but they share similar issues as most unit-test-based AATs. These include forcing instructors to design closed-ended or structured assignments compatible with the tool and limiting the opportunities for students to take control of their assignments. Additionally, maintainability, readability, and documentation assessment is often non-existent or limited static analysis, such as conforming to a style guide or matching specific patterns. The typical structure used in these AATs makes it highly challenging to automatically grade elements such as correct use of object-oriented concepts, class and variable naming, and documentation quality, as these are typically provided by starter code designed to work with the AAT.

While semi-automated assessment does not allow for near-instantaneous feedback, it allows feedback to be delivered faster than manual assessment. Semi-automated assessment typically aids the grader by assessing the correctness of the submission and allowing the human grader to assess the other skills. Providing feedback quickly is essential to high student satisfaction [11]. However, the typical use of semi-automated assessment does not resolve the issues around close-ended assignments, especially providing students with a specific structure. The human grader can manually assess documentation quality, how students have named their local variables and the code style. But they can still not assess key skills like the correct use of object-oriented principles, as this is typically defined in the provided starter code.

One potential solution to automatically assess open-ended assignments, at least partially, is to automate the grading of maintainability, readability and documentation and manually grade the correctness. This semi-automated approach would allow automated grading of elements commonly shared between solutions, such as documentation quality, variable naming, adherence to code style, and code design principles. While allowing human graders to focus on assessing the implemented functionality and whether the student's solution met the requirements in the open-ended assignment. This approach would allow instructors to set open-ended assignments while offering students partial near-instantaneous feedback on skills that are rarely graded automatically. Furthermore, having the human graders only assess the correctness can reduce the overall assessment time and potentially reduce the variety in grades typically produced when multiple human graders grade the more subjective skills, including maintainability, readability, and documentation.

Additionally, further research could investigate the effect of providing the results of the automatically assessed elements near-instantaneously to the student and providing the manually assessed element after the deadline. For example, students could receive continuous feedback on their maintainability, readability, and documentation, and their final grade and feedback when the correctness has been manually assessed, providing students with feedback on areas that typically take time to grade and allowing instructors to set open-ended assignments.

7.4 Language Paradigms Graded (RQ2)

While surveys into popular programming languages[9,10,11] have yielded that JavaScript, a web-based language, is the most commonly used programming language, OOP languages are still very prominent. Besides providing fundamental skills, the popularity of OOP languages could be why they are the most commonly automatically assessed paradigm. Additionally, most OOP languages

[9] GitHub – The top programming languages 2022 (accessed 30/01/23): https://octoverse.github.com/2022/top-programming-languages

[10] JetBrains – The State of Developer Ecosystem 2022 (accessed 30/01/23): https://www.jetbrains.com/lp/devecosystem-2022/

[11] StackOverflow – Developer Survey 2022 (accessed 30/01/23): https://survey.stackoverflow.co/2022/#technology

have a framework that can be used for web development in conjunction with web-based languages, such as Spring[12] for Java and Django[13] for Python.

These frameworks could provide a potential route to teaching web development. Most introductory courses teach an OOP-based language, allowing students to learn server-side-based web development without first needing to learn client-side web-based languages. Furthermore, these OOP-based frameworks often support unit tests, potentially allowing for existing AATs to be used to assess web-development assignments. However, further research should be conducted into the automated assessment of web-based languages, as these assessments are rarely automated.

7.5 Evaluation Techniques

7.5.1 Approaches. Conducting student and instructor surveys has many benefits, including providing insight into the users' experience with the tool, how well it worked for them, and how it could be improved. However, surveys cannot validate the accuracy of AATs, as the surveys only collect the user's opinion of how well the tool performed. To validate the accuracy of AATs, the results of the proposed tool should be compared to a benchmark, either other tools or, ideally, a human-graded dataset.

While comparing to a benchmark dataset can validate the tool's accuracy, human graders are typically better at understanding the nuances in a student's submission. This allows them to provide more accurate assessments, especially for partial or uncompilable submissions. However, using multiple human graders to assess large courses is common practice, which can introduce some variability in awarded grades and feedback given. Evaluating against other tools can demonstrate improved accuracy when running against the same benchmark.

Conducting a mixture of quantitative evaluation in the form of comparing against benchmarks, both against other tools and human graders, and qualitative evaluation, such as student and instructor surveys, could provide the most in-depth analysis of AATs. This allows researchers to show how their tool improves upon other tools and compares to the gold standard of human graders while also providing valuable insights into the user's experience, both from a student and instructor's point of view.

7.5.2 Data Availability. Publicly distributing datasets would allow instructors and researchers to compare similar and new AATs against a shared dataset, allowing them to make informed decisions on which tools to use or adapt for their purposes instead of developing another automated grader from scratch. Furthermore, releasing datasets alongside the study allows researchers to reproduce and validate experiment results, aiding instructors in choosing which AAT to use. Among the released datasets, few are utilised in multiple papers. Those utilised are typically from large-scale courses using smaller-scale programming exercises from online judges, such as Hacker Rank [99, 184]. The lack of validated available benchmarks can make it difficult for researchers to validate the results of their tools, especially for AATs that focus on less commonly assessed skills, including maintainability, readability, and documentation.

In addition to increasing reproducibility and providing data for benchmarking, releasing publicly available datasets could also allow researchers to find relevant data for their research, whether automated assessment-related or other code-based research, without requiring the researchers to develop a new dataset. This could decrease the overall time spent on a project and produce cutting-edge research faster.

[12]Spring – A Framework for Java Microservices and Web-Apps: https://spring.io/
[13]Django – A web framework for Python: https://www.djangoproject.com/

7.6 Limitations of AAT Provided Feedback (RQ6)

Most AATs provided feedback in one form or another, with dynamic analysis tools typically showing whether the test cases succeeded and if they failed the expected output and actual output or the exception/compiler message if one was thrown. Providing the test results can aid students' learning, especially when providing near-instantaneous feedback allowing students to get feedback on their progress. However, the lack of detailed feedback can frustrate students when they cannot figure out why certain test cases are failing, for example, when the expected and actual outputs look identical to the student but have an unnoticed trailing space. Furthermore, passing compiler and runtime exceptions directly to the students without post-processing is inadequate and presents a barrier to progress and a source of discouragement [7].

Feedback utilising static analysis also shares similar limitations. The tools that use code repair typically suggest edits that can be made to make the code compilable but with a limited explanation of why the suggested fix makes the code compile. Tools based on software metrics or linters often provide overwhelming feedback to the student, typically highlighting each occurrence of a readability, maintainability, or documentation issue. Furthermore, as these AATs are typically based on tools designed for professional software engineers, the feedback supplied to the student can be confusing and contain feedback on topics they have yet to learn about.

While there are limitations to feedback provided by AATs, human-provided feedback is also imperfect. Instructors can provide more nuanced and directed feedback for particular students. However, this takes time, and when assessing large cohorts is practically impossible. AATs typically provide enough feedback to aid most students' learning while allowing instructors more time to aid struggling students. Further research into the effects of AATs on student learning compared to assessment by human graders could be undertaken in the future.

7.7 Performance Against Human Assessors (RQ4, RQ5)

Evaluating AATs against human assessors is a common method of assessing the quality of AATs, second to conducting user studies. Most evaluate how accurate or well the AAT grades correlate with human graders, with few comparing the automatically generated and human-generated feedback. Those that evaluate automatically generated feedback against human-generated feedback primarily focus on whether the instructors agree with the generated feedback provided or if the generated feedback is accurate regarding errors detected or categorisation of issues within the code.

Further research is required to evaluate the learning effect of automatically generated grades and feedback compared to human-provided grades and feedback, mainly which elements are assessed and the quality and quantity of the feedback provided to the students.

7.8 Results Compared to Related Systematic Literature Reviews

Most of the related work investigates AATs outside of our publication window of 2017 to 2021; here, we compare our results with the previous reviews in this area to discuss any potential long-term trends when combined with our results. Souza et al. [64] investigated AATs between 1985 and 2013 and found that most tools were fully automated, with less than a quarter of tools that they reviewed opting to develop a semi-automated tool. This trend for fully automated tools has not changed in our review, and we also found that most tools were fully automated, with only 14% of tools using a semi-automated approach.

Similarly, Keuning et al. [33] focused on conducting a systematic literature review of feedback provided by AATs between 1960 and 2015. They provide a more in-depth analysis of feedback provided by AATs by utilising Narciss's [47] feedback categories and found that most AATs provide

feedback on finding mistakes using test-based feedback. Our results also show this trend of providing feedback on mistakes, primarily using the output of dynamic analysis. Keuning et al. also found that some AATs provide feedback on how to proceed. Our review also found instances of AATs providing feedback on how to proceed, typically tools that provide feedback based on code repair.

Furthermore, Keuning et al. categorised the tools by language paradigm assessed and had similar results to our review, with most tools assessing object-oriented languages and few tools supporting the assessment of logic, functional, or other paradigms. They also analysed the quality of the AATs they reviewed. They found that most utilised empirical approaches, such as comparing to learning objectives, conducting student and teacher surveys, or evaluating the AAT based on the time taken to complete a task, with other tools being evaluated analytically, anecdotally or not at all.

We found similar trends in assessing object-oriented languages and most tools being evaluated using empirical methods. However, we provided a more in-depth analysis of evaluation techniques and found that most tools are evaluated by surveys or by comparing the results against manual grading.

While the other systematic reviews discussed investigate papers published in years that do not overlap our review, Paiva et al. [53]'s review investigates publications between 2010 and 2021. They investigated assessment techniques and found that whitebox dynamic analysis and static analysis are gaining more traction as methods to assess the functionality of submissions.

For code quality and software metrics, they discuss tools that utilise existing code quality tools and software metrics. While there is an overlap to Paiva et al.'s review, our review provides a more detailed analysis of the techniques used to assess submissions by introducing categorising tools by both the key skills and the approaches used – whereas Paiva et al. primarily focus on the domains graded, such as visual programming, computer graphics, and software testing, how secure the code execution is, and the effectiveness of the tools.

7.9 Threats to Validity

Limiting the search for primary studies to IEEExplore, ACM DL, and Scopus may have led to some relevant primary studies not published in an IEEE, ACM or a Scopus-indexed publication being missed. Additionally, given that some titles and abstracts did not clearly state what their papers addressed, some papers may have been mistakenly excluded during the title and abstract screening. To mitigate incorrectly excluding papers at this stage, the screeners opted to include such papers in the title and abstract screening to allow further analysis during the introduction and conclusion screening stage. If, during the screening stage, the screeners disagreed on whether a paper should be included, they discussed their disagreements and came to a consensus if the paper should be included.

We opted not to conduct a snowball search, where after the screening stage, included papers references are searched for any other papers that match the inclusion criteria. We decided against conducting a snowball search, as we felt that our final number of included papers was enough to provide a practical overview of the state of the art without delaying the publication of our review. However, as a result, there are other papers we most likely have missed in our initial search that do not contain the keywords in our search string in their titles or abstracts.

Commercial AATs may not have been included in our study as they do not have any associated academic literature. While many institutions use these commercial tools, the lack of literature discussing them makes them out of scope for our literature review.

Although both screeners conducted the first round of annotations during the full-text screening stage, additional annotations were needed while extracting the results. These additional annotations were compiled by a single screener, with the final list of applied annotations being validated

by the second screener. Therefore, some papers may have been mislabeled or missing relevant annotations.

While writing our results for this paper, there was a rise in research into using **large language models** (**LLMs**) within computer science education. This rise in LLMs can affect automated assessment tools, especially with smaller, constrained assignments that are more susceptible to the code generation functionality of LLMs. One study found that if ChatGPT is given clear and straightforward instructions, it can generate effective solutions for trivial and constrained assignments [49]. Another study found that the solutions obtained for non-trivial programming assignments are not sufficient for the competition of the course, the LLM can correct solutions based on feedback from an AAT [60].

8 CONCLUSION

This systematic review categorised state-of-the-art automated grading and feedback tools by the graded programming skills, techniques for awarding a grade or generating feedback, programming paradigms for automatically assessing programming assignments, and how these tools were evaluated. Furthermore, we investigated why automated assessment tools focus on assessing specific programming fundamental skills, discussed potential reasons for choosing a particular language and degree of automation, and investigated how researchers evaluated tools and how evaluation could be improved.

We found that most AATs in the scope of this review focus on assessing correctness using a dynamic approach, most of which used unit testing to provide the grade and feedback. Feedback is typically limited to whether the unit test has passed or failed and the expected and actual outcomes if the test has failed. This can leave students frustrated, as often the feedback does not provide enough detail to help the student to progress.

Another common approach to assess correctness is a static approach comparing a student's submission to a reference solution or a set of correct student submissions. Static analysis is also used to assess both maintainability, readability and the presence of documentation. However, these skills are assessed less often and typically in conjunction with correctness grading.

Instructors focus on assessing correctness, which is typically seen as one of the most crucial skills. They can determine if students have used a specific language feature or algorithm and if students can understand and convert requirements into a complete code base. While correctness is often seen as one of the most crucial skills, maintainability, readability, and documentation are also vital. Maintainable, readable, and well-documented code is typically easier to develop further, especially when working in teams or on future releases.

Most tools offered fully automated assessment, allowing for near-instantaneous feedback and multiple resubmissions without increasing the grading workload. Receiving feedback quickly increases student satisfaction, and multiple submissions allow students more opportunities to succeed. However, fully automated tools typically limit the scope of the assignment to a smaller scale and limit opportunities to show creativity.

Some tools opt for semi-automated approaches, where human graders manually assess elements of the assignment, typically maintainability, readability and documentation, and automatically assess correctness. While semi-automated approaches do not allow for near-instantaneous feedback, they are faster than manual assessment. However, typical implementations do not resolve the issues around limiting the scope of assignments, as correctness is typically assessed using the same methodology as full-automated assessment.

In terms of language paradigms assessed, most assess object-oriented languages, such as Java, Python, and C++. Other language paradigms, such as functional and logic languages, have AATs, but these are researched less often. Object-oriented languages are the primary focus for many

automated assessments due to the prominence of these languages in education and industry. Other language paradigms are gaining more popularity, especially web-based languages like JavaScript.

Most papers evaluate an AAT, whether one they have developed or used in a course. The primary evaluation technique was to conduct student surveys about their thoughts on the tool and how using the tool aided their learning. Another common evaluation technique was to compare the AAT to human graders, most focusing on the accuracy of the assessment using the human graders' marks as a benchmark. The dataset used for evaluation is typically not published. While most papers perform some form of evaluation, the evaluation is typically focused on a single tool, is conducted by the tool's developer and has mostly positive results.

While evaluating tools with student and teacher surveys can provide valuable insights into the users' experience with the tools, they do not evaluate the accuracy of the awarded grades and feedback. Evaluating the accuracy against a benchmark or to human graders in conjunction with the user's experience allows instructors to compare similar tools for considerations in their courses. Releasing evaluation datasets would allow researchers to reproduce and validate experiment results and evaluate different tools against a common set of assignments. This would improve the evaluation by providing verifiable and comparable benchmarks for the tool's accuracy.

8.1 Recommendations

With most research into automated assessment of programming assignments focusing on assessing correctness for small-scale closed assignments, we encourage researchers to investigate how to automatically assess maintainability, readability, and documentation, as these are key skills that are not evaluated by most automated assessment tools. Furthermore, we suggest that future research investigates how to assess open-ended assignments. Including semi-automated approaches to automating the assessment of maintainability, readability, and documentation while manually assessing correctness or designing open-ended assignments to automate the assessment fully.

As web-based languages become more prominent, future research could investigate the automatic assessment of web-based languages, including JavaScript and TypeScript. In addition to web-based languages, automatic assessment of web application development could be investigated further by investigating how to assess the use of popular web frameworks, user-experience design, and web-testing frameworks.

While researching new methods of automatic assessment for open-ended assignments, maintainability, readability and documentation and web-based languages, we suggest that authors produce more robust evaluation practices and attempt to publish the datasets they used for their evaluation. While some tools are evaluated by comparing their results to human graders, most are only evaluated by student or teacher surveys or performance analytics, such as compute power required or run times. Evaluating tools with a survey can provide meaningful insights into the end-users opinions of a tool, but they cannot adequately determine the accuracy of the tool. The publication of annotated datasets would allow researchers to evaluate a tool's accuracy to similar tools on the same dataset, providing more significant evidence of the tool's performance. Evaluating against a benchmark dataset and using user surveys could provide a good mix of qualitative and quantitative evidence to support the performance of their tools.

8.2 Data Availability

Our final list of annotated papers and data processing pipeline is available on GitHub.[14]

[14]Raw data and data processing repository: https://github.com/m-messer/Automated-Assessment-SLR-Data-Processing

A SUPPLEMENTARY MATERIAL

A.1 Publications by Skill Graded and Category of Automatic Assessment Tools

		Category		
		Dynamic	Static	Machine Learning
Skill	Correctness	[74, 76–91, 93–99, 101, 103, 106–115, 118–125, 128–130, 133, 134, 136, 137, 139–145, 149–152, 154, 156, 158–168, 173, 175, 185, 187, 188, 191, 193]	[74, 75, 77–79, 84–86, 89, 91, 92, 94, 95, 105, 116, 117, 119, 123–126, 128, 132, 137, 138, 145, 147–149, 153, 160, 165, 169–172, 176, 178–180, 182, 184–186, 189, 190, 192]	[74, 92, 104, 110, 155, 174, 177, 184]
	Maintainability	[120, 133]	[76, 100, 102, 115, 135, 139, 169, 170, 181–183]	
	Readability		[87, 93, 102, 109, 111, 112, 118, 120, 122, 126, 127, 130–132, 134–136, 146, 156, 157, 168, 169, 174, 181, 183, 187, 191, 193]	[104]
	Documentation		[116]	

A.2 Publications by Technique Implemented

		Technique	References
Category	Static	Static Analysis	[76, 79, 87, 93, 95, 100, 102, 109, 111, 115, 116, 118, 122, 127, 131, 132, 137, 143, 146, 149, 156, 168, 169, 173, 174, 182, 183, 185, 187, 193]
		Pattern Matching	[79, 84, 92, 94, 124, 125, 138, 147, 148, 153, 170–172, 174, 176, 179–181, 184, 185, 189, 190, 192]
		Model Solution Required	[77, 81, 89, 100, 113, 115, 125, 126, 139, 141, 148, 152, 170, 171, 174, 176, 179, 180, 184, 185, 189, 190]
		Style Check	[76, 89, 91, 93, 96, 112, 116, 120, 122, 123, 130, 131, 136, 149, 157, 187, 191]
		Program Repair	[74, 75, 79, 85, 117, 156, 160, 176, 178, 179, 183, 186]
		Model Solution Closeness	[79, 117, 148, 153, 170, 171, 174, 180, 184–186]
		Code Repair for Feedback	[74, 117, 125, 141, 172, 176, 178, 186]
		Source Code Metrics	[76, 102, 126, 132, 169, 170, 183]
	Dynamic	Unit Testing	[76–91, 93–99, 101, 103, 106–114, 118–125, 128–130, 133, 134, 136, 137, 139–145, 149–152, 154, 156, 158–168, 173, 175, 185, 187, 188, 191, 193]
		Output Matching	[79, 94, 98, 147, 154, 155, 192]
		Property-Based Testing	[83, 115, 129]
	Other	Machine Learning	[74, 92, 102, 104, 110, 155, 156, 174, 177, 184]
		Other	[77–79, 81, 81, 82, 85, 85, 86, 88, 89, 97, 101, 102, 105, 109, 110, 113, 115, 116, 116–119, 119, 120, 120, 120, 121, 123, 125, 126, 126, 128, 133, 134, 134, 135, 135–137, 139, 139, 140, 145, 148–150, 153, 153, 153, 155, 157, 157, 159, 162, 162, 165, 170–172, 174, 179, 179–181, 181, 182, 185, 186, 186, 187, 189, 190, 192]

A.3 Publications by Degree of Automation

Degree of Automation	References
Fully Automated	[74–76, 79–82, 84, 86–93, 95, 98–107, 110–118, 120–126, 128, 129, 131–139, 141, 143–150, 152–161, 163–168, 170–173, 175, 176, 179–181, 183–193]
Semi-Automated	[77, 78, 83, 94, 96, 108, 109, 119, 125, 130, 140, 142, 151, 162, 169, 177, 182]
Unclear	[85, 97, 127, 174, 178]

A.4 Publications by Language Paradigm

Language Paradigm	References
OOP	[74, 75, 77, 79, 81–86, 88, 90, 91, 93, 95–98, 100–102, 104–112, 114, 116–118, 121, 123–137, 139, 140, 143–146, 148, 150, 152, 153, 155, 156, 158–160, 162–170, 172–175, 177, 179, 181–183, 186–189, 191, 192]
Functional	[76, 83, 89, 113, 115, 120, 141, 147, 159, 176]
Low-Level	[103, 149, 161]
Web	[157, 170, 193]
Query	[185]
Graphics	[77, 78, 150, 190]
Logic	[138]
Agnostic	[80, 92, 98, 142, 154, 173, 180]
Unknown	[87, 94, 99, 119, 122, 151, 171, 178, 184]

A.5 Publications by Evaluation Technique

Evaluation Technique	References
Student Survey	[80, 82, 83, 88, 90, 91, 100, 102–105, 112–115, 117, 121, 122, 124, 126, 129, 134, 135, 142, 145, 149, 150, 152, 162–164, 166, 168, 175, 176, 190, 191, 193]
Manual Grading	[74, 75, 79, 94, 98, 101, 108, 111, 113, 119, 123, 125, 126, 131, 138, 142–144, 149, 150, 155, 160, 165, 167, 169, 171, 172, 180, 182, 184–187, 189, 192]
Analytics	[77, 87, 95, 112, 116, 127, 131–134, 140, 146, 153, 157–159, 166, 168, 173, 186]
Auto Grading	[83, 84, 88, 92, 94, 121, 124–126, 139, 148, 158, 161, 166, 174]
None	[78, 110, 118, 120, 128, 130, 151, 156, 170, 181, 183]
User Study	[124, 141, 155, 167, 175, 179]
Grading Accuracy	[79, 81, 97, 177, 185]
Teacher Survey	[91, 109, 114, 135, 152, 178]
Compared to Other Tools	[85, 154, 176, 179]
Anecdotal	[76, 86, 96, 107, 188]
Other	[82, 89, 93, 96, 99, 106, 127, 129, 136, 137, 146, 163, 174, 186]

A.6 Publications by Dataset Availability

Dataset Availability	References
Unavailable	[76–81, 83–94, 96–98, 101–111, 113–138, 140, 142, 144–147, 149–152, 154–168, 170–173, 175, 177–183, 187–193]
Available	[74, 75, 99, 100, 112, 141, 174, 176, 184, 185]
On Request	[82, 95, 125, 139, 143, 148, 153, 169, 186]

ACKNOWLEDGMENTS

We appreciate the reviewers' insightful remarks and recommendations.

REFERENCES

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

[2] Hussam Aldriye, Asma Alkhalaf, and Muath Alkhalaf. 2019. Automated grading systems for programming assignments: A literature review. *International Journal of Advanced Computer Science and Applications* 10, 3 (2019), 215–222. DOI: https://doi.org/10.14569/IJACSA.2019.0100328

[3] Basma S. Alqadi and Jonathan I. Maletic. 2017. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* . ACM, New York, NY, 15–20. DOI: https://doi.org/10.1145/3017680.3017761

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE Software* 25, 5 (2008), 22–29. DOI: https://doi.org/10.1109/MS.2008.130

[5] Maha Aziz, Heng Chi, Anant Tibrewal, Max Grossman, and Vivek Sarkar. 2015. Auto-grading for parallel programs. In *Proceedings of the Workshop on Education for High-Performance Computing* . ACM, New York, NY, Article 3, 8 pages. DOI: https://doi.org/10.1145/2831425.2831427

[6] Thoms Ball. 1999. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* . Springer-Verlag, Berlin, 216–234.

[7] Brett A. Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* . ACM, New York, NY, 126–131. DOI: https://doi.org/10.1145/2839509.2844584

[8] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. "I Know It When I See It" perceptions of code quality: ITiCSE'17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* . ACM, New York, NY, 70–85. DOI: https://doi.org/10.1145/3174781.3174785

[9] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. DOI: https://doi.org/10.1109/32.295895

[10] Chih-Yueh Chou and Yan-Jhih Chen. 2021. Virtual teaching assistant for grading programming assignments: Non-dichotomous pattern based program output matching and partial grading approach. In *Proceedings of the 2021 IEEE 4th International Conference on Knowledge Innovation and Invention* . 170–175. DOI: https://doi.org/10.1109/ICKII51822.2021.9574713

[11] James Williams, David Kane, and Gillian Cappuccini-Ansfield. 2008. Student satisfaction surveys: The value in taking an historical perspective. *Quality in Higher Education* 14, 2 (2008), 135–155. DOI: https://doi.org/10.1080/13538320802278347 arXiv:https://doi.org/10.1080/13538320802278347

[12] Draylson Micael de Souza, Michael Kölling, and Ellen Francine Barbosa. 2017. Most common fixes students use to improve the correctness of their programs. In *Proceedings of the 2017 IEEE Frontiers in Education Conference*. 1–9. DOI: https://doi.org/10.1109/FIE.2017.8190524

[13] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information* . ACM, New York, NY, 68–75. DOI: https://doi.org/10.1145/1085313.1085331

[14] Juan Carlos Rodríguez del Pino, Enrique Rubio Royo, and Zenón Hernández Figueroa. 2012. A virtual programming lab for Moodle with automatic assessment and anti-plagiarism features. In *Proceedings of The 2012 International Conference on e-Learning, e-Business, Entreprise Information Systems, & e-Government*. Retrieved from http://hdl.handle.net/10553/9773

[15] Paul Denny, James Prather, and Brett A. Becker. 2020. Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 480–486. DOI: https://doi.org/10.1145/3341525.3387384

[16] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting early engagement with programming assignments using scheduled automated feedback. In *Proceedings of the 23rd Australasian Computing Education Conference* . ACM, New York, NY, 88–95. DOI: https://doi.org/10.1145/3441636.3442309

[17] Dante D. Dixson and Frank C. Worrell. 2016. Formative and summative assessment in the classroom. *Theory Into Practice* 55, 2 (2016), 153–159. DOI: https://doi.org/10.1080/00405841.2016.1148989 arXiv:https://doi.org/10.1080/00405841.2016.1148989

[18] Yu Dong, Jingyang Hou, and Xuesong Lu. 2020. An intelligent online judge system for programming training. In *Database Systems for Advanced Applications*. Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.), Springer International Publishing, Cham, 785–789. DOI: https://doi.org/10.1007/978-3-030-59419-0_57

[19] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* 5, 3 (2005), 4–es. DOI: https://doi.org/10.1145/1163405.1163409

[20] Aleksandr Efremov, Ahana Ghosh, and Adish Singla. 2020. Zero-shot learning of hint policy via reinforcement learning and program synthesis. In *Proceedings of the 13th International Conference on Educational Data Mining*. 338–394.

[21] Derar Eleyan, Abed Othman, and Amna Eleyan. 2020. Enhancing software comments readability using Flesch reading ease score. *Information* 11, 9 (2020), 1–25. DOI: https://doi.org/10.3390/info11090430

[22] Alex Gerdes, Johan T. Jeuring, and Bastiaan J. Heeren. 2010. Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 441–445. DOI: https://doi.org/10.1145/1734263.1734412

[23] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* . ACM, New York, NY, 160–168. DOI: https://doi.org/10.1145/3230977.3231002

[24] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

[25] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* . ACM, New York, NY, 36–47. DOI: https://doi.org/10.1145/3524610.3527909

[26] Rowan Hart, Brian Hays, Connor McMillin, El Kindi Rezig, Gustavo Rodriguez-Rivera, and Jeffrey A. Turkstra. 2023. Eastwood-Tidy: C linting for automated code style assessment in programming courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* . ACM, New York, NY, 799–805. DOI: https://doi.org/10.1145/3545945.3569817

[27] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Communications of the ACM* 3, 10 (10 1960), 528–529. DOI: https://doi.org/10.1145/367415.367422

[28] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* . ACM, New York, NY, 86–93. DOI: https://doi.org/10.1145/1930464.1930480

[29] David Insa and Josep Silva. 2015. Semi-automatic assessment of unrestrained Java code: A library, a DSL, and a workbench to assess exams and exercises. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 39–44. DOI: https://doi.org/10.1145/2729094.2742615

[30] Yiannis Kanellopoulos, Panos Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, and Nikos Tsirakis. 2010. Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering and Applications* 1, 3 (2010), 17–36. DOI: https://doi.org/10.5121/ijsea.2010.1302

[31] Cem Kaner, Jack Falk, and Hung Q. Nguyen. 1999. *Testing Computer Software*. John Wiley & Sons.

[32] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 110–115. DOI: https://doi.org/10.1145/3059009.3059061

[33] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education* 19, 1, Article 3 (09 2018), 43 pages. DOI: https://doi.org/10.1145/3231711

[34] Noela J. Kipyegen and William P. K. Korir. 2013. Importance of software documentation. *International Journal of Computer Science Issues* 10, 5 (09 2013), 223–228.

[35] B. Kitchenham and S. Charters. 2007. Guidelines for performing systematic literature reviews in software engineering. *Technical Report, Technical Report, Ver. 2.3, EBSE Technical Report*. Vol. 5, EBSE; 2007.

[36] Malcolm S. Knowles. 1975. Self-directed learning: A guide for learners and teachers. Association Press.

[37] Y. Ben-David Kolikant and M. Mussai. 2008. "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education* 18, 2 (2008), 135–151. DOI: https://doi.org/10.1080/08993400802156400 arXiv:https://doi.org/10.1080/08993400802156400

[38] Michael Kölling. 1999. The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-Oriented Programming* 11, 8 (1999), 8–15.

[39] Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. 2020. An interactive learning method to engage students in modeling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* . ACM, New York, NY, 12–22. DOI: https://doi.org/10.1145/3377814.3381701

[40] Adidah Lajis, Shahidatul Arfah Baharudin, Diyana Ab Kadir, Nadilah Mohd Ralim, Haidawati Mohd Nasir, and Normaziah Abdul Aziz. 2018. A review of techniques in automatic programming assessment for practical skill test. *Journal of Telecommunication, Electronic and Computer Engineering* 10, 2–5 (07 2018), 109–113. Retrieved from https://jtec.utem.edu.my/jtec/article/view/4394

[41] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A comparison of immediate and scheduled feedback in introductory programming projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* . ACM, New York, NY, 885–891. DOI: https://doi.org/10.1145/3478431.3499372

[42] Paul W. McBurney. 2015. Automatic documentation generation via source code summarization. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2, 903–906. DOI: https://doi.org/10.1109/ICSE.2015.288

[43] Paul W. McBurney, Siyuan Jiang, Marouane Kessentini, Nicholas A. Kraft, Ameer Armaly, Mohamed Wiem Mkaouer, and Collin McMillan. 2018. Towards prioritizing documentation effort. *IEEE Transactions on Software Engineering* 44, 9 (2018), 897–913. DOI: https://doi.org/10.1109/TSE.2017.2716950

[44] T. J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. DOI: https://doi.org/10.1109/TSE.1976.233837

[45] Marcus Messer. 2022. Automated Grading and Feedback Tools: A Systematic Review. *Open Science Foundation Preregistrations*. DOI: https://doi.org/10.17605/OSF.IO/VXTF9

[46] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2023. Machine learning-based automated grading and feedback tools for programming: A meta-analysis. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* . ACM, New York, NY, 491–497. DOI: https://doi.org/10.1145/3587102.3588822

[47] Susanne Narciss. 2008. *Feedback Strategies for Interactive Learning Tasks*. Taylor and Francis. 125–143 pages. DOI: https://doi.org/10.4324/9780203880869-13

[48] Sidhidatri Nayak, Reshu Agarwal, and Sunil Kumar Khatri. 2022. Automated assessment tools for grading of programming assignments: A review. In *Proceedings of the 2022 International Conference on Computer Communication and Informatics*. 1–4. DOI: https://doi.org/10.1109/ICCCI54379.2022.9740769

[49] Eng Lieh Ouh, Benjamin Kok Siew Gan, Kyong Jin Shim, and Swavek Wlodkowski. 2023. ChatGPT, can you generate solutions for my coding exercises? An evaluation on its effectiveness in an undergraduate Java programming course. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* . ACM, New York, NY, 54–60. DOI: https://doi.org/10.1145/3587102.3588794

[50] Mourad Ouzzani, Hossam Hammady, Zbys Fedorowicz, and Ahmed Elmagarmid. 2016. Rayyan–a web and mobile app for systematic reviews. *Systematic Reviews* 5, 1 (12 2016), 1–10. DOI: https://doi.org/10.1186/S13643-016-0384-4

[51] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2017. The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Data Mining* 10, 1 (8 2017), 1–35. DOI: https://doi.org/10.5281/zenodo.3554697

[52] Matthew J. Page, Joanne E. McKenzie, Patrick M. Bossuyt, Isabelle Boutron, Tammy C. Hoffmann, Cynthia D. Mulrow, Larissa Shamseer, Jennifer M. Tetzlaff, Elie A. Akl, Sue E. Brennan, Roger Chou, Julie Glanville, Jeremy M. Grimshaw, Asbjørn Hróbjartsson, Manoj M. Lalu, Tianjing Li, Elizabeth W. Loder, Evan Mayo-Wilson, Steve McDonald, Luke A. McGuinness, Lesley A. Stewart, James Thomas, Andrea C. Tricco, Vivian A. Welch, Penny Whiting, and David Moher. 2021. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *Systematic Reviews* 10, 1 (2021), 89. DOI: https://doi.org/10.1186/s13643-021-01626-4

[53] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education* 22, 3, Article 34 (06 2022), 40 pages. DOI: https://doi.org/10.1145/3513140

[54] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 92–97. DOI: https://doi.org/10.1145/3059009.3059026

[55] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*. Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, 1093–1102. Retrieved from https://proceedings.mlr.press/v37/piech15.html

[56] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education* 18, 1, Article 1 (10 2017), 24 pages. DOI: https://doi.org/10.1145/3077618

[57] Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura. 2020. Source code assessment and classification based on estimated error probability using attentive LSTM language model and its application in programming education. *Applied Sciences* 10, 8 (2020), 1–21. DOI: https://doi.org/10.3390/app10082973

[58] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology* 13, 5, Article 84 (06 2022), 44 pages. DOI: https://doi.org/10.1145/3519312

[59] Jef Raskin. 2005. Comments are more important than code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation. *Queue* 3, 2 (03 2005), 64–65. DOI: https://doi.org/10.1145/1053331.1053354

[60] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can generative pre-trained transformers (GPT) pass assessments in higher education programming courses?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* . ACM, New York, NY, 117–123. DOI: https://doi.org/10.1145/3587102.3588792

[61] Kevin Sendjaja, Satrio Adi Rukmono, and Riza Satria Perdana. 2021. Evaluating control-flow graph similarity for grading programming exercises. In *Proceedings of the 2021 International Conference on Data and Software Engineering*. 1–6. DOI: https://doi.org/10.1109/ICoDSE53690.2021.9648464

[62] Sadia Sharmin. 2021. Creativity in CS1: A literature review. *ACM Transactions on Computing Education* 22, 2, Article 16 (11 2021), 26 pages. DOI: https://doi.org/10.1145/3459995

[63] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* . ACM, New York, NY, 15–26. DOI: https://doi.org/10.1145/2491956.2462195

[64] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. 2016. A systematic literature review of assessment tools for programming assignments. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training* . 147–156. DOI: https://doi.org/10.1109/CSEET.2016.48

[65] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. 2002. Code quality analysis in open source software development. *Information Systems Journal* 12, 1 (2002), 43–60. DOI: https://doi.org/10.1046/j.1365-2575.2002.00117.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1046/j.1365-2575.2002.00117.x

[66] Ioanna Stamouli and Meriel Huggard. 2006. Object oriented programming and program correctness: The students' perspective. In *Proceedings of the 2nd International Workshop on Computing Education Research* . ACM, New York, NY, 109–118. DOI: https://doi.org/10.1145/1151588.1151605

[67] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* . ACM, New York, NY, 160–164. DOI: https://doi.org/10.1145/2999541.2999555

[68] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *Proceedings of the 2013 21st International Conference on Program Comprehension* . 83–92. DOI: https://doi.org/10.1109/ICPC.2013.6613836

[69] Zahid Ullah, Adidah Lajis, Mona Jamjoom, Abdulrahman Altalhi, Abdullah Al-Ghamdi, and Farrukh Saleem. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education* 26, 6 (2018), 2328–2341. DOI: https://doi.org/10.1002/cae.21974 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21974

[70] Leo C. Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 738–744. DOI: https://doi.org/10.1145/3287324.3287463

[71] Orr Walker and Nathaniel Russell. 2021. Automatic assessment of the design quality of Python programs with personalized feedback. In *Proceedings of the 14th International Conference on Educational Data Mining*. 495–501.

[72] Jinshui Wang, Yunpeng Zhao, Zhengyi Tang, and Zhenchang Xing. 2020. Combining dynamic and static analysis for automated grading SQL statements. *Taiwan Ubiquitous Information* 5, 4 (2020), 179–190.

[73] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Advances in Web-Based Learning - ICWL 2012*. Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht (Eds.), Springer Berlin Heidelberg, Berlin, 228–239.

PUBLICATIONS INCLUDED AS PART OF THE SEARCH PROCESS

[74] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted example generation for compilation errors. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering*. 327–338. DOI: https://doi.org/10.1109/ASE.2019.00039

[75] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* . ACM, New York, NY, 139–150. DOI: https://doi.org/10.1145/3377814.3381703

[76] José Bacelar Almeida, Alcino Cunha, Nuno Macedo, Hugo Pacheco, and José Proença. 2018. Teaching how to program using automated assessment and functional glossy games (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 82 (2018), 17 pages. DOI: https://doi.org/10.1145/3236777

[77] Carlos Andujar, Cristina Raluca Vijulie, and Alvar Vinacua. 2019. A parser-based tool to assist instructors in grading computer graphics assignments. In *Eurographics 2019 - Education Papers*. Marco Tarini and Eric Galin (Eds.), The Eurographics Association. DOI: https://doi.org/10.2312/eged.20191025

[78] Carlos Andujar, Cristina R. Vijulie, and Àlvar Vinacua. 2020. Syntactic and semantic analysis for extended feedback on computer graphics assignments. *IEEE Computer Graphics and Applications* 40, 3 (2020), 105–111. DOI: https://doi.org/10.1109/MCG.2020.2981786

[79] Sara Mernissi Arifi, Rachid Ben Abbou, and Azeddine Zahi. 2020. Assisted learning of C programming through automated program repair and feed-back generation. *Indonesian Journal of Electrical Engineering and Computer Science* 20, 1 (2020), 454–464. DOI: https://doi.org/0.11591/ijeecs.v20.i1.pp454-464

[80] Marini Abu Bakar, Norleyza Jailani, Rodziah Latih, Muriati Mukhtar, Isrul Esa, and Abdullah Mohd Zin. 2018. Automarking system: A support tool for learning of programming. *International Journal on Advanced Science Engineering and Information Technology* 8, 4 (2018), 1313–1320. DOI: https://doi.org/10.18517/ijaseit.8.4.6416

[81] M. Rifky I. Bariansyah, Satrio Adi Rukmono, and Riza Satria Perdana. 2021. Semantic approach for increasing test case coverage in automated grading of programming exercise. In *Proceedings of the 2021 International Conference on Data and Software Engineering*. 1–6. DOI: https://doi.org/10.1109/ICoDSE53690.2021.9648439

[82] Chelsea Barraball, Moeketsi Raselimo, and Bernd Fischer. 2020. An interactive feedback system for grammar development (tool paper). In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* . ACM, New York, NY, 101–107. DOI: https://doi.org/10.1145/3426425.3426935

[83] Alessandro Bertagnon and Marco Gavanelli. 2020. MAESTRO: A semi-autoMAted evaluation SysTem for pROgramming assignments. In *Proceedings of the 2020 International Conference on Computational Science and Computational Intelligence*. 953–958. DOI: https://doi.org/10.1109/CSCI51800.2020.00177

[84] Anis Bey, Patrick Jermann, and Pierre Dillenbourg. 2017. An empirical study comparing two automatic graders for programming. MOOCs context. In *Data Driven Approaches in Digital Education*. Élise Lavoué, Hendrik Drachsler, Katrien Verbert, Julien Broisin, and Mar Pérez-Sanagustín (Eds.), Springer International Publishing, Cham, 537–540. DOI: https://doi.org/10.1007/978-3-319-66610-5_57

[85] Geoff Birch, Bernd Fischer, and Michael Poppleton. 2019. Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs. *Software & Systems Modeling* 18, 1 (2019), 445–471. DOI: https://doi.org/10.1007/s10270-017-0612-y

[86] Yadira Boada and Alejandro Vignoni. 2021. Automated code evaluation of computer programming sessions with MATLAB Grader. In *Proceedings of the 2021 World Engineering Education Forum/Global Engineering Deans Council*. 500–505. DOI: https://doi.org/10.1109/WEEF/GEDC53299.2021.9657355

[87] Chris Brown and Chris Parnin. 2021. Nudging students toward better software engineering behaviors. In *Proceedings of the 2021 IEEE/ACM 3rd International Workshop on Bots in Software Engineering*. 11–15. DOI: https://doi.org/10.1109/BotSE52550.2021.00010

[88] Yun-Zhan Cai and Meng-Hsun Tsai. 2019. Improving programming education quality with automatic grading system. In *Innovative Technologies and Learning*. Lisbet Rønningsbakk, Ting-Ting Wu, Frode Eika Sandnes, and Yueh-Min Huang (Eds.), Springer International Publishing, Cham, 207–215. DOI: https://doi.org/10.1007/978-3-030-35343-8_22

[89] Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. 2017. Scaling up functional programming education: Under the hood of the OCaml MOOC. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 4 (08 2017), 25 pages. DOI: https://doi.org/10.1145/3110248

[90] Marílio Cardoso, António Vieira de Castro, and Alvaro Rocha. 2018. Integration of virtual programming lab in a process of teaching programming EduScrum based. In *Proceedings of the 2018 13th Iberian Conference on Information Systems and Technologies*. 1–6. DOI: https://doi.org/10.23919/CISTI.2018.8399261

[91] Marílio Cardoso, Rui Marques, António Vieira de Castro, and Álvaro Rocha. 2021. Using virtual programming lab to improve learning programming: The case of algorithms and programming. *Expert Systems* 38, 4 (2021), e12531. DOI: https://doi.org/10.1111/exsy.12531 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/exsy.12531

[92] Ted Carmichael, Mary Jean Blink, John C. Stamper, and Elizabeth Gieske. 2018. Linkage objects for generalized instruction in coding (LOGIC). In *Proceedings of the 31st International Florida Artificial Intelligence Research Society Conference* . Keith Brawner and Vasile Rus (Eds.), AAAI Press, 443–446. Retrieved from https://aaai.org/ocs/index.php/FLAIRS/FLAIRS18/paper/view/17702

[93] Hsi-Min Chen, Wei-Han Chen, and Chi-Chen Lee. 2018. An automated assessment system for analysis of coding convention violations in Java programming assignments⋆industry 4.0 view project an automated assessment system for analysis of coding convention violations in Java programming assignments ⋆. *Journal of Information Science and Engineering* 34, 5 (2018), 1203–1221. DOI: https://doi.org/10.6688/JISE.201809_34(5).0006

[94] Chih-Yueh Chou and Yan-Jhih Chen. 2021. Virtual teaching assistant for grading programming assignments: Non-dichotomous pattern based program output matching and partial grading approach. In *Proceedings of the 2021 IEEE 4th International Conference on Knowledge Innovation and Invention*. 170–175. DOI: https://doi.org/10.1109/ICKII51822.2021.9574713

[95] Sammi Chow, Kalina Yacef, Irena Koprinska, and James Curran. 2017. Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization* . ACM, New York, NY, 5–10. DOI: https://doi.org/10.1145/3099023.3099065

[96] Benjamin Clegg, Maria-Cruz Villa-Uriol, Phil McMinn, and Gordon Fraser. 2021. Gradeer: An open-source modular hybrid grader. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training*. 60–65. DOI: https://doi.org/10.1109/ICSE-SEET52601.2021.00015

[97] Benjamin S. Clegg, Phil McMinn, and Gordon Fraser. 2020. The influence of test suite properties on automated grading of programming exercises. In *Proceedings of the 2020 IEEE 32nd Conference on Software Engineering Education and Training*. 1–10. DOI: https://doi.org/10.1109/CSEET49119.2020.9206231

[98] Ricardo Conejo, Beatriz Barros, and Manuel F. Bertoa. 2019. Automated assessment of complex programming tasks using SIETTE. *IEEE Transactions on Learning Technologies* 12, 4 (2019), 470–484. DOI: https://doi.org/10.1109/TLT.2018.2876249

[99] Daniel Coore and Daniel Fokum. 2019. Facilitating course assessment with a competitive programming platform. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 449–455. DOI: https://doi.org/10.1145/3287324.3287511

[100] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. 2021. A comparison of inquiry-based conceptual feedback vs. traditional detailed feedback mechanisms in software testing education: An empirical investigation. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 87–93. DOI: https://doi.org/10.1145/3408877.3432417

[101] David Croft and Matthew England. 2020. Computing with CodeRunner at coventry university: Automated summative assessment of Python and C++ code. In *Proceedings of the 4th Conference on Computing Education Practice* . ACM, New York, NY, Article 1, 4 pages. DOI: https://doi.org/10.1145/3372356.3372357

[102] Gilbert Cruz, Jacob Jones, Meagan Morrow, Andres Gonzalez, and Bruce Gooch. 2017. An AI system for coaching novice programmers. In *Learning and Collaboration Technologies. Technology in Education*. Panayiotis Zaphiris and Andri Ioannou (Eds.), Springer International Publishing, Cham, 12–21. DOI: https://doi.org/10.1007/978-3-319-58515-4_2

[103] João Damas, Bruno Lima, and António J. Araújo. 2021. AOCO - a tool to improve the teaching of the ARM assembly language in higher education. In *Proceedings of the 2021 30th Annual Conference of the European Association for Education in Electrical and Information Engineering* . 1–6. DOI: https://doi.org/10.1109/EAEEIE50507.2021.9530951

[104] Melissa Day, Manohara Rao Penumala, and Javier Gonzalez-Sanchez. 2019. Annete: An intelligent tutoring companion embedded into the eclipse IDE. In *Proceedings of the 2019 IEEE 1st International Conference on Cognitive Machine Intelligence*. 71–80. DOI: https://doi.org/10.1109/CogMI48466.2019.00018

[105] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. 2020. Customizable and scalable automated assessment of C/C++ programming assignments. *Computer Applications in Engineering Education* 28, 6 (2020), 1449–1466. DOI: https://doi.org/10.1002/cae.22317 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22317

[106] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting early engagement with programming assignments using scheduled automated feedback. In *Proceedings of the 23rd Australasian Computing Education Conference* . ACM, New York, NY, 88–95. DOI: https://doi.org/10.1145/3441636.3442309

[107] Ignacio Despujol, Leonardo Salom, and Carlos Turró. 2020. Integrating the evaluation of out of the platform auto-evaluated programming exercises with personalized answer in Open edX. In *Proceedings of the 2020 IEEE Learning with MOOCS*. 14–18. DOI: https://doi.org/10.1109/LWMOOCS50143.2020.9234387

[108] Prasun Dewan, Andrew Wortas, Zhizhou Liu, Samuel George, Bowen Gu, and Hao Wang. 2021. Automating testing of visual observed concurrency. In *Proceedings of the 2021 IEEE/ACM 9th Workshop on Education for High Performance Computing*. 32–42. DOI: https://doi.org/10.1109/EduHPC54835.2021.00010

[109] Anton Dil and Joseph Osunde. 2018. Evaluation of a tool for Java structural specification checking. In *Proceedings of the 10th International Conference on Education Technology and Computers* . ACM, New York, NY, 99–104. DOI: https://doi.org/10.1145/3290511.3290528

[110] Yu Dong, Jingyang Hou, and Xuesong Lu. 2020. An intelligent online judge system for programming training. In *Database Systems for Advanced Applications*. Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.), Springer International Publishing, Cham, 785–789. DOI: https://doi.org/10.1007/978-3-030-59419-0_57

[111] Bob Edmison and Stephen H. Edwards. 2020. Turn up the heat! Using heat maps to visualize suspicious code to help students successfully complete programming problems faster. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* . ACM, New York, NY, 34–44. DOI: https://doi.org/10.1145/3377814.3381707

[112] Hans Fangohr, Neil O'Brien, Ondrej Hovorka, Thomas Kluyver, Nick Hale, Anil Prabhakar, and Arti Kashyap. 2020. Automatic feedback provision in teaching computational science. In *Computational Science–ICCS 2020*. Springer International Publishing, Cham, 608–621. https://link.springer.com/chapter/10.1007/978-3-030-50436-6_45#editor-information

[113] Molly Q. Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards answering "Am I on the Right Track?" Automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* . ACM, New York, NY, 13–24. DOI: https://doi.org/10.1145/3358711.3361626

[114] Nebojša Gavrilović, Aleksandra Arsić, Dragan Domazet, and Alok Mishra. 2018. Algorithm for adaptive learning process and improving learners' skills in Java programming language. *Computer Applications in Engineering Education* 26, 5 (2018), 1362–1382. DOI: https://doi.org/10.1002/cae.22043 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22043

[115] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: An adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 65–100. DOI: https://doi.org/10.1007/s40593-015-0080-x

[116] John Gerdes. 2017. Developing applications to automatically grade introductory visual basic courses. In *Proceedings of the Americas Conference on Information Systems*.

[117] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *SIGPLAN Not.* 53, 4 (06 2018), 465–480. DOI: https://doi.org/10.1145/3296979.3192387

[118] Thorsten Haendler, Gustaf Neumann, and Fiodor Smirnov. 2020. RefacTutor: An interactive tutoring system for software refactoring. In *Computer Supported Education*. H. Chad Lane, Susan Zvacek, and James Uhomoibhi (Eds.), Springer International Publishing, Cham, 236–261. DOI: https://doi.org/10.1007/978-3-030-58459-7_12

[119] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 278–283. DOI: https://doi.org/10.1145/3159450.3159502

[120] Aliya Hameer and Brigitte Pientka. 2019. Teaching the art of functional programming using automated grading (experience report). *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 115 (07 2019), 15 pages. DOI: https://doi.org/10.1145/3341719

[121] Qiang Hao, David H. Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H. Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (2022), 105–127. DOI: https://doi.org/10.1080/08993408.2020.1860408 arXiv:https://doi.org/10.1080/08993408.2020.1860408

[122] Qiang Hao and Michail Tsikerdekis. 2019. How automated feedback is delivered matters: Formative feedback and knowledge transfer. In *Proceedings of the 2019 IEEE Frontiers in Education Conference*. 1–6. DOI: https://doi.org/10.1109/FIE43999.2019.9028686

[123] Qiang Hao, Jack P. Wilson, Camille Ottaway, Naitra Iriumi, Kai Arakawa, and David H. Smith. 2019. Investigating the essential of meaningful automated formative feedback for programming assignments. In *Proceedings of the 2019 IEEE Symposium on Visual Languages and Human-Centric Computing*. 151–155. DOI: https://doi.org/10.1109/VLHCC.2019.8818922

[124] Emlyn Hegarty-Kelly and Dr Aidan Mooney. 2021. Analysis of an automatic grading system within first year computer science programming modules. In *Proceedings of the 5th Conference on Computing Education Practice* . ACM, New York, NY, 17–20. DOI: https://doi.org/10.1145/3437914.3437973

[125] David Insa, Sergio Pérez, Josep Silva, and Salvador Tamarit. 2021. Semiautomatic generation and assessment of Java exercises in engineering education. *Computer Applications in Engineering Education* 29, 5 (2021), 1034–1050. DOI: https://doi.org/10.1002/cae.22356 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22356

[126] David Insa and Josep Silva. 2018. Automatic assessment of Java code. *Computer Languages, Systems & Structures* 53 (2018), 59–72. DOI: https://doi.org/10.1016/j.cl.2018.01.004

[127] Julian Jansen, Ana Oprescu, and Magiel Bruntink. 2017. The impact of automated code quality feedback in programming education. In *Post-proceedings of the 10th Seminar on Advanced Techniques and Tools for Software Evolution* . Vol. 210.

[128] Gregor Jerše and Matija Lokar. 2017. Learning and teaching numerical methods with a system for automatic assessment. *International Journal for Technology in Mathematics Education* 24, 3 (2017), 121–127.

[129] An Ju, Ben Mehne, Andrew Halle, and Armando Fox. 2018. In-class coding-based summative assessments: Tools, challenges, and experience. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 75–80. DOI: https://doi.org/10.1145/3197091.3197094

[130] Angelika Kaplan, Jan Keim, Yves R. Schneider, Maximilian Walter, Dominik Werle, Anne Koziolek, and Ralf Reussner. 2020. Teaching Programming at Scale. *Software Engineering im Unterricht der Hochschulen (SEUH'20)*. Retrieved from https://ceur-ws.org/Vol-2531/paper01.pdf

[131] Oscar Karnalim and Simon. 2021. Promoting code quality via automated feedback on student submissions. In *Proceedings of the 2021 IEEE Frontiers in Education Conference*. 1–5. DOI: https://doi.org/10.1109/FIE49875.2021.9637193

[132] Remin Kasahara, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2019. Applying gamification to motivate students to write high-quality code in programming assignments. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 92–98. DOI: https://doi.org/10.1145/3304221.3319792

[133] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software* 175 (2021), 110905. DOI: https://doi.org/10.1016/j.jss.2021.110905

[134] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* . ACM, New York, NY, Article 4, 10 pages. DOI: https://doi.org/10.1145/3428029.3428043

[135] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 562–568. DOI: https://doi.org/10.1145/3408877.3432526

[136] Sándor Király, Károly Nehéz, and Olivér Hornyák. 2017. Some aspects of grading Java code submissions in MOOCs. *Research in Learning Technology* 25 (2017), 1–25. DOI: https://doi.org/10.25304/RLT.V25.1945

[137] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An automatic assessment management system for interactive learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 284–289. DOI: https://doi.org/10.1145/3159450.3159602

[138] Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic extraction of AST patterns for debugging student programs. In *Artificial Intelligence in Education.* Elisabeth André, Ryan Baker, Xiangen Hu, Ma. Mercedes T. Rodrigo, and Benedict du Boulay (Eds.), Springer International Publishing, Cham, 162–174. DOI: https://doi.org/10.1007/978-3-319-61425-0_14

[139] Duc Minh Le. 2022. Model-based automatic grading of object-oriented programming assignments. *Computer Applications in Engineering Education* 30, 2 (2022), 435–457. DOI: https://doi.org/10.1002/cae.22464 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22464

[140] Haden Hooyeon Lee. 2021. Effectiveness of real-time feedback and instructive hints in graduate CS courses via automated grading system. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 101–107. DOI: https://doi.org/10.1145/3408877.3432463

[141] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 158 (2018), 30 pages. DOI: https://doi.org/10.1145/3276528

[142] V. C. S. Lee, Y. T. Yu, C. M. Tang, T. L. Wong, and C. K. Poon. 2018. ViDA: A virtual debugging advisor for supporting learning in computer programming courses. *Journal of Computer Assisted Learning* 34, 3 (2018), 243–258. DOI: https://doi.org/10.1111/jcal.12238 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/jcal.12238

[143] Abe Leite and Saúl A. Blanco. 2020. Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 44–50. DOI: https://doi.org/10.1145/3328778.3366921

[144] Janet Liebenberg and Vreda Pieterse. 2018. Investigating the feasibility of automatic assessment of programming tasks. *Journal of Information Technology Education: Innovations in Practice* 17 (11 2018), 201–223. DOI: https://doi.org/10.28945/4150

[145] Simon Liénardy, Laurent Leduc, Dominique Verpoorten, and Benoit Donnet. 2020. Café: Automatic correction and feedback of programming challenges for a CS1 course. In *Proceedings of the 22nd Australasian Computing Education Conference* . ACM, New York, NY, 95–104. DOI: https://doi.org/10.1145/3373165.3373176

[146] David Liu and Andrew Petersen. 2019. Static analyses in Python programming courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 666–671. DOI: https://doi.org/10.1145/3287324.3287503

[147] Xiao Liu, Yeoneo Kim, Junseok Cheon, and Gyun Woo. 2019. A partial grading method using pattern matching for programming assignments. In *Proceedings of the 2019 8th International Conference on Innovation, Communication and Engineering.* 157–160. DOI: https://doi.org/10.1109/ICICE49024.2019.9117506

[148] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic grading of programming assignments: An approach based on formal semantics. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training.* 126–137. DOI: https://doi.org/10.1109/ICSE-SEET.2019.00022

[149] Zikai Liu, Tingkai Liu, Qi Li, Wenqing Luo, and Steven S. Lumetta. 2021. End-to-end automation of feedback on student assembly programs. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering* . 18–29. DOI: https://doi.org/10.1109/ASE51524.2021.9678837

[150] Evan Maicus, Matthew Peveler, Andrew Aikens, and Barbara Cutler. 2020. Autograding interactive computer graphics applications. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 1145–1151. DOI: https://doi.org/10.1145/3328778.3366954

[151] Evan Maicus, Matthew Peveler, Stacy Patterson, and Barbara Cutler. 2019. Autograding distributed algorithms in networked containers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 133–138. DOI: https://doi.org/10.1145/3287324.3287505

[152] Hamza Manzoor, Amit Naik, Clifford A. Shaffer, Chris North, and Stephen H. Edwards. 2020. Auto-grading Jupyter notebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 1139–1144. DOI: https://doi.org/10.1145/3328778.3366947

[153] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. 2017. Automated personalized feedback in introductory Java programming MOOCs. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering*. 1259–1270. DOI: https://doi.org/10.1109/ICDE.2017.169

[154] Igor Mekterović, Ljiljana Brkić, Boris Milašinović, and Mirta Baranović. 2020. Building a comprehensive automated programming assessment system. *IEEE Access* 8 (2020), 81154–81172. DOI: https://doi.org/10.1109/ACCESS.2020.2990980

[155] Eerik Muuli, Kaspar Papli, Eno Tõnisson, Marina Lepp, Tauno Palts, Reelika Suviste, Merilin Säde, and Piret Luik. 2017. Automatic assessment of programming assignments using image recognition. In *Data Driven Approaches in Digital Education*. Élise Lavoué, Hendrik Drachsler, Katrien Verbert, Julien Broisin, and Mar Pérez-Sanagustín (Eds.), Springer International Publishing, Cham, 153–163. DOI: https://doi.org/10.1007/978-3-319-66610-5_12

[156] Ramez Nabil, Nour Eldeen Mohamed, Ahmed Mahdy, Khaled Nader, Shereen Essam, and Essam Eliwa. 2021. EvalSeer: An intelligent gamified system for programming assignments assessment. In *Proceedings of the 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference*. 235–242. DOI: https://doi.org/10.1109/MIUCC52538.2021.9447629

[157] Bao-An Nguyen, Kuan-Yu Ho, and Hsi-Min Chen. 2020. Measure students' contribution in web programming projects by exploring source code repository. In *Proceedings of the 2020 International Computer Symposium*. 473–478. DOI: https://doi.org/10.1109/ICS51289.2020.00099

[158] Narges Norouzi and Ryan Hausen. 2018. Quantitative evaluation of student engagement in a large-scale introduction to programming course using a cloud-based automatic grading system. In *Proceedings of the 2018 IEEE Frontiers in Education Conference*. 1–5. DOI: https://doi.org/10.1109/FIE.2018.8658833

[159] Norbert Oster, Marius Kamp, and Michael Philippsen. 2017. AuDoscore: Automatic Grading of Java or Scala Homework. *Automatische Bewertung von Programmieraufgaben (ABP'17)*.

[160] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 92–97. DOI: https://doi.org/10.1145/3059009.3059026

[161] Hyunchan Park and Youngpil Kim. 2020. CLIK: Cloud-based Linux kernel practice environment and judgment system. *Computer Applications in Engineering Education* 28, 5 (2020), 1137–1153. DOI: https://doi.org/10.1002/cae.22289 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22289

[162] Beatriz Pérez. 2021. Enhancing the learning of database access programming using continuous integration and aspect oriented programming. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training*. 221–230. DOI: https://doi.org/10.1109/ICSE-SEET52601.2021.00032

[163] Giuseppina Polito and Marco Temperini. 2021. A gamified web based system for computer programming learning. *Computers and Education: Artificial Intelligence* 2 (2021), 100029. DOI: https://doi.org/10.1016/j.caeai.2021.100029

[164] Giuseppina Polito, Marco Temperini, and Andrea Sterbini. 2019. 2TSW: Automated assessment of computer programming assignments, in a gamified web based system. In *Proceedings of the 2019 18th International Conference on Information Technology Based Higher Education and Training*. 1–9. DOI: https://doi.org/10.1109/ITHET46829.2019.8937377

[165] Chung Keung Poon, Tak-Lam Wong, Chung Man Tang, Jacky Kin Lun Li, Yuen Tak Yu, and Victor Chung Sing Lee. 2018. Automatic assessment via intelligent analysis of students' program output patterns. In *Blended Learning. Enhancing Learning Success*. Simon K. S. Cheung, Lam-for Kwok, Kenichi Kubota, Lap-Kei Lee, and Jumpei Tokito (Eds.), Springer International Publishing, Cham, 238–250. DOI: https://doi.org/10.1007/978-3-319-94505-7_19

[166] Dhananjai M. Rao. 2019. Experiences with auto-grading in a systems course. In *Proceedings of the 2019 IEEE Frontiers in Education Conference* . 1–8. DOI: https://doi.org/10.1109/FIE43999.2019.9028450

[167] Ruan Reis, Gustavo Soares, Melina Mongiovi, and Wilkerson L. Andrade. 2019. Evaluating feedback tools in introductory programming classes. In *Proceedings of the 2019 IEEE Frontiers in Education Conference*. 1–7. DOI: https://doi.org/10.1109/FIE43999.2019.9028418

[168] Felipe Restrepo-Calle, Jhon J. Ramírez-Echeverry, and Fabio A. González. 2020. Using an interactive software tool for the formative and summative evaluation in a computer programming course: An experience report. *Global Journal of Engineering Education* 22, 3 (2020), 174–185.

[169] Arthur Rump, Ansgar Fehnker, and Angelika Mader. 2021. Automated assessment of learning objectives in programming assignments. *Intelligent Tutoring Systems (ITS'21)*. Lecture Notes in Computer Science, 299–309. DOI: https://doi.org/10.1007/978-3-030-80421-3_33

[170] Avneesh Sarwate, Creston Brunch, Jason Freeman, and Sebastian Siva. 2018. Grading at scale in EarSketch. In *Proceedings of the 5th Annual ACM Conference on Learning at Scale* . ACM, New York, NY, Article 50, 4 pages. DOI: https://doi.org/10.1145/3231644.3231708

[171] Kevin Sendjaja, Satrio Adi Rukmono, and Riza Satria Perdana. 2021. Evaluating control-flow graph similarity for grading programming exercises. In *Proceedings of the 2021 International Conference on Data and Software Engineering* . 1–6. DOI: https://doi.org/10.1109/ICoDSE53690.2021.9648464

[172] Saksham Sharma, Pallav Agarwal, Parv Mor, and Amey Karkare. 2018. TipsC: Tips and corrections for programming MOOCs. In *Artificial Intelligence in Education*. Carolyn Penstein Rosé, Roberto Martínez-Maldonado, H. Ulrich Hoppe, Rose Luckin, Manolis Mavrikis, Kaska Porayska-Pomsta, Bruce McLaren, and Benedict du Boulay (Eds.), Springer International Publishing, Cham, 322–326. DOI: https://doi.org/10.1007/978-3-319-93846-2_60

[173] Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. 2020. An open-source, API-based framework for assessing the correctness of code in CS50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 487–492. DOI: https://doi.org/10.1145/3341525.3387417

[174] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* . ACM, New York, NY, 263–272. DOI: https://doi.org/10.1145/2939672.2939696

[175] Marcus Soll, Melf Johannsen, and Chris Biemann. 2020. Enhancing a theory-focused course through the introduction of automatically assessed programming exercises-lessons learned. In *Proceedings of the Impact Papers at EC-TEL 2020*. Retrieved from https://ceur-ws.org/Vol-2676/paper6.pdf

[176] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. Context-aware and data-driven feedback generation for programming assignments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* . ACM, New York, NY, 328–340. DOI: https://doi.org/10.1145/3468264.3468598

[177] Fábio Rezende De Souza, Francisco De Assis Zampirolli, and Guiou Kobayashi. 2019. Convolutional neural network applied to code assignment grading. *CSEDU 2019 - Proceedings of the 11th International Conference on Computer Supported Education* 1 (2019), 62–69. DOI: https://doi.org/10.5220/0007711000620069

[178] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the design space of automatically synthesized hints for introductory programming assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems* . ACM, New York, NY, 2951–2958. DOI: https://doi.org/10.1145/3027063.3053187

[179] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing*. 107–115. DOI: https://doi.org/10.1109/VLHCC.2017.8103457

[180] Shao Tianyi, Kuang Yulin, Huang Yihong, and Quan Yujuan. 2019. PAAA: An implementation of programming assignments automatic assessing system. In *Proceedings of the 2019 4th International Conference on Distance Education and Learning* . ACM, New York, NY, 68–72. DOI: https://doi.org/10.1145/3338147.3338187

[181] Leo C. Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 738–744. DOI: https://doi.org/10.1145/3287324.3287463

[182] Sebastián Vallejos, Luis S. Berdun, Marcelo G. Armentano, Álvaro Soria, and Alfredo R. Teyseyre. 2018. Soploon: A virtual assistant to help teachers to detect object-oriented errors in students' source codes. *Computer Applications in Engineering Education* 26, 5 (2018), 1279–1292. DOI: https://doi.org/10.1002/cae.22021 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22021

[183] Quentin Vaneck, Thomas Colart, Benoît Frénay, and Benoît Vanderose. 2021. A tool for evaluating computer programs from students. In *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence* . ACM, New York, NY, 23–26. DOI: https://doi.org/10.1145/3472673.3473961

[184] Arjun Verma, Prateksha Udhayanan, Rahul Murali Shankar, Nikhila KN, and Sujit Kumar Chakrabarti. 2021. Source-code similarity measurement: Syntax tree fingerprinting for automated evaluation. In *Proceedings of the 1st International Conference on AI-ML Systems* . ACM, New York, NY, Article 8, 7 pages. DOI: https://doi.org/10.1145/3486001.3486228

[185] Jinshui Wang, Yunpeng Zhao, Zhengyi Tang, and Zhenchang Xing. 2020. Combining dynamic and static analysis for automated grading SQL statements. *Journal of Network Intelligence* 5, 4 (2020), 179–190.

[186] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* . ACM, New York, NY, 481–495. DOI: https://doi.org/10.1145/3192366.3192384

[187] Zhikai Wang and Lei Xu. 2019. Grading programs based on hybrid analysis. In *Web Information Systems and Applications*. Weiwei Ni, Xin Wang, Wei Song, and Yukun Li (Eds.), Springer International Publishing, Cham, 626–637. DOI: https://doi.org/10.1007/978-3-030-30952-7_63

[188] Dee A. B. Weikle, Michael O. Lam, and Michael S. Kirkpatrick. 2019. Automating systems course unit and integration testing: Experience report. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* . ACM, New York, NY, 565–570. DOI: https://doi.org/10.1145/3287324.3287502

[189] M. L. Wickramasinghe, H. P. Wijethunga, S. R. Yapa, D. M. D. Vishwajith, Udara Srimath S. Samaratunge Arachchillage, and Nelum Amarasena. 2020. Smart exam evaluator for object-oriented programming modules. In *Proceedings of the 2020 2nd International Conference on Advancements in Computing*. Vol. 1, 287–292. DOI: https://doi.org/10.1109/ICAC51239.2020.9357320

[190] Burkhard C. Wünsche, Zhen Chen, Lindsay Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. 2018. Automatic assessment of OpenGL computer graphics assignments. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* . ACM, New York, NY, 81–86. DOI: https://doi.org/10.1145/3197091.3197112

[191] Yi-Xiang Yan, Jung-Pin Wu, Bao-An Nguyen, and Hsi-Min Chen. 2020. The impact of iterative assessment system on programming learning behavior. In *Proceedings of the 2020 9th International Conference on Educational and Information Technology* . ACM, New York, NY, 89–94. DOI: https://doi.org/10.1145/3383923.3383939

[192] Y. T. Yu, C. M. Tang, and C. K. Poon. 2017. Enhancing an automated system for assessment of student programs using the token pattern approach. In *Proceedings of the 2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering*. 406–413. DOI: https://doi.org/10.1109/TALE.2017.8252370

[193] Lucas Zamprogno, Reid Holmes, and Elisa Baniassad. 2020. Nudging student learning strategies using formative feedback in automatically graded assessments. In *Proceedings of the 2020 ACM SIGPLAN Symposium on SPLASH-E* . ACM, New York, NY, 1–11. DOI: https://doi.org/10.1145/3426431.3428654

Received 19 June 2023; revised 2 November 2023; accepted 30 November 2023