





Creating an Academic Prometheus in Brazil: Weaving Check50, Autolab and MOSS into a Unified Autograder

Kevin Monteiro do Nascimento Ponciano¹^a, Abrantes Araújo Silva Filho¹^b,
Jean-Rémi Bourguet¹^c and Elias de Oliveira²^d

¹Department of Computer Science, Vila Velha University, Vila Velha, Brazil

²Postgraduate Program of Informatics (PPGI), Federal University of Espírito Santo, Vitória, Brazil

Keywords: Autograder, Programming Activities, Criteria-Based Evaluation, Virtual Environments, Plagiarism Detection.

Abstract: The evaluation of programming exercises submitted by a large volume of students presents an ongoing challenge for educators. As the number of students engaging in programming courses continues to rise, the burden of assessing their work becomes increasingly demanding. To address this challenge, automated systems known as autograders have been developed to streamline the evaluation process. Autograders recognize solutions and assign scores based on predefined criteria, thereby assisting teachers in efficiently assessing student programs. In this paper, we propose the creation of a comprehensive autograding platform in a Brazilian university by leveraging open-source technologies pioneered by prestigious universities such as Harvard, Carnegie Mellon, and Stanford. Job processing servers, interface components, and anti-plagiarism modules are integrated to provide educators with an evaluation tool, ensuring efficiency in grading processes and fostering enriched learning experiences. Through data analysis of the students' submissions, we aim to emphasize the platform's effectiveness and pinpoint areas for future enhancements to better cater to the needs of educators and students.


1 INTRODUCTION


With the rapid evolution of technology in recent times, there has been a significant increase in the demand for higher education courses related to the IT market. Fields such as Computer Science, Computer Engineering, and Information Systems are experiencing heightened interest, leading many universities to open new slots and accommodate larger class sizes. To meet this demand, a single professor often finds themselves responsible for multiple classes in IT-related undergraduate programs. On the other hand, when examining teaching methodologies, it becomes apparent that the majority of programming subjects follow a standard pattern. This involves explaining an algorithm, the theory behind it, and then instructing students to recreate the algorithm in a specific programming language (Bergin et al., 1996).


However, challenges arise when the professor needs to assess each student's programming activ-


ity, including evaluating compilation errors, syntax, adherence to instructions, and the overall quality of the code. Although it is the professor's responsibility to evaluate these activities, the process becomes overwhelming and exhausting as they spend numerous hours analyzing variations of the same algorithm. Fatigue can lead to overlooking errors or successes during this correction period. Swift feedback is crucial for students to understand their mistakes and learn how to improve. Given the number of students, limited class hours, and a packed curriculum, it becomes challenging for the professor to pinpoint these areas for all students (Au, 2011). If this assessment is done manually, it is very challenging to provide detailed evaluation with quick feedback for a large volume of students (Breslow et al., 2013).

Based on emerging trends such as the integration of adaptive learning technologies, utilization of artificial intelligence and machine learning, and focus on remote and online learning, new systems have been developed for the automatic recognition of potential solutions and mapping these solutions to scores assigned automatically based on criteria established by teachers. These automated systems are generally referred to as "Autograders" — see for example the

^a <https://orcid.org/0009-0001-2393-7424>

^b <https://orcid.org/0009-0008-0121-7566>

^c <https://orcid.org/0000-0003-3686-1104>

^d <https://orcid.org/0000-0003-2066-7980>

proposal in Nordquist (2007). One of the objectives of this work is to develop an autograder for the Vila Velha University (UVV) capable of assisting teachers in the standardized and detailed evaluation of student programs.

Another serious issue found in student programs is plagiarism materialized by the act of copying code. Even if a course has a clear and rigorous academic integrity policy, students often copy program code from each other (and/or copy code from the internet), submitting the copied program as if it were their own creation. Plagiarism is a serious problem because it prevents the teacher from knowing if the class is learning the material and which aspects of the content were challenging for students and need reinforcement in class. It is also very difficult for the teacher to manually inspect all codes of all students and determine if a program is original or plagiarized, as the number of unique pairs that may contain plagiarism increases quadratically (Heres and Hage, 2017). It is essential for the teacher to be able to identify and appreciate original solutions produced by students and penalize the plagiarism of program codes. Automatic methods for measuring similarity between program codes have been used for many years to help humans detect plagiarism (Clough et al., 2003), and there are already approaches to identify similarities between specific programming codes (e.g., C (Sharrock et al., 2019), SQL (Hu et al., 2022)) or graphs produced in conceptual modeling (e.g., ERD (Del Pino Lino and Rocha, 2018), UML (Ionita et al., 2013)). There are already specific systems for detecting plagiarism in program code that provide the teacher with a visually and easily interpretable report indicating the likelihood of plagiarism in student codes (John and Boateng, 2021). Therefore, another objective of this work is to integrate automated plagiarism detection tools into the autograder that will be produced to assist the teacher.

Many prestigious universities, including Harvard University and Carnegie Mellon University, have developed their own tools for code self-correction in their Computer Science courses, offering open-source solutions to construct comprehensive and tailored autograding systems. This paper introduces the creation of a self-correction platform leveraging open-source technologies such as “check50”¹ (Sharp et al., 2020), developed by Harvard University, serving as a framework for assessing correctness of code, “Autolab”² (Milojicic, 2011), a project originating from Carnegie Mellon University, employed as the front-end component, “gradelab50”³, a tool to grade a stu-

dent’s submission based on check50’s json report and a given grading scheme, and “MOSS”⁴ (Schleimer et al., 2003), created at Stanford University, as the anti-plagiarism module within the system. This paper delves into the collaborative integration of these tools, offering insights into the development and functionality of a versatile and adaptable autograding solution.

The remainder of the paper is structured as follows: In Section 2, we will present some related works. In Section 3, we will introduce the components we integrated. In Section 4, we will describe the architecture system of our proposal. In Section 5, we will present some student feedback about the autograder. Finally, in Section 6, we will conclude and outline some perspectives.

2 RELATED WORK

The landscape of autograders, tools designed to automate the grading and evaluation of student programming assignments, encompasses a diverse array of solutions and methodologies.

Barlow et al. (2021) present a survey of the most popular currently available autograders, reflecting the growing interest and adoption of automated grading systems in educational settings.

One prominent example is AutoGrader, a framework developed by Helmick (2007) at Miami University for the automatic evaluation of student programming assignments written in Java. Notably, AutoGrader supports static code analysis through tools like PMD, enabling the detection of inefficiencies, bugs, and suboptimal coding practices. While initially tailored for Java, the underlying principles of automated grading are adaptable to multiple programming languages.

OverCode, introduced by Glassman et al. (2015), presents a novel approach to visualizing and exploring large sets of programming solutions. Through a combination of static and dynamic analysis, OverCode clusters similar solutions, providing educators with insights into students’ problem-solving approaches.

Anticipating common mistakes made by novice programmers, Hogg and Jump (2022) develop test suites integrated with autograders to provide understandable failure messages, enhancing the learning experience.

Sridhara et al. (2016) employ fuzz testing to identify behavioral discrepancies between student solutions and reference implementations, enhancing the robustness of autograding systems.

¹<https://github.com/cs50/check50>

²<https://github.com/autolab/Autolab>

³<https://pypi.org/project/gradelab50/>

⁴<http://theory.stanford.edu/~aiken/moss/>

Integrating autograders with Learning Management Systems (LMS) or Massive Open Online Course (MOOC) platforms has been explored in various works (Danutama and Liem, 2013; Norouzi and Hausen, 2018; Sharrock et al., 2019; Calderón et al., 2020; Ureel II and Wallace, 2019).

While traditional autograders primarily evaluate based on passed tests, Liu et al. (2019) propose approaches to handle semantically different execution paths between student submissions and reference implementations, addressing nuanced evaluation scenarios.

The issue of plagiarism detection within autograders has also received attention (zu Eissen and Stein, 2006; Ali et al., 2011; Apriyani et al., 2020), with methodologies ranging from text-based detection to code similarity analysis, contributing to academic integrity in programming education.

Furthermore, autograders have found applications beyond traditional coursework, including competitive programming (Arifin and Perdana, 2019), large-scale programming classes (Sharrock et al., 2019), and program repairs (Gulwani et al., 2018), block-based coding assignments (Damle et al., 2023) or exam environment (Ju et al., 2018), illustrating their versatility and utility across diverse educational contexts.

Recognized benefits of autograders include improved student-tutor interactions, enhanced course quality, increased learning success, and improved code quality (Norouzi and Hausen, 2018; Marwan et al., 2020; Hagerer et al., 2021), underlining their positive impact on programming education.

Autograders are also able to capture the formative steps that were involved in the development of the final submission (Acuña and Bansal, 2022), providing valuable insights into the iterative learning process undertaken by students.

Recent research efforts have focused on refining autograder functionalities, such as implementing penalty schemes to encourage reflective feedback engagement (Leinonen et al., 2022) and advocating policy changes in submission formats (Butler and Herman, 2023). Additionally, developments like real-time actionable feedback on code style (Choudhury et al., 2016) further augment the pedagogical value of autograders.

In summary, the evolution and proliferation of autograders have significantly transformed programming education, offering scalable, efficient, and insightful assessment mechanisms.

3 COMPONENTS INTEGRATION

The choice of Autolab, check50, gradelab50, and MOSS is due to their specialized functions for education. Autolab simplifies everything with its simple interface, offering fast feedback and assisting in course and assessment organization. check50 allows for the quick creation of *checks* to assess the correctness of student code, and is supported by an active community. gradelab50 is used to grade a student's submission based on check50's json report and a given grading scheme, and outputs a json report in a format that can be used by Autolab. Finally, MOSS helps detect plagiarism for free and accurately, ensuring honesty in submissions.

3.1 Check50 Integration

check50 is a tool for checking student code introduced in 2012 in CS50 at Harvard⁵ that provides a simple, functional framework for writing checks (Sharp et al., 2020). check50 allows teachers to automatically grade code on correctness and to provide automatic feedback while students are coding. It is a correctness-testing tool made available to both students and teachers, automatically runs a suite of tests against students' code to evaluate the correctness of each submission.

As a result, check50 has allowed us to provide students with immediate feedback on their progress as they complete an assignment while also facilitating automatic and consistent grading, allowing teaching staff to spend more time giving tailored, qualitative feedback. check50 consists of a tool divided into two parts. The first part encompasses the source code of the tool, which incorporates verification methods, reading input files, formatting text outputs, as well as APIs and other elements that ensure the full functioning of the library. The second part consists of validators (checks), which allow educators, following a standard structure, to develop specific problems. These include validation steps, customization of feedback for approved or rejected steps, and the creation of a model code that check50 uses to determine the correctness of the code submitted by the student.

A great advantage of check50 is its modularity, allowing a complete separation between the tool and its validators. This enables any educator to use pre-existing validators, developed by the community, or create their own validators. These can be executed both online, with the validators hosted on GitHub, and locally, in a check50 installation.

⁵<https://www.edx.org/cs50>

Creating validators can be done locally or through repositories on GitHub, without the need to be on the same machine where check50 is installed. To build a basic validator, only a .yaml file named .cs50.yaml is required, which defines the name of the validation, the command to be executed, the expected output, and the required exit code as described in Listing 1. Note that for more complex validations, it is recommended to use Python in a file named __init__.py.

```
check50:
  checks:
    ola: # define a check named ola
    - run: python3 ola.py # runs ola.py
      stdout: Olá! # expect Olá! in
        stdout
      exit: 0 # expect exitcode 0
    olas: # define a check named olas
    - run: python3 olas.py # runs olas.py
      stdin: 2 # insert 2 into
        stdin
      stdout: ola ola # ola ola in
        stdout
      exit: 0 # expect exitcode
        0
```

Listing 1: Simple checks.

The Listing 2 illustrates advanced checks, initially verifying the compilation of the ola.c file. Subsequently, upon successful compilation, it proceeds to validate whether the file appropriately outputs the expected message.

```
import check50
import check50.c

@check50.check()
def exists():
    """O arquivo ola.c existe?"""
    check50.exists("ola.c")

@check50.check(exists)
def compiles():
    """Verifica se o arquivo ola.c compila:"""
    check50.c.CFLAGS={'ggdb': True, 'lm': True, 'std': 'c17', 'Wall': True, 'Wpedantic': True}
    check50.c.compile("ola.c", exe_name="ola", cc="gcc", max_log_lines=50, lcs50=True)

@check50.check(compiles)
def uvv():
    """Responde corretamente ao nome UVV?"""
    check50.run("./ola").stdin("UVV").stdout("Ola, UVV!").exit()

@check50.check(compiles)
def kevin():
    """Responde corretamente ao nome Kevin?"""
    check50.run("./ola").stdin("Kevin").stdout("Olá, Kevin!").exit()
```

Listing 2: Advanced checks.

Additionally, it is possible to configure check50 in detail through the .cs50.yaml file as described in Listing 3, specifying which files will be submitted to the tests and which will be excluded, as well as including test dependencies, such as external libraries that will be installed during the execution of check50.

```
check50:
  files: &check50_files
  - !exclude "*"
  - !require ola.c
```

Listing 3: Test details.

To locally execute a test, use the command: check50 --dev path/to/check/ -o json on the files specified in the .cs50.yaml. It will generate a JSON file with both successful and failed tests, as described in Listing 4.

```
{
  "slug": "../autograder/.check50/",
  "results": [
    {
      "name": "existe",
      "description": "ola.c existe?",
      "passed": true,
      "log": [
        "checking that ola.c exists..." ],
      "cause": null,
      "data": {},
      "dependency": null
    },
    {
      "name": "compila",
      "description": "ola.c compila?",
      "passed": true,
      "log": [
        "running gcc ola.c -o ola..." ],
      "cause": null,
      "data": {},
      "dependency": "existe"
    },
    {
      "name": "resposta",
      "description": "correto?",
      "passed": true,
      "log": [
        "running ./ola...",
        "checking for output Olá!...",
        "checking exited with 0..." ],
      "cause": null,
      "data": {},
      "dependency": "compila"
    }
  ],
  "version": "3.3.7"
}
```

Listing 4: JSON file results.

3.2 Autolab Integration

Autolab is an open source course management and autograding service started at Carnegie Mellon by Professor David O'Hallaron (see Milojicic (2011)). Many courses from other schools including University of Washington, Peking University, Cornell Uni-

versity, among others use the service. Autolab consists of two main components: a Ruby on Rails web app, and Tango, a Python job processing server. The web app offers a full suite of course management tools including scoreboards, configurable assignments, PDF and code handouts, grade sheets, and plagiarism detection. The job processing server accepts job requests to run students' code along with an instructor written autograding script within a virtual machine.

After receiving the submitted files, the Autolab frontend forwards them to Tango, a backend system, through HTTP requests. Tango, in turn, inserts these files into a job queue, preparing them for evaluation. The assignment of jobs to available containers or virtual machines is done via SSH, ensuring that each submission is processed in an isolated and secure environment. This system allows Tango to efficiently direct jobs through the evaluation process. Tango has the ability to direct jobs to appropriate VM instances based on the corresponding course. For example, jobs for course CS1 are exclusively directed to the CS1 VM instance, ensuring that the evaluation takes place in the most suitable environment.

After a job is completed, the feedback is transferred back to Tango via SSH. This feedback is then forwarded to the Autolab frontend through HTTP requests. The frontend is responsible for presenting the comments to users, either through the browser or CLI. Additionally, the system can update grades on the student's report card, if necessary, and store comments in the database for future access.

3.3 MOSS Integration

MOSS (for a Measure Of Software Similarity) is an automatic system for determining the similarity of programs introduced in 1994 at Stanford (Schleimer et al., 2003). The main application of MOSS is actually used to detect plagiarism in programming classes. The algorithm behind MOSS is a significant improvement over other cheating detection algorithms. It saves teachers and teaching staff a lot of time by pointing out the parts of programs that are worth a more detailed examination.

The tool is developed to identify similarities between codes, but it does not have the ability to automatically determine if one code is a copy of another. The responsibility to analyze the similarities detected by MOSS falls on the educator. The developers suggest using MOSS as a resource to assess the amount of similarity between codes, helping to identify unusual correspondences that may require further investigation.

The implementation of MOSS is facilitated by a bash configuration script, which allows users to submit their code for analysis by the MOSS server, simplifying the submission process. Users can specify the programming language of the tested codes using the `-l` option. This allows for a more accurate analysis, as MOSS supports various languages. In Listing 5, Lisp programs are compared for example.

```
moss -l lisp foo.lisp bar.lisp
```

Listing 5: Moss example use.

Moreover, the `-d` option signifies that submissions are structured by directory, considering files within the same directory as components of a unified program, as presented in Listing 6. This comparison involves programs composed of both `.c` and `.h` files.

```
moss -d foo/*.c foo/*.h bar/*.c bar/*.h
```

Listing 6: Moss example use.

A crucial aspect in similarity analysis is to avoid considering as plagiarism the code that is common to all students, such as the code provided by the instructor. This is done through the `-b` option, which specifies a base file, excluding from the report the code also present in this file. To adjust the system's sensitivity, the `-m` option sets the maximum number of times a code snippet can appear before being disregarded, helping to differentiate between legitimate sharing and potential plagiarism. Finally, the `-n` option sets the number of corresponding files to be shown in the results. Using these options provides users with significant flexibility in the use of MOSS, allowing for adjustments as needed to obtain more precise and relevant similarity analyses, facilitating the identification of unusual correspondences that may require further investigation.

The similarity scores produced by MOSS are useful for judging the relative amount of matching between different pairs of programs and for more easily seeing which pairs of programs stick out with unusual amounts of matching. But the scores are certainly not a proof of plagiarism.

4 SYSTEM ARCHITECTURE

In order to use check50, Autolab and MOSS together, some modifications were necessary. In Figure 1, we illustrate the system architecture diagram detailing the components of our Brazilian Autograder solution.

Students initiate the evaluation process by submitting their code, engaging in an interactive assessment

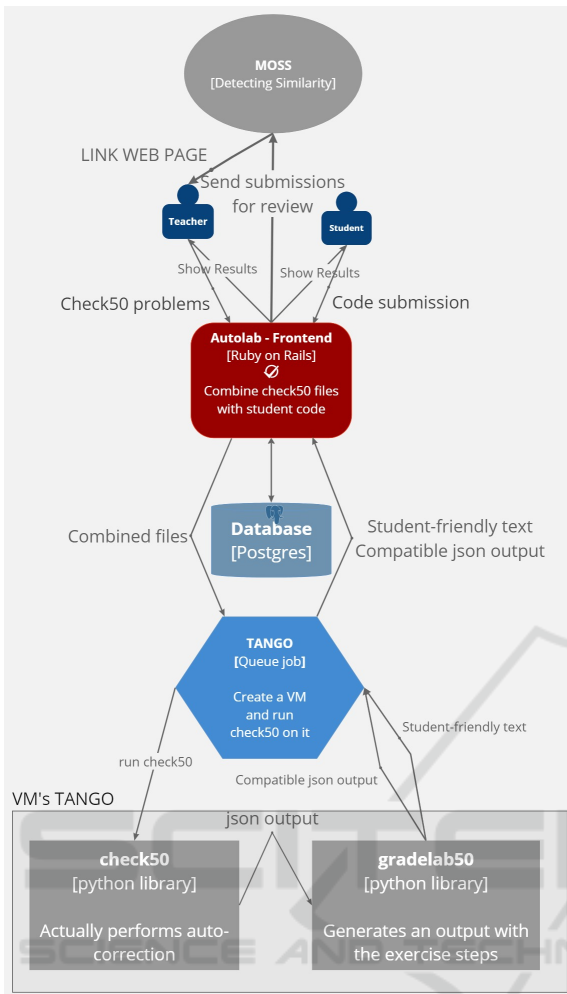


Figure 1: System architecture diagram.

facilitated by Autolab. Autolab acts as the intermediary, for combining check50 files with student code to create a consolidated files database. Tango then orchestrates the creation of a Virtual Machine (VM) and executes check50 within it, showcasing the back-end processing of code submissions. This pivotal step ensures the execution of automated tests on student submissions within a controlled environment. The process concludes with two concurrent outcomes: the Tango JSON output from the VM and the operation of check50 and gradelab50. These components are responsible for executing auto-correction and generating outputs detailing the exercise steps. Additionally, teachers have the option to submit assignments for review, enabling plagiarism detection or similarity checking and maintaining academic integrity and originality in student work.

The course home page of our autograder is depicted in Figure 2.



Figure 2: Course home page.

Figure 3 illustrates the submission page of our autograder.

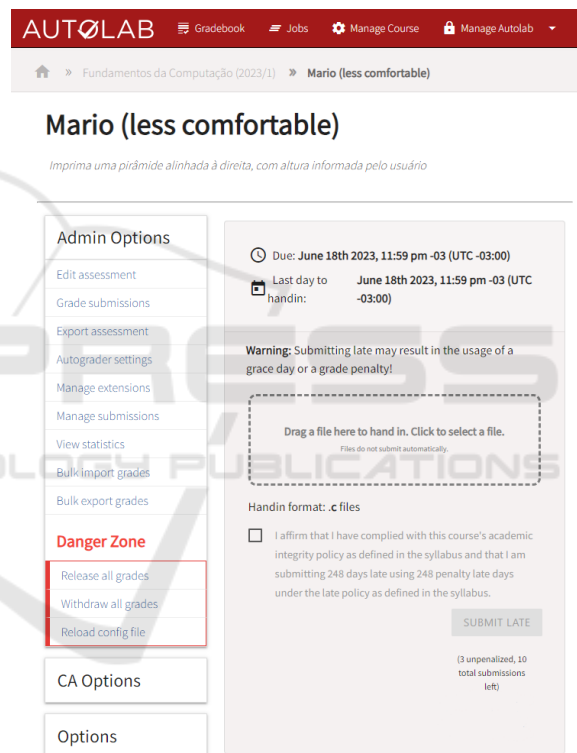


Figure 3: Submission page.

In the VMs created by Tango, it is possible to perform any type of self-assessment in numerous programming languages, as long as the test output adheres to the required JSON format for Autolab to interpret and render on the frontend whether the test passed, failed, the grade for each step, and any necessary hints.

We first modified the Dockerfile of Tango's VMs to install check50. Then, we modified the Makefiles that Autolab executes to start the autograder. Therefore, in addition to running a simple Python test, for example, it can also run check50.

```

all:
tar xvf autograde.tar
cp -r credit.py src
(cd src; python3 driver.py;)
clean:
rm -rf *\~ src

```

Listing 7: Default Makefile.

```

all:
tar xvf autograde.tar
-check50 --dev /src/ -o json
clean:
rm -rf *\~ src

```

Listing 8: Modified Makefile with check50.

In the Makefile presented in Listing 7, the systems extracts the necessary files to perform the test, copies the file submitted by the student to the src folder, and then executes the Python code responsible for auto-correction.

In the second Makefile presented in Listing 8, the systems simply extracts the folder already structured with the necessary files to perform the auto-correction following the check50 standard, and then executes check50.

Consequently, Autolab performs self-corrections for each code submitted by the student using check50. However, at this stage, we encountered a conflict between the two components. Autolab expects the test to return a JSON with the problems passed in the evaluation with their respective grades, while check50 returns a completely different JSON and without the necessary grades for Autolab to register them on the student's report card.

To bridge this gap and tackle this incompatibility, we used grade50, developed by Professor Patrick Totzke⁶ from the University of Liverpool. grade50 reads the JSON report produced by check50 and, based on parameters established in a .yaml file, generates a textual output. This output can be formatted either in a .jinja2 template file or in a JSON format.

Listing 9 defines a scoring schema for two main categories: "Compilation" and "Correctness". Each category contains specific checks, with points assigned per test. Custom comments are defined for each possible outcome, providing clear and targeted feedback to students.

⁶<https://github.com/pazz/grade50>

```

- name: "Compilação"
  checks:
    - name: "existe"
      points: 20
      fail_comment: "FALHOU (0/20 pontos):
                    Arquivo ola.c não encontrado."
      pass_comment: "PASSOU (20/20 pontos):
                    O arquivo ola.c existe."
    - name: "compila"
      points: 30
      fail_comment: "FALHOU (0/30 pontos):
                    Arquivo não compila."
      pass_comment: "PASSOU (30/30 pontos):
                    Sucesso: arquivo compilado."

- name: "Corretude"
  checks:
    - name: "resposta"
      points: 50
      fail_comment: "FALHOU (0/50 pontos):
                    O output não está correto."
      pass_comment: "PASSOU (50/50 pontos):
                    O output está correto."

```

Listing 9: Scoring schema.

The output generated by grade50 follows the template of the .jinja2 file. It provides a summary of the points obtained per test group, followed by details on specific comments for each group. This structure facilitates the understanding of results by students and teachers, highlighting areas of success and areas needing improvement.

grade50 greatly improves check50's capabilities by incorporating a quantitative aspect to code evaluation, thereby enriching the learning experience through comprehensive insights into student performance. However, the output of grade50 also differs from the JSON expected by Autolab. With the authorization of Professor Patrick Totzke, we modified grade50 into a component that we called gradelab50 to produce an output exactly in the format expected by Autolab as presented in Figure 4.

After these minor changes, check50 performs self-assessments of student submissions, gradelab50 retrieves the output generated by check50, scores the submissions according to the successfully passed steps, and Autolab records the grades on the students' report card and provides the feedback generated by gradelab50 to the students.

Autolab streamlines MOSS usage by incorporating it directly into its platform, replacing command-line instructions with a user-friendly interface. Within this interface, users only need to specify certain configurations, such as file compression status, file language, and the base file selection. Once these options are defined, files can be uploaded via the web browser for analysis and submission as depicted in Figure 5.

Autolab's backend manages the execution of the pre-indexed MOSS script on the server. Upon com-



Figure 4: Correction Feedback Page.

pletion of the script, users are directed to the MOSS webpage, where they can review the submitted files and identify potential instances of plagiarism among students as shown in Figure 6.

5 BENEFITS AND DRAWBACKS

In Fall 2023, 1,144 submissions were made by 82 students (average of 14 submissions/student), marking an unprecedented volume of submissions. This contrasts with the period when teachers were required to manually execute and correct student work. The tasks performed by the students were basically of two broad types: simple exercises, so that students could develop fundamental programming skills, and PSETs (Problem Sets), so that students could develop computational thinking skills and train the ability to solve problems. The exercises were developed by the subject teachers, and the PSETs were adapted from Har-

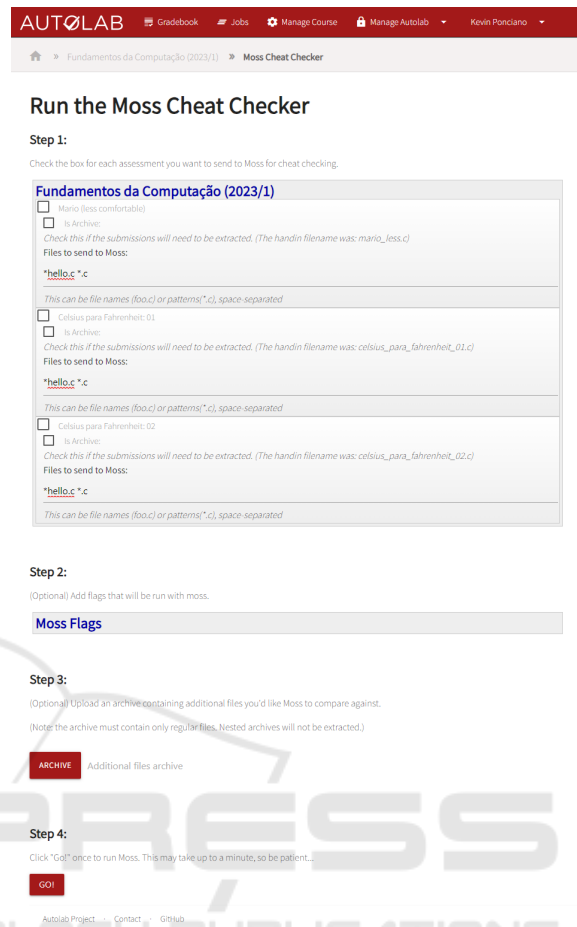


Figure 5: Graphical User Interface of Autolab with MOSS.

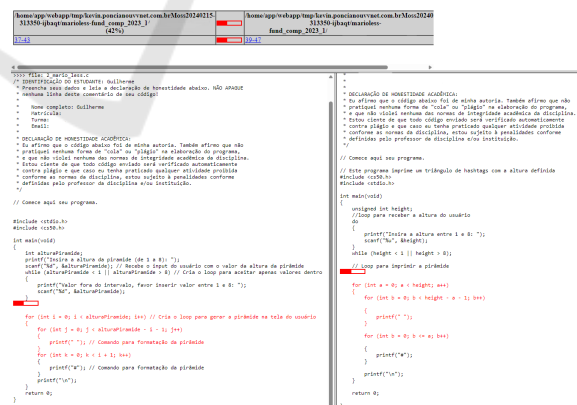


Figure 6: MOSS webpage.

vard Course CS50x⁷.

The PSETs were translated into Brazilian Portuguese and some were adapted to reflect the reality

⁷<https://cs50.harvard.edu/x/2023/>



Figure 7: American (above) x Brazilian (below) coins.

in Brazil. For example, in the original PSET Cash⁸, American coins were used, and in the translated and adapted PSET, Brazilian coins were used, as shown in Figure 7. All check50 “checks” have been rewritten to reflect the adapted version of the PSETs.

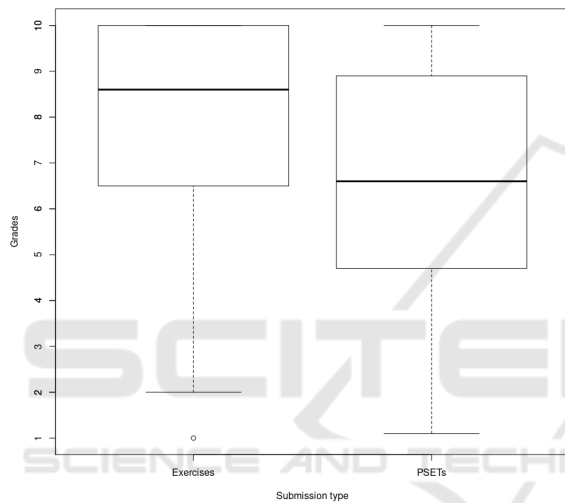


Figure 8: Grades by submission type.

As expected, students’ grades (on a scale from 0 to 10) were better in the simple exercises and lower in the PSETs, as depicted in Figure 8, considering all submissions and all exercises and PSETs.

When we analyzed a specific programming exercise, for example, the calculation of body mass index, we noticed that, as expected, students’ grades were higher according to the number of submissions they make to Autolab, as described in Figure 9.

At the moment we did not limit the number of submissions that a student could make: he could submit as many times as he wanted, until he got a good grade. The student realized the first submission, received feedback from Autolab, debugged his program and submitted it again, until he succeeded.

Our autograder solution has helped our teachers to improve the consistency and the efficiency of grading student’s assignments. Furthermore, the autograder allowed teachers to spend more time on qualitative

⁸<https://cs50.harvard.edu/x/2023/psets/1/cash/>

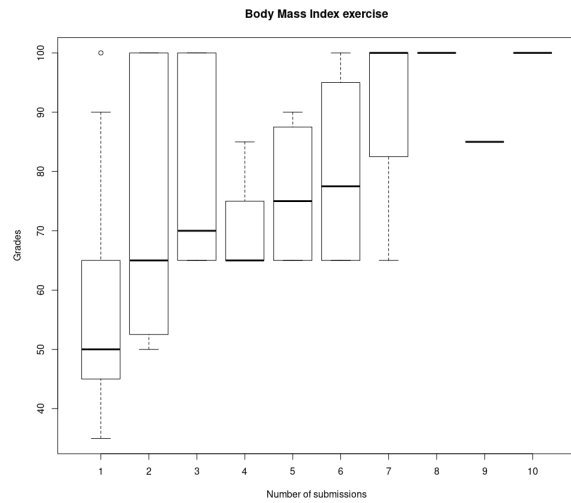


Figure 9: Grades by number of submissions.

feedback for students.

Another clear benefit was that, by reducing the need for manual code correction, teachers were able to take advantage of the time saved to enhance programming exercises and develop new exercises designed to assess specific student skills.

Students in our CS1 course have also taken advantage of the instantaneous feedback on code’s correctness provided by check50, and students’ motivation level has increased with informal and friendly competition through the anonymous grade scoreboard.

The interface provided by Autolab, which allows the teacher to make notes directly on the student’s code, was cited by the students themselves as one of the system’s most useful features.

In general, our platform allowed teachers and students to optimize their time and make better use of interaction periods.

However, some problems occurred. We have found that some students have begun to use our autograder as a debug tool, in a kind of trial and error coding, instead of thinking about the problem and creating a solution. These students were unable to establish healthy habits for planning, compiling, running and debugging their programs. Despite being real problems, they are pedagogically manageable during office hours or other direct meetings with students.

Another limitation we faced pertained to the analysis of student data. Although Autolab provides several tools for reporting and simple data analysis, including monitoring of low performance students, there is no easy way to export data in a format more suitable to our needs. The analysis of the grades we show was actually based on manual data collection and verification, a process that is slow and prone to errors. In future work, one of our priorities will be to

gain a deeper understanding of how Autolab stores all data and to develop routines for retrieving data that are best suited to our needs.

6 CONCLUSIONS

One of the crucial aspects in the teaching-learning process of computer science disciplines is the learning assessment phase and the evaluation of codes written by students. Ideally, this assessment should be standardized and detailed, with students receiving prompt feedback on what is correct and/or incorrect. However, teachers face a significant challenge in achieving standardized and detailed evaluation with quick feedback in classes (in-person or virtual) with a large number of students. The main issue is the time it takes for the professor to provide feedback to the students.

In this paper, we have discussed the development and integration of a comprehensive autograding system tailored for programming assignments in educational settings. To mitigate these challenges, we proposed the integration of automated tools, including check50, Autolab, and MOSS, to streamline the grading process and enhance the overall learning experience. Our approach leverages open-source technologies and existing frameworks developed by renowned institutions, such as Harvard University and Carnegie Mellon University. By combining these tools into a cohesive autograding platform, we aimed to provide educators with efficient means to assess student programs, detect plagiarism, and deliver timely feedback.

Our evaluation of the system's performance showcased significant improvements in grading efficiency and consistency, as evidenced by the substantial increase in the number of submissions processed within a semester. Generally, the feedbacks provided by an autograder prior to students' submission of each assignment compel students to spend more time debugging and allow the students to obtain higher correctness scores (Norouzi and Hausen, 2018; Sharp et al., 2020).

Future work may also involve further refining the integration of supplementary features, such as peer-review by students (Silva et al., 2020a), or incorporating intelligent tutoring systems based on Item Response Theory (Silva et al., 2020d,c,b). These enhancements aim to augment the pedagogical efficacy of a more comprehensive and multi-functional autograder.

REFERENCES

- Acuña, R. and Bansal, A. (2022). Using programming auto-grader formative data to understand student growth. In *IEEE Frontiers in Education Conference, FIE 2022, Uppsala, Sweden, October 8-11, 2022*, pages 1–8. IEEE.
- Ali, A. M. E. T., Abdulla, H. M. D., and Snásel, V. (2011). Overview and comparison of plagiarism detection tools. In Snásel, V., Pokorný, J., and Richta, K., editors, *Proceedings of the DATESO 2011: Annual International Workshop on Databases, TExts, Specifications and Objects, Pisek, Czech Republic, April 20, 2011*, volume 706 of *CEUR Workshop Proceedings*, pages 161–172. CEUR-WS.org.
- Apriyani, N., Atmadja, A. R., Fuadi, R. S., and Gerhana, Y. A. (2020). The detector of plagiarism in autograder for php programming languages. In *ICONISTECH-1 2019: Selected Papers from the 1st International Conference on Islam, Science and Technology, ICONISTECH-1 2019, 11-12 July 2019, Bandung, Indonesia*, page 205. European Alliance for Innovation.
- Arifin, J. and Perdana, R. S. (2019). Ugrade: Auto-grader for competitive programming using contestant pc as worker. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.
- Au, W. (2011). Teaching under the new taylorism: high-stakes testing and the standardization of the 21st century curriculum. *Journal of Curriculum Studies*, 43:25–45.
- Barlow, M., Cazalas, I., Robinson, C., and Cazalas, J. (2021). Mocsid: an open-source and scalable online ide and auto-grader for introductory programming courses. *Journal of Computing Sciences in Colleges*, 37(5):11–20.
- Bergin, J., Roberts, J., Pattis, R., and Stehlik, M. (1996). *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., USA, 1st edition.
- Breslow, L., Pritchard, D. E., DeBoer, J., Stump, G. S., Ho, A. D., and Seaton, D. T. (2013). Studying learning in the worldwide classroom research into edx's first mooc. *Research & Practice in Assessment*, 8:13–25.
- Butler, L. and Herman, G. L. (2023). First try, no (auto-grader) warm up: Motivating quality coding submissions. In *2023 ASEE Annual Conference & Exposition*.
- Calderón, D., Petersen, E., and Rodas, O. (2020). Salp: A scalable autograder system for learning programming—a work in progress. In *2020 IEEE Integrated STEM Education Conference (ISEC)*, pages 1–4. IEEE.
- Choudhury, R. R., Yin, H., and Fox, A. (2016). Scale-driven automatic hint generation for coding style. In Micarelli, A., Stamper, J. C., and Panourgia, K., editors, *Intelligent Tutoring Systems - 13th International Conference, ITS 2016, Zagreb, Croatia, June 7-10, 2016. Proceedings*, volume 9684 of *Lecture Notes in Computer Science*, pages 122–132. Springer.

- Clough, P. et al. (2003). Old and new challenges in automatic plagiarism detection. *National plagiarism advisory service*, 41:391–407.
- Damle, P., Bull, G., Watts, J., and Nguyen, N. R. (2023). Automated structural evaluation of block-based coding assignments. In Doyle, M., Stephenson, B., Dorn, B., Soh, L., and Battestilli, L., editors, *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 2, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, page 1300. ACM.
- Danutama, K. and Liem, I. (2013). Scalable autograder and lms integration. *Procedia Technology*, 11:388–395.
- Del Pino Lino, A. and Rocha, A. (2018). Automatic evaluation of erd in e-learning environments. In *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5.
- Glassman, E. L., Scott, J., Singh, R., Guo, P. J., and Miller, R. C. (2015). Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput. Hum. Interact.*, 22(2):7:1–7:35.
- Gulwani, S., Radiček, I., and Zuleger, F. (2018). Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480.
- Hagerer, G., Lahesoo, L., Anschutz, M., Krusche, S., and Groh, G. (2021). An analysis of programming course evaluations before and after the introduction of an autograder. In *19th International Conference on Information Technology Based Higher Education and Training, ITHET 2021, Sydney, Australia, November 4-6, 2021*, pages 1–9. IEEE.
- Helmick, M. T. (2007). Interface-based programming assignments and automatic grading of java programs. In Hughes, J. M., Peiris, D. R., and Tymann, P. T., editors, *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2007, Dundee, Scotland, UK, June 25-27, 2007*, pages 63–67. ACM.
- Heres, D. and Hage, J. (2017). A quantitative comparison of program plagiarism detection tools. In *Proceedings of the 6th Computer Science Education Research Conference*, page 73–82, New York, NY, USA. Association for Computing Machinery.
- Hogg, C. and Jump, M. (2022). Designing autograders for novice programmers. In Merkle, L., Doyle, M., Sheard, J., Soh, L., and Dorn, B., editors, *SIGCSE 2022: The 53rd ACM Technical Symposium on Computer Science Education, Providence, RI, USA, March 3-5, 2022, Volume 2*, page 1200. ACM.
- Hu, Y., Miao, Z., Leong, Z., Lim, H., Zheng, Z., Roy, S., Stephens-Martinez, K., and Yang, J. (2022). I-rex: An interactive relational query debugger for SQL. In Merkle, L., Doyle, M., Sheard, J., Soh, L., and Dorn, B., editors, *SIGCSE 2022: The 53rd ACM Technical Symposium on Computer Science Education, Providence, RI, USA, March 3-5, 2022, Volume 2*, page 1180. ACM.
- Ionita, A. D., Cernian, A., and Florea, S. (2013). Automated UML model comparison for quality assurance in software engineering education. *eLearning & Software for Education*.
- John, S. and Boateng, G. (2021). "i didn't copy his code": Code plagiarism detection with visual proof. In Roll, I., McNamara, D. S., Sosnovsky, S. A., Luckin, R., and Dimitrova, V., editors, *Artificial Intelligence in Education - 22nd International Conference, AIED 2021, Utrecht, The Netherlands, June 14-18, 2021, Proceedings, Part II*, volume 12749 of *Lecture Notes in Computer Science*, pages 208–212. Springer.
- Ju, A., Mehne, B., Halle, A., and Fox, A. (2018). In-class coding-based summative assessments: tools, challenges, and experience. In Polycarpou, I., Read, J. C., Andreou, P., and Armoni, M., editors, *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, Larnaca, Cyprus, July 02-04, 2018*, pages 75–80. ACM.
- Leinonen, J., Denny, P., and Whalley, J. (2022). A comparison of immediate and scheduled feedback in introductory programming projects. In Merkle, L., Doyle, M., Sheard, J., Soh, L., and Dorn, B., editors, *SIGCSE 2022: The 53rd ACM Technical Symposium on Computer Science Education, Providence, RI, USA, March 3-5, 2022, Volume 1*, pages 885–891. ACM.
- Liu, X., Wang, S., Wang, P., and Wu, D. (2019). Automatic grading of programming assignments: an approach based on formal semantics. In Beecham, S. and Damian, D. E., editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 126–137. IEEE / ACM.
- Marwan, S., Gao, G., Fisk, S. R., Price, T. W., and Barnes, T. (2020). Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In Robins, A. V., Moskal, A., Ko, A. J., and McCauley, R., editors, *ICER 2020: International Computing Education Research Conference, Virtual Event, New Zealand, August 10-12, 2020*, pages 194–203. ACM.
- Milojicic, D. S. (2011). Autograding in the Cloud: Interview with David O'Hallaron. *IEEE Internet Comput.*, 15(1):9–12.
- Nordquist, P. (2007). Providing accurate and timely feedback by automatically grading student programming labs. In Arabnia, H. R. and Clincy, V. A., editors, *Proceedings of the 2007 International Conference on Frontiers in Education: Computer Science & Computer Engineering, FECS 2007, June 25-28, 2007, Las Vegas, Nevada, USA*, pages 41–46. CSREA Press.
- Norouzi, N. and Hausen, R. (2018). Quantitative evaluation of student engagement in a large-scale introduction to programming course using a cloud-based automatic grading system. In *IEEE Frontiers in Education Conference, FIE 2018, San Jose, CA, USA, October 3-6, 2018*, pages 1–5. IEEE.
- Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Windowing: Local algorithms for document fingerprinting. In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San*

- Diego, California, USA, June 9-12, 2003, pages 76–85. ACM.
- Sharp, C., van Assema, J., Yu, B., Zidane, K., and Malan, D. J. (2020). An open-source, api-based framework for assessing the correctness of code in CS50. In Gianakos, M. N., Sindre, G., Luxton-Reilly, A., and Divitini, M., editors, *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*, pages 487–492. ACM.
- Sharrock, R., Bonfert-Taylor, P., Hiron, M., Blockelet, M., Miller, C., Goudzwaard, M., and Hamonic, E. (2019). Teaching C programming interactively at scale using taskgrader: an open-source autograder tool. In *Proceedings of the Sixth ACM Conference on Learning @ Scale, L@S 2019, Chicago, IL, USA, June 24-25, 2019*, pages 56:1–56:2. ACM.
- Silva, W., Alves, J., Brito, J. O., Bourguet, J., and de Oliveira, E. (2020a). An easy-to-read visual approach to deal with peer reviews and self-assessments in virtual learning environments. In *ICBDE '20: The 3rd International Conference on Big Data and Education, London, UK, April 1-3, 2020*, pages 73–79. ACM.
- Silva, W., Spalenza, M., Bourguet, J., and de Oliveira, E. (2020b). Lukewarm starts for computerized adaptive testing based on clustering and IRT. In Lane, H. C., Zvacek, S., and Uhomoibhi, J., editors, *Computer Supported Education - 12th International Conference, CSEDU 2020, Virtual Event, May 2-4, 2020, Revised Selected Papers*, volume 1473 of *Communications in Computer and Information Science*, pages 287–301. Springer.
- Silva, W., Spalenza, M., Bourguet, J., and de Oliveira, E. (2020c). Recommendation filtering à la carte for intelligent tutoring systems. In Boratto, L., Faralli, S., Marras, M., and Stilo, G., editors, *Bias and Social Aspects in Search and Recommendation - First International Workshop, BIAS 2020, Lisbon, Portugal, April 14, 2020, Proceedings*, volume 1245 of *Communications in Computer and Information Science*, pages 58–65. Springer.
- Silva, W., Spalenza, M., Bourguet, J., and de Oliveira, E. (2020d). Towards a tailored hybrid recommendation-based system for computerized adaptive testing through clustering and IRT. In Lane, H. C., Zvacek, S., and Uhomoibhi, J., editors, *Proceedings of the 12th International Conference on Computer Supported Education, CSEDU 2020, Prague, Czech Republic, May 2-4, 2020, Volume 1*, pages 260–268. SCITEPRESS.
- Sridhara, S., Hou, B., Lu, J., and DeNero, J. (2016). Fuzz testing projects in massive courses. In Haywood, J., Aleven, V., Kay, J., and Roll, I., editors, *Proceedings of the Third ACM Conference on Learning @ Scale, L@S 2016, Edinburgh, Scotland, UK, April 25 - 26, 2016*, pages 361–367. ACM.
- Ureel II, L. C. and Wallace, C. (2019). Automated critique of early programming antipatterns. In Hawthorne, E. K., Pérez-Quiñones, M. A., Heckman, S., and Zhang, J., editors, *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, pages 738–744. ACM.
- zu Eissen, S. M. and Stein, B. (2006). Intrinsic plagiarism detection. In Lalmas, M., MacFarlane, A., Rürger, S. M., Tombros, A., Tsirikika, T., and Yavlinsky, A., editors, *Advances in Information Retrieval, 28th European Conference on IR Research, ECIR 2006, London, UK, April 10-12, 2006, Proceedings*, volume 3936 of *Lecture Notes in Computer Science*, pages 565–569. Springer.