Fernandez-Gauna *et al.*
Int J Educ Technol High Educ (2023) 20:17
https://doi.org/10.1186/s41239-023-00386-6

International Journal of Educational Technology in Higher Education

RESEARCH ARTICLE Open Access

Automatic feedback and assessment of team-coding assignments in a DevOps context

Borja Fernandez-Gauna[1*], Naiara Rojo[2] and Manuel Graña[1]

*Correspondence:
borja.fernandez@ehu.eus

[1] Computational Intelligence Group, University of the Basque Country (UPV/EHU), Po Manuel Lardizabal, 1, 20018 Donostia-San Sebastian, Spain
[2] Faculty of Engineering, University of the Basque Country (UPV/EHU), Nieves Cano 12, 01006 Vitoria-Gasteiz, Spain

Abstract

We describe an automated assessment process for team-coding assignments based on DevOps best practices. This system and methodology includes the definition of Team Performance Metrics measuring properties of the software developed by each team, and their correct use of DevOps techniques. It tracks the progress on each of metric by each group. The methodology also defines Individual Performance Metrics to measure the impact of individual student contributions to increase in Team Performance Metrics. Periodically scheduled reports using these metrics provide students valuable feedback. This process also facilitates the process of assessing the assignments. Although this method is not intended to produce the final grade of each student, it provides very valuable information to the lecturers. We have used it as the main source of information for student and team assessment in one programming course. Additionally, we use other assessment methods to calculate the final grade: written conceptual tests to check their understanding of the development processes, and cross-evaluations. Qualitative evaluation of the students filling relevant questionnaires are very positive and encouraging.

Keywords: Team-coding, Automatic, Assessment, Assignments, DevOps

Introduction

Computer Science courses often include team-coding assignments during the learning and student evaluation process. To overcome these assignments, students need to complete (or write from scratch) code that satisfies a specification provided by the lecturers. It is usual in introductory courses that students work in pairs over a single version of the code, but in more advanced courses, it is often preferred that students work in groups to develop of soft skills such as teamwork, that are highly desirable by future employers (De Prada et al., 2022). In these scenarios, each team member works on his/her own personal computer, and makes changes to the team's shared code-base. These changes by different team members need to be integrated in a coordinated and orderly manner. Version Control Systems (VCS) offer functionalities for this purpose such as tracking changes, version control, code merging, and so on. Using a VCS in a coding assignment provides a learning outcome to the student that can be assessed by itself. Some coding

courses go one step further and involve the use of some of the automated processes used in DevOps for improved software quality. The current increasing interest on DevOps (Khan et al., 2022) makes the resulting learning outcome very interesting.

The manual assessment of coding assignments can be very time-consuming for lecturers. Thus, automated assessment methods are a current hot research track on computer science education methodologies (Cai & Tsai, 2019; Gaona et al., 2021; Gonzalez-Carrillo et al., 2021; Gordillo, 2019; Hegarty-E & Mooney, 2021). Most of the works found in the literature are based on software testing techniques (Strandberg et al., 2022), which provide measurable metrics to assess the correctness of the code written by the students (Gaona et al., 2021; Gonzalez-Carrillo et al., 2021; Hegarty-E & Mooney, 2021). Such metrics can be used to assess the performance of a team as a whole, but grading students by the overall performance of the team is ineffective for passive students, and can lead to tensions within the team and lack of motivation of the most brilliant students (Petkova et al., 2021). It is desirable to develop methods that assess individual contributions, so that the contribution of individual students can be traced for personal grading (Planas-Llado et al., 2021; Britton et al., 2017). Little progress has been reported in the literature toward this goal. We found only one metric based on the frequency of code edits (Hamer et al., 2021), but this metric is very rough and inaccurate, because wrong edits are counted positively.

The use of VCS in coding assignments offers opportunities to gather insightful information about the coding process that has been exploited in the development of automated student assessment methods (Hamer et al., 2021; Macak et al., 2021). To the best of our knowledge, no proposal has been made as to how to assess DevOps practices during a team-coding assignment.

In this paper, we deal with with two main research questions:

- RQ 1: Can the individual student contributions in team-coding assignments be automatically assessed? Accurate metrics would be desirable to measure how much of the performance of a team is due to each of the members. This would provide means to assess not only teams as a whole, but also to give better grades to students who contribute more.
- RQ 2: Can we assess DevOps practices in team-coding assignments in an automated manner? Modern software development involves the use of automated processes to assure software quality. It would be very interesting to have metrics that help us assess the use of these practices by the students in an automated way.

This paper is structured as follows. In "Background" we provide some background, and in "Related work" we review related works from the literature. "Methodology" presents the proposed automated assessment process and, finally, "Discussion and conclusions" presents our conclusions.

Background

Generally speaking, the process of developing software involves one or more developers modifying the source code files. These source code files are compiled to generate executable instances. Keeping track of the changes in the source files is considered an

absolute requirement in any serious software development project. Version Control Systems (VCS) provide mechanisms to store and restore snapshots of the code taken at different points in time. They store changes so that a developer can know who, when, and why code was changed. They also offer the possibility to restore a previous snapshot, or even create branches where developers can work over independent versions of the code without interfering with each other. Changes in the code are stored in a *repository* (database), which can be: (a) local (single repository hosted on the users machine), (b) centralized (a server hosts the repository), or (c) distributed (each user works on a local copy of the repository and synchronizes changes to the server when the user is satisfied with his/her contribution).

Git version control

Git is the most popular distributed VCS. Its success is partly due to the free hosting services offered by providers such *GitHub*[1]. *GitHub* alone claims to host over 200 million repositories where over 73 million developers work on software development projects.

In Git systems, different versions of the software are saved as a set of incremental changes with respect to the previous version at a specific time. Code changes are grouped and saved as *commits* whenever a developer wants to save the state of the project. A commit is a collection of changes stored in a set of files that has the following attributes:

- the author of the commit,
- a message describing the changes,
- a timestamp,
- the incremental changes in source files with respect to the previous commit (the parent),
- the Id of the commit, which is the result of using a hash function on the commit's attributes, and
- the Id of the parent(s) commit(s).

special commits (*merge commits*) merge two versions of the code (maintained in two separated *branches*) into a single version.

Commit messages are useful to describe the actions taken, and the goal of a specific set of changes. The common practice is to reference an issue using the syntax *#nmessage*, where $n \in \mathbb{N}$ is the id number of the referenced issue, and *message* is the actual description of the changes. An *issue* is a task within the software development task. This task can be a bug that needs to be fixed, a feature that needs to be added to the code, or any other related task. Issues (Perez-Verdejo et al., 2021) are often managed from outside the VCS, i.e., from the project management system offered by GitHub.

Branches in a repository consist on sequences of labels linked to a single commit. Although branches may have different interpretations, they are most commonly used to work simultaneously on different aspects of a project. The initial branch in a repository

[1] https://github.com.

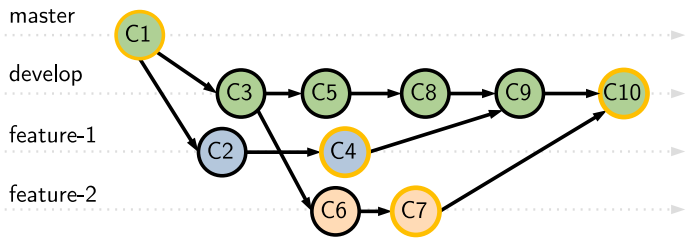

Fig. 1 Visual representation of a Git commit tree. Nodes in the graph represent commits and arrows represent parenthood between commits (arrows go from parent to children). The head commits of three branches (*develop*, *feature-1* and *feature-2*) are highlighted with an orange outline

is labeled *master*, and developers may create additional branches to work without changing the source code used by other developers. When a commit is added to a branch, the pointer is updated to this commit. Branches can be merged.

It is common practice to have a *develop* branch (Cortes Rios et al., 2022) shared by all the developers, and a different branch for each of the current lines of work. These can be related to a specific issue. Once the goal of a branch is accomplished, it is merged into the code-base shared with the rest of the developers (i.e. the *develop* branch). When branch a is merged into branch b, the changes done in branch a since the time both branches diverged, are added to branch b as a new commit. This new merge commit consists basically on a selection of the changes performed on each of the branches being merged.

Figure 1 shows a sample repository represented as a directed graph $G = (N, V)$, where nodes in N represent commits and vertices in V represent a parent-child relationship (the arrow points at the child). From the initial commit $C1$ on *develop*, a branch labeled *feature-1* was created, two commits (*C2* and *C4*) were added to this branch, and then merged to develop in commit *C9*, which, by the time both branches were merged had been added three new commits (*C3*, *C5* and *C8*). A third branch, labeled *feature-2*, was created from develop after commit *C3* and, similarly, merged to develop in commit *C10*. At any time, branches point to the latest commit added (*head commit*). The head commits of the three branches have an orange outline in the graph.

Distributed teamwork

In a Distributed Version Control System (DVCS) (i.e. Git), each developer has a local repository that will be synchronized with the shared repository using two operations: *pull* and *push*. The former integrates into the local repository any new changes found in the shared remote repository, whereas the latter integrates into the remote repository local changes that have not been pushed into the shared remote repository.

This distributed architecture allows for quite a lot of different workflows and integration strategies, but for simplicity, we will use *Gitflow* (Cortes Rios et al., 2022) in the remaining of this paper. Figure 2 illustrates this workflow showing four consecutive snapshots of a distributed repository with two developers contributing changes.

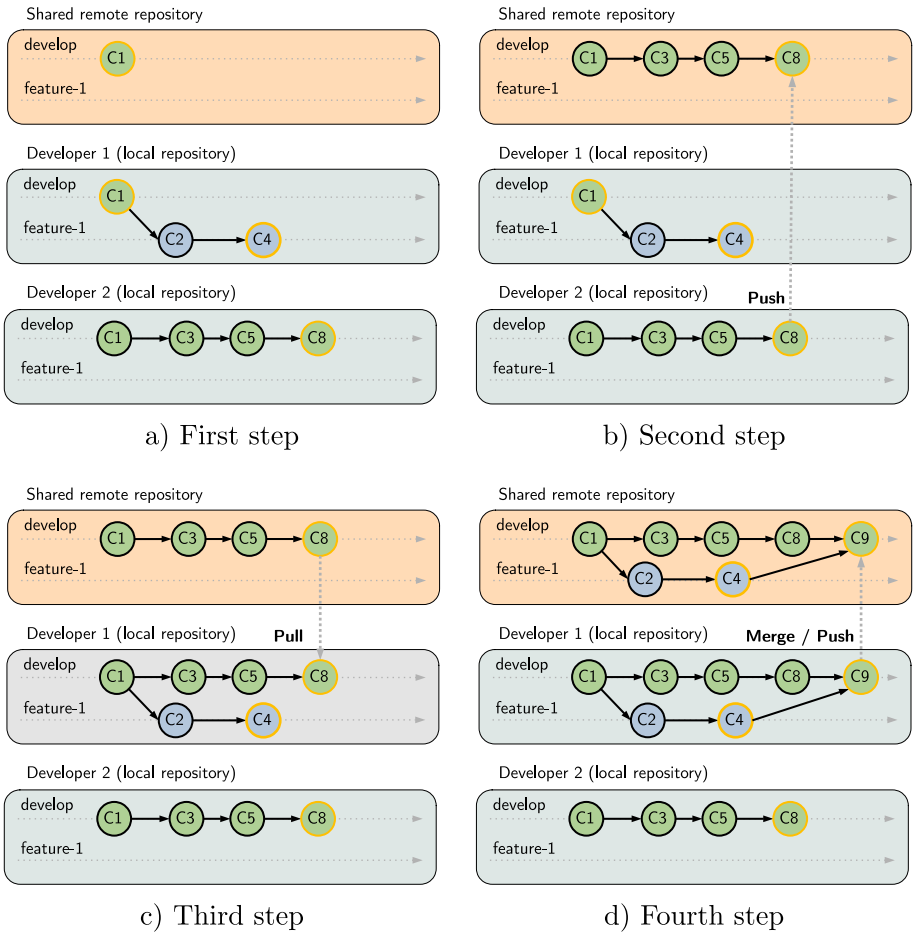

Fig. 2 An illustration of DVCS working in a four-step sequence of operations

- In the initial snapshot (Fig. 2a), only one commit has been added to the shared develop branch *(C1)*. Developers *1* and *2* each have a local repository. *Developer 1* has created a new branch labeled *feature-1* from *develop* and added two commits (*C2* and *C4*), while *Developer 2* has worked directly on the *develop* branch and has added *3* commits (*C3*, *C5* and *C8*).
- Figure 2b shows the next step in the sequence: no new commits were available on the remote repository and *Developer 2* has pushed its new commits to the shared repository.
- In the next snapshot shown in Fig. 2c, *Developer 1* has finished the work on *feature-1* and wants to integrate his/her changes into the shared branch *develop*. Before merging, the developer needs to integrate the new changes on the shared remote repository (*pull*).
- Finally, once the local repository is up to date, local changes in branch *feature-1* can be merged into the *develop* branch and pushed to the remote machine (Fig. 2d).

DevOps

DevOps is a culture within the software development community that aims to facilitate collaboration between two teams that traditionally have operated separately: software development and IT operations. Although there is no formal definition for this term (Almeida et al., 2022), this movement is generally considered to be complementary with Agile software development (i.e., extreme programming). The most salient feature of DevOps is the use of automation processes that simplify the software development workflow:

- *Continuous Integration* (CI): developers are encouraged to integrate often their local changes into the shared remote code-base (Holck and Jorgensen, 2012). These frequent integrations prevent merge conflicts, diverging code branches, and duplicated code. When changes are integrated in the shared remote code-base, executable instances are automatically built so that automated tests can be run in order that possible bugs are detected as soon as possible. The maturity of the CI tests is directly related to the quality of the software (Wang et al., 2022). If a version of the code fails to compile into an executable instance, we say that "it does not build".
- *Continuous Delivery* (CD): after a successful integration, additional automated steps are performed, including more testing, staging and deployment. The output of this process (if all steps are successful) is a release ready to be shipped to end-users.
- *Continuous Deployment* (CD): this automated process is related to the former, but focuses on making new releases available to end-users automatically and continuously. This allows for faster releases.

These automated processes are usually automatically triggered on a remote machine when changes are pushed to a shared branch in a repository. When one of these processes is triggered, an agent creates a virtual environment for each of the different target platforms. In these virtual environments, the processes listed above are executed. The design, construction and implementation of DevOps management systems is highly sophisticated, demanding professionals with a set of competencies that are mostly lacking in current computer engineering studies (Assyne et al., 2022).

Related work

Manually assessing assignments in coding courses is a time-consuming task that can become overwhelming with large groups of students and/or large-sized code. This has motivated recent scientific efforts with the aim to automatically assess coding assignments. The different approaches to automatically assess coding assignments can be clustered in two groups: static and dynamic analysis.

Static analysis methods do not need to actually build or run the artifacts (Jurado, 2021) and, instead, directly analyze the source code. Different applications of this type of analysis have been proposed: from assessing the design in Object Oriented Programming coding assignments (Le Minh, 2021; von Wangenheim et al., 2018), to establishing similarity between solutions so that they can be clustered for the purpose of sending and

shared feedback sent to the authors (Rubinstein et al. 2019; Clune et al., 2020). Static analysis can also be used to provide insight into interesting aspects of the quality model defined by the ISO 9126 (ISO, 2001) (i.e. maintainability). Metrics such as the Maintainability Index (Oman and Hagemeister, 1994; Coleman et al., 1994) or the SIG Maintainability Index (Heitlager et al., 2007) can be evaluated using static analysis.

On the other hand, dynamic analysis builds and runs the code and assesses the correctness of its behavior. This type of analysis is best suited for close-ended assignments where the input/output relationship is known in advance. Most usually, this assessment is based on running pre-defined (Gaona et al., 2021; Hegarty-E & Mooney, 2021; Gonzalez-Carrillo et al., 2021), or automatically generated automated software tests (Insa et al., 2021) against the solution provided by the students. A formal specification-based approach has also been proposed (Liu et al., 2019): the students' executable is executed capturing traces and comparing those with traces generated by a reference solution. This technique is only suitable for small coding assignments. Unit testing-based methods have been proposed for assessing graphical coding assignments (Wunsche et al., 2018), using pre-computed images as reference for comparison with images captured from the students' code. Assessing software correctness using tests assumes that the source code can be built and executed. An alternative approach that allows to grade non-building code (Parihar et al., 2017) is to use program reparing, automatically estimating how many repairs the code needs to build, and then using this estimation to deduct points from the grade. This might be preferable to a null grade in introductory courses.

VCS offer very interesting opportunities toward facilitating the evaluation of coding assignments. The most immediate advantage is that they allow to easily distribute and collect assignments (Hegarty-E & Mooney, 2021; Clifton et al., 2007). More interesting opportunities raise from exploiting the information stored in the commits. Specifically, commits are often considered as a unit to measure contributions to a repository (Hamer et al., 2021), and thus, measure inequalities in commits by the members of each team to help evaluate the individual contribution to the group effort. The most clear shortcoming of this approach is that not all commits contribute the same toward the project's goal, and it can be easily distorted by trivial changes. On the other hand, complexity metrics such as the *Contribution Complexity* (Hamer et al., 2021) can be used to assess how complex a commit is. This is based on the assumption that complex contributions may be more relevant for the finalization of the assignment than simpler ones. Some authors have focused their work on providing enhanced visualization of the sequence of commits in different branches. Visual analysis allows lecturers to quickly understand the branching process used in a repository (Youngtaek et al., 2021). Process mining techniques (Macak et al., 2021) have also been used to observe and understand the underlying committing process.

The literature shows an increasing interest in DevOps and the underlying automated processes (Khan et al., 2022; Saidani et al., 2022; Theunissen et al., 2022) but, to the best of our knowledge, no working proposals have been made as to how to assess the practices involved by DevOps processes in team-coding assignments. We have found only one article (Cai & Tsai, 2019) where the authors use DevOps processes for team-coding

assignments. In this work, the automatic grading system assesses the performance of a group by the percentage of automated tests passed by each team. This allows students to get immediate feedback, but the use of the DevOps processes by the students is not evaluated.

Methodology

In this section, we present our approach to assess team-coding assignments automatically. This method uses an automated process written in *C#* that periodically generates and updates reports assessing team and individual student progress. This software is distributed as an open-source project (GitRepoTracker) that can be downloaded[2] from GitHub.

Our approach offers two main advantages with respect to more traditional approaches:

- Students are provided feedback in a timely manner and they can plan their work better, and
- lecturers can effortlessly track the progress made by each team and each student, allowing them to better guide students toward the assignment goals.

In the following, we will present this method exemplified on a specific programming course, but the principles and techniques can be adapted and tailored to a broad variety of courses featuring team-coding assignments.

Academic context

We have applied the proposed methodology using an in-house built working DevOps system support for two years while teaching the course *Software Quality Control and Assurance*, which is an optional course in the fourth year of the *Computer Management and Information Systems Engineering* degree of the *University of the Basque Country (UPV/EHU)* in the *Faculty of Engineering of Vitoria-Gasteiz*. This subject is centered around a team-coding assignment, whose goal is to develop software that satisfies the specification provided by the lecturers while using DevOps practices during development, including the use of Continuous Integration and Continuous Delivery (CI/CD). Teams must write their own software tests (we will refer to these as *control tests*) along with the code in their own repository. The ultimate goal of the assignment is to pass several sets of automated tests written by the lecturers (*assessment tests*). Each set of tests has an associated date (*deadline*), and students must pass by this date a minimum percentage of the automated tests in the set.

The *Learning Outcomes* (LO) of this team project include the following:

- Working in an organized development team following directives set by the team leader (*LO1*).
- Defining and setting a quality assurance system guided by ISO 9126 (*LO2*).
- Developing software that satisfies the specification given in the assignment (*LO3*).

[2] https://github.com/borjafdezgauna/GitRepoTracker.

- Keeping track of changes in the source code using a distributed Version Control System (*LO4*).
- Using DevOps processes to improve the quality of the software (*LO5*).

In this course, students are graded using three different assessment tools. The final grade of a student is calculated as the weighted sum of these:

- The grade calculated by the automated process that we will describe below in this section (60% of the final grade),
- *individual conceptual tests* that evaluate individual understanding of the work done by the group and the processes involved in the teamwork (20%), and
- *cross-evaluations* (20%), so that students can evaluate the contribution of each member of the team (including themselves).

Assessment metrics

To assess the learning outcomes in an automated manner, we defined a set of quantitative assessment metrics that are aligned with the learning outcomes, and can be used to generate automatic feedback in order to help the lecturers to assess the assignments progress. We distinguish two types of metrics: *Team Performance Metrics* (TPM), and *Individual Performance Metrics* (IPM). While TPM are designed to assess the performance of the team as a whole, IPM measure individual performance as a the contribution made by an individual to the team's performance. The grade calculated by the automated process is calculated as the weighted sum of the student's group's TPM (60%), and his/her own IPMs (40%).

Team performance metrics

We will start defining Team Performance Metrics that allow to measure the performance of a group as a whole.

Build/test status of the shared branches (TPM1). Breaking the build and/or bringing back previously fixed errors to the shared code can lead to considerable amounts of wasted time. For this reason, not *breaking the build* is considered a primary goal in software development environments (Holck & Jorgensen, 2012). Thus, we consider this a requirement, and penalize teams with shared branches whose current version does not build and/or does not pass the control tests. This metric is used to assess the learning outcomes *LO4* and *LO5*.

Percentage of time that shared branches were in a valid status (TPM2). Whereas *TPM1* only checks the current status of shared branches, *TPM2* checks the status of every commit pushed to the shared branches. Because each commit has its date/time, it can be easily calculated how much time each shared branch has been in a valid state. The use of this metric encourages students to keep a valid status at all times by checking that the source code can be built and pass control tests before pushing it to the shared branches. As *TPM1*, this metric is used to assess *LO4* and *LO5*.

Code coverage of the control tests (TPM3). Code coverage measures the fraction of the code being run by a set of tests. Tests with a higher code coverage value are more exhaustive than tests with lower values. This metric is widely used by all kinds of software development teams, but has been neglected by previous published works. It is aligned with learning outcome *LO2*, and it can only be measured if the code can be built. To calculate the coverage, we use a tool called Coverlet[3]

Percentage of assessment tests passed (TPM4). By means of assessing the degree of conformity to the specification, we test each team's code against our assessment tests. This measure is clearly aligned with learning outcome *LO3*, but also with *LO5* because automated testing lies at the core of DevOps processes. This metric can only be measured if the code can be built into a running executable. Any programming project usually involves tasks of different orders of complexity. In order to make this metric fair, students who solve harder tasks should be given a higher mark than students solving easier ones. Our approach to do this consists on assessing the complexity of the tasks manually when we design the assignment, and then, writing the number of assessment tests related to each of the tasks proportional to their complexity. The more complex a task, the more assessment tests we write related to this task. This way, students that solve more complex tasks obtain a higher reward than students solving more simple tasks.

Adherence to the coding style rules (TPM5). In software development teams, it is usual to define and enforce basic common coding style rules that may include comments, indentations, white spaces, or naming conventions. Using a uniform coding style improves code readability, and reduces the time to understand code written by a different developer. In our team-coding assignments, we provide the students some coding style rules at the beginning. We use regular expressions to detect broken rules in a team's source code, and calculate the score as a function of the times these rules are broken. This metric is aligned with *LO1*.

Individual performance metrics

Next, we will define the Individual Performance Metrics that measure the contribution of each individual to the performance of the group.

Bonus to team leaders (IPM1). Each team chooses a leader at the beginning of the assignment and, we bonus their extra work coordinating the group.

Percentage of commit messages that satisfy some objective criteria (IPM2). Commits store messages describing changes done in the source and references to issues related to these changes. This allows developers to perform string searches for specific changes. We set a minimum number of characters and look for issue references using regular expressions and determine what percentage of the commit messages written by each student meets the criteria. This measure is part of the assessment of the learning outcome *LO4*.

[3] https://github.com/coverlet-coverage, that calculates two main statistics: the rate of code lines ($r_l : [0, 1]$), and the rate of branches covered by the tests ($r_b : [0, 1]$). We use a heuristic calculation to capture these two statistics as a single numerical value: $c = (r_l + r_b) * 50$. This calculation gives a metric in range [0, 100] that works well for our purposes.

Contribution to the code coverage of the control tests (IPM3). As part of a student's individual performance, we calculate the contribution of the student to improve the team's code coverage (*TPM3*). Roughly, this is the percentage of the team's code coverage (measured the same way as *TPM3*) that was due to a particular student's changes in code. "Incremental metric calculation" explains how this metric is calculated incrementally. Naturally, this metric needs code to built into a running executable, and it is related to *LO2* (as *TPM3*).

Contribution to pass control tests (IPM4). Conceptually akin to *IPM*2, this metric calculates the percentage of the team's control tests passed that is due to the student's changes in code. It also requires code to be built and it is calculated using the method in "Incremental metric calculation", and this metric is linked with *LO4* and *LO5*.

Contribution to pass assessment tests (IPM5). Also similar to *IPM*3, this metric calculates the percentage of the assessment tests passed (*TPM4*) due to the student's changes in code. The calculation also requires buildable code and is calculated with the method described in "Incremental metric calculation", and this metric is linked with *LO3* and *LO5*.

Percentage of commits pushed to the shared branch that build and pass control tests (IPM6). This metric is designed to measure the individual contribution to *TPM1*: being the team's goal to keep shared branches in a valid state, we calculate the percentage of commits pushed by each student to the shared repository branches that build and pass control tests. Git repositories do not store information about which commits are pushed to a remote server. We used the GitHub API to gather this information from GitHub push events. This metric is used to assess learning outcomes *LO4* and *LO5*.

Contribution to adherence to style rules (IPM7). In a similar fashion to how *IPM3* and *IPM4* are calculated, we calculate the score indicating the degree of adherence to coding style rules for each commit. Then, we use the same procedure described in "Incremental metric calculation" to attribute each student their due contribution to the team's score. In our implementation, we use regular expressions to calculate the adherence score, meaning this metric can be measured on non-building code too. This metric is related to *LO1*.

Commit regularity (IPM8). One of the issues in team-coding assignments is the lack of a timely effort by all members of a team. Most students tend to work harder near deadlines (Hamer et al., 2021). This becomes a problem in team assignments, because students find themselves unable to be helpful to the team after a period of less involvement on the project. This metric divides the duration of the assignment in periods and calculates the percentage of periods the author has committed changes. The length of these periods can be configured depending on the particular assignment or preferences. We use 3.5 day periods because our course has two sessions every week. Students are not forced to commit the same days these sessions occur, but they can also commit changes from home some days before and after the sessions. While this metric is arguably insufficient to determine the amount of effort or improvement contributed, it encourages students to be constant, which in turn, improves the performance of the teams and also helps students progressively acquire the learning outcomes. This metric assesses *LO1*.

Other metrics. We considered and tried some other metrics but decided not to use them:

- Contribution to the team performance could be approximated by counting the number of commits done by each student (Hamer et al., 2021). In our opinion, it is not a good metric because it doesn't quantify the actual contribution. Some students may add significant contributions in a single commit, while other may do a lot of commits with relatively irrelevant changes. This assessment metric can be easily distorted by commits of innocuous changes.
- An alternative approach would be to calculate which percent of the final code has been written by each student. This metric can be calculated from the output of the command *git blame*, that establishes who wrote each of the line of codes in each file. We consider this metric a good measurement of individual contribution to team performance, but it is a poor indicator of the contribution in the sense that some students may solve the same problem with much more code than others. The contribution would be the same, but those writing longer code would be attributed more credit than the rest.

Incremental metric calculation

The proposed system and methodology for automated assessment builds all the snapshots of the source code, calculating the assessment metrics for each of those snapshots. This allows us to distribute credit among the team members calculating their contribution to some of the metrics proposed in the previous subsection. We will illustrate this process and comment on the decisions we took in our implementation.

Figure 3 illustrates the process on a trivial case: a sequence of three commits where all three consecutive versions of the code can be built. The first commit (*C1*) obtained a score of 40% for some of the metrics, and the following commit (*C2*) by *Student 2*, obtained a 60% score. The last commit (*C3* by *Student 1*) lowered the measure to 55%. In this case, contributions (or attributed changes to the measures) to the measured metric can be calculated with respect to the parent commit:

- *Student 2* is attributed a +20% change for commit *C2* (60−40%), and
- *Student 1* is attributed a −5% change for authoring commit C3 (55−60%).

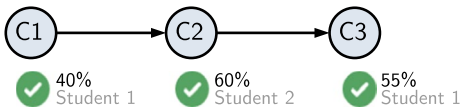

Fig. 3 Sample sequence of commits by two different authors showing a measure expressed as a percentage value. Green check symbols indicate that all three versions of the code can be built

Next, let us consider branch merges. These can be done either by using *merge* or *rebase* Git commands for merging branches. Figure 5 shows a sequence of commits, where commit *C3* (with a 60% measure) merges commits *C1* (40%) and *C1* (50%). The obvious question here is: *what metric change do we attribute to the author of the merge commit?*. Merging commits is an operation that basically consists on selecting changes from the two versions of the code being merged. No new code as such is created in merge commits. This operation is so trivial that sometimes Git is capable of merging branches automatically without human interaction from the developer when there are no conflicting changes. There is no reliable way to predict how merging will affect metrics in the branches merged. For example, if compliance with new tests had been added in both of the merged branches, the merged code could have a higher code coverage than any of the predecessors. With this in mind, an option would be to disregard any changes in metrics between merge commits and their predecessors. The downside of this choice becomes apparent in another example: consider a student manually merging two branches and wrongly choosing not to integrate some of the changes. If this merge is fixed in the following commit, the student may be attributed raises in some of the metrics that were originally fixed in a different commit (potentially by a different author) and, due to *resetting* metrics in merge commits, the student may gain artificially generated credit. This could be exploited by students to increase their grade. Nevertheless, we consider this potential risk worth the benefits, and we decided to ignore metric changes in merge commits.

Another interesting situation raises if we consider measures that can only be calculated on code that builds correctly (i.e., *IPM3*). Consider the sequence of commits in Fig. 4. The second commit (*C2* by *Student 2*) does not build. The trivial solution would be to assign the measured value in commit *C2* to 0%. This would attribute a +40%

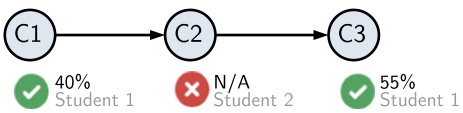

Fig. 4 Sample sequence of commits by two different authors showing a measure expressed as a percentage value. The code in the first and the last commits can be built (green check symbols), but the second cannot (red cross symbol). The measure in this sample can only be taken from building code, and thus, the second commits shows a *Not available* (N/A) label

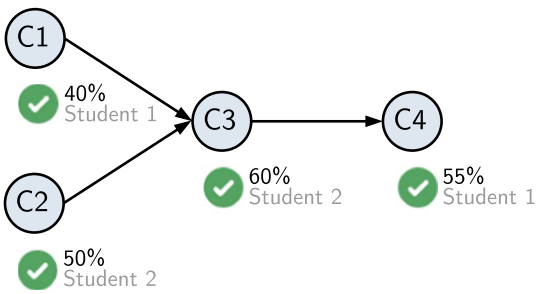

Fig. 5 Sample sequence of commits by two different authors showing a measure expressed as a percentage value. Commit *C3* merges commits *C1* and *C2*. All four commits build

change to *Student 2* (0−40%) and a −55% change to *Student 1* (55−0%). We performed some tests with real assignments and found out that doing this led to disproportionate differences between students, because some students repeatedly fixed errors introduced by others. Another option would be, for those metrics that required building code to be calculated, to compare not with the immediate predecessor, but with the first one that can be built (going backward in history). Again, this technique has a downside too. We will illustrate it with an example based on Fig. 4: in *C2*, *Student 2* may have increased the coverage of the tests but accidentally introduced an error that prevented building the source code. *Student 1* may have fixed then this error in *C3*, and thus, would attributed the +15% metric change (in this example, code coverage) that was actually done by *Student 2*. We weighted both options and decided to use the second one: compare with the first building predecessor. We believe that letting students know about this decision actually encourages them to double-check that they only push building code that passes control tests.

Another variation of this situation is shown in Fig. 6: a merge commit (*C3*) breaks the build. *What predecessor do we consider in order to calculate metric changes due to C4?* There is no clear way to select one, so we decided to ignore changes in metrics requiring building code for commits with no building predecessor before the first merge commit. This decision has a downside: any metric change introduced in *C4* will

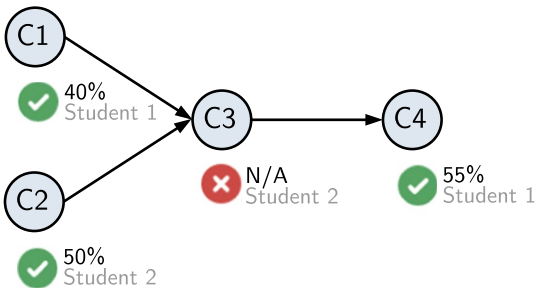

Fig. 6 Sample sequence of commits by two different authors showing a measure expressed as a percentage value. Commit *C3* merges commits *C1* and *C2*. The author made some mistake in the merge operation and this caused commit *C3* to not build

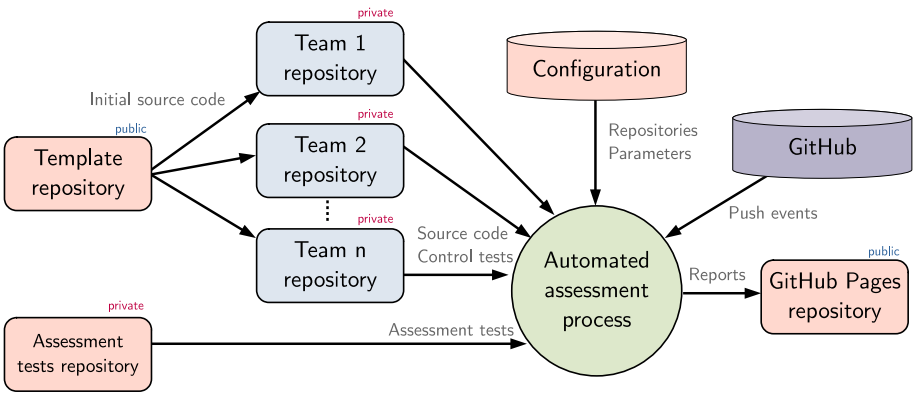

Fig. 7 Scheme of the proposed automated assessment process

not be attributed to the author. Our tests have shown that this potential error is negligible if students are encouraged to commit changes frequently enough.

Automated assessment process

The assignment assessing automated process we propose is based on automated processes similar to those used in DevOps. In our coding courses, we schedule this process to run every hour, but it could also be triggered automatically each time changes are pushed to a team's repository. The architecture of this process is graphically represented in Fig. 7. Projects are stored in Git repositories hosted in GitHub, and the assessment process also uses the GitHub API to query information that is not stored in the repository itself (i.e. which commits have been pushed to the server). Initially, each team creates a new private repository using the public template provided by the lecturer. The members of the team and the lecturers are the only users allowed to access these private repositories to discourage plagiarism. Teams work on their own repository, that will include their solution to the assignment, and their own tests (control tests). In order to run the assessment tests on the code developed by a team, it is necessary to define the interfaces (classes and methods) that will be used from the testing code. These cannot be changed during the duration of the assignment.

Every time the automated assessment process is triggered, changes from each of the team repositories in the configuration file are pulled. The automated assessment process performs incremental updates, so that only changes committed from last update need to be processed. Using the configuration parameters, Team Performance Reports (TPR) and Individual Performance Reports (IPR) are generated. These reports are uploaded to the GitHub Pages service, which allows to publish web pages using the same VCS that code projects do. Figure 7 shows a graphical representation of how the components of this architecture interact. The inputs of the automated process are: a) the repositories with the code written by each team, b) the repository with the assessment tests, c) the configuration file with the parameters of the process, d) the information returned by the GitHub API queries. The output is HTML code that is periodically uploaded to the GitHub Pages repository (technically, changes are pushed because it is managed as a repository).

Algorithm The team-coding assignments of n teams of m members each are processed and assessed in a two-step procedure:

1. First, changes committed to the repositories are processed, generating a set of statistics ($stats_i$, where $i = 1, n$ is the index of the team) that contains the following information for every commit: author, id, date, parent(s) commit(s), build result, control test result, code coverage result, assessment test result, and code analysis result. This first step is described by Algorithm 1.
2. Then, using as input the statistics from the previous step ($stats_i$), Team Performance Metrics are calculated and the incremental metric calculation described in "Incremental metric calculation" is used to calculate Individual Performance Metrics. With

these measures and the assessment parameters set in the configuration file, teams and students are evaluated, generating the Team Performance Reports (TPR_i) and Individual Performance Reports ($IPR_{i,j}$, where $j = 1, m$ is the index of the team member). These are made accessible to the students using the GitHub Pages service. This process is described by Algorithm 2.

For the sake of reproducibility, we describe here the most important commands used in used in the algorithms:

- *ParseGitLog*: We used the command

  $$git\ log\ [branch]\ --reverse\ --parents$$

  This command outputs all the commits in the given branch from older to newer, showing the parent(s) commit(s) of each commit, its Id, author, date and message.
- *RunControlTests*: We used the command

  $$dotnet\ test\ --collect:"XPlatCodeCoverage"\ -v\ normal$$

  to run the control tests from within a team repository's folder. The *collect* flag is used to collect code coverage information.
- *RunAssessmentTests*: In order to run our assessment tests with the code developed by the students, we first add to the assessment test project a reference to the team's project. Then, we build and run the assessment tests with the command

  $$dotnet\ test\ -v\ normal$$

  (there is no reason to measure the code coverage of the assessment tests). Finally, we removed the reference to their code restoring the previous version of our assessment test code using *git restore* .

Algorithm 1 Scheme of the proposed automated assessment process.

Require: $configuration$
Ensure: $stats_i\ \ i = 1, n\ \ j = 1, m$
clone/pull $configuration.Assessment\_Test\_Repository$
for all $repo_i\ \ in\ \ configuration.Repositories$ **do**
  $stats_i \leftarrow LoadOrInitialize(i)$
  $clone/pull\ \ repo$
  $new\_commits \leftarrow ParseGitLog()$
  $pushed\_commits \leftarrow GithubAPI.PushEvents()$
  for all $commit \in new\_commits$ **do**
    $CheckoutSharedBranch(repo_i)$
    $builds \leftarrow Build(repo_i)$
    $controlTestsResults \leftarrow RunControlTests(repo_i)$
    $assessTestsResults \leftarrow RunAssessmentTests(repo_i)$
    $stats_i.Commits \Leftarrow commit$
    $stats_i.CommitStats \Leftarrow commit, builds, controlTestsResults, assessTestsResults$
  end for
  $Writestats_i$
end for

Algorithm 2 Repository analysis algorithm.

Require: $configuration, stats_i \quad i = 1, n$
Ensure: $TPR_i, IPR_{i,j} \quad i = 1, n \quad j = 1, m$
for $i \quad i = 1, n$ **do**
  Read $stats_i$
  ▷ TPM1: Build/test status of the shared branches
  $TPR_i \Leftarrow TPM1(configuration, stats_i.Commits.Last)$
  ▷ TPM2: Percentage of time that shared branches were in a valid status
  $TPR_i \Leftarrow TPM2(configuration, stats_i.Commits.Pushed)$
  ▷ TPM3: Code coverage of the control tests
  $TPR_i \Leftarrow TPM3(configuration, stats_i.Commits.Last)$
  ▷ TPM4: Percentage of assessment tests passed
  $TPR_i \Leftarrow TPM4(configuration, stats_i.Commits.Last)$
  ▷ TPM5: Adherence to the coding style rules
  $TPR_i \Leftarrow TPM5(configuration, stats_i.Commits.Last)$
  $GeneratePlots(TPR_i, stats_i)$
  Write TPR_i
  Commit and push TPR_i
  for all $member \in configuration.Members$ **do**
    ▷ IPM1: Bonus to team leaders
    $TPR_{i,j} \Leftarrow IPM1(configuration, member)$
    $memberCommits \leftarrow CommitsAuthoredBy(member, stats_i.Commits)$
    ▷ IPM2: Percentage of commit messages that satisfy some objective criteria
    $IPR_{i,j} \Leftarrow IPM2(configuration, memberCommits)$
    ▷ IPM6: Percentage of commits pushed to the shared branch that build and pass control tests
    $IPR_{i,j} \Leftarrow IPM6(configuration, memberCommits, stats_i.Commits)$
    ▷ IPM8: Commit regularity
    $IPR_{i,j} \Leftarrow IPM8(configuration, memberCommits)$
    for all $commit \in memberCommits$ **do**
      if $IsNotMergeCommit(commit)$ **then**
        $predecessor \leftarrow BuildingPredecessor(commit)$
        ▷ IPM3: Contribution to the code coverage of the control tests
        $IPR_{i,j} \Leftarrow IPM3(configuration, commit, predecessor)$
        ▷ IPM4: Contribution to pass control tests
        $IPR_{i,j} \Leftarrow IPM4(configuration, commit, predecessor)$
        ▷ IPM5: Contribution to pass assessment tests
        $IPR_{i,j} \Leftarrow IPM5(configuration, commit, predecessor)$
        $predecessor \leftarrow ImmediatePredecessor(commit)$
        ▷ IPM7: Contribution to adherence to style rules
        $IPR_{i,j} \Leftarrow IPM7(configuration, commit, predecessor)$
      end if
    end for
    Write $IPR_{i,j}$
    Commit and push $IPR_{i,j}$
  end for
end for

Reports Students receive feedback in the form of reports published via GitHub Pages that include their Team Performance Report and their own Individual Performance Report. The assessments in these reports are based on the metrics defined in "Assessment metrics".

A sample Team Performance Report is shown in Fig. 10. Each of the evaluation items corresponds to one of the performance metrics, and the values measured are mapped to a score using the parameters from the configuration file (minimum/maximum value, weight, and so on). For some of the items in the report, additional information is shown to help the students understand the score. For example, for each set of assessment control tests, students can view the name of the failed tests. This is provided as a hint to help them identify what the possible source of the errors are, but we would not recommend showing the actual error message. That would miss the point, which is that their tests should be exhaustive enough to guarantee quality.

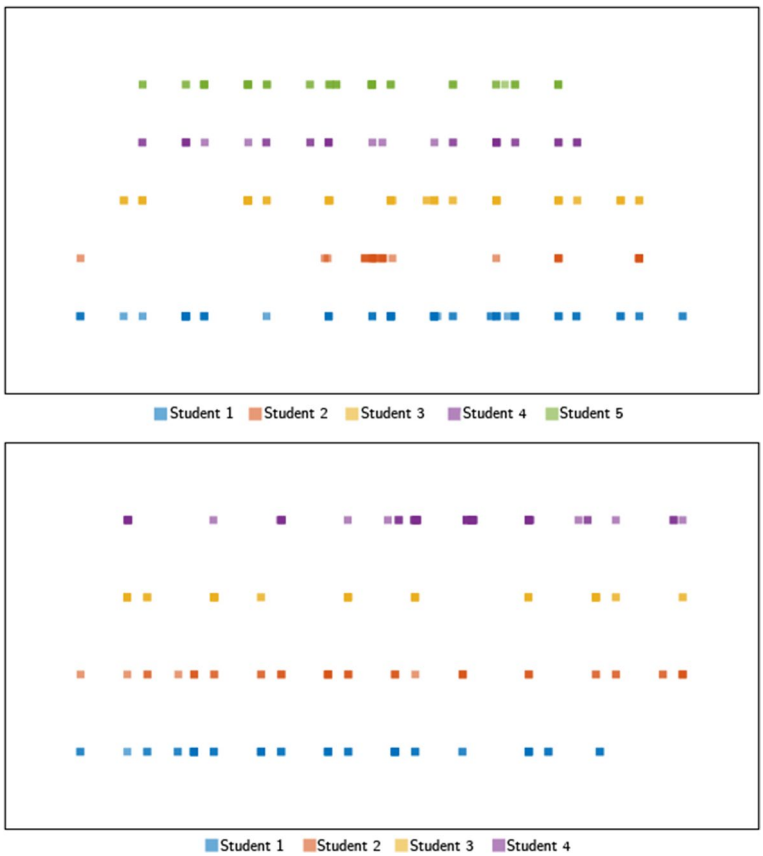

Fig. 8 Two samples of the heat-maps used to represent the temporal distribution of the commits done by each participant in a real team-coding assignment. The names of the students have been anonymized for personal data privacy reasons

Figure 11, on the other hand, shows two sample Individual Performance Reports. The format is very similar to the Team Performance Report, and the additional information includes in which commits they made which changes to the metrics measured.

The reports also include two plots that help them visualize their effort and progress over time: (a) heat-map showing the distribution of each team member's commits, and (b) the assessment test score obtained in each commit over time.

Two samples of the former are shown in Fig. 8. The heat-maps show the distribution of the commits done by each team member over time. Commits by each participant are colored using the same color, and they are positioned on the same vertical position along the y axis. The x axis represents time, where the left border of the image is the start of the project and the y axis is the current date.

On the other hand, Fig. 9 shows two samples of the latter. These show the percentage of passed tests for each set of assessment tests defined in the assignment over time: the green line represents the percentage of tests passed in a first set of assessment tests (labeled *Parser*), and the orange line represents the results in a second set of assessment tests (*Queries*). Each set of assessment tests is configured with a deadline date (*deadline*) and a minimum score (*min*), and these are represented with a vertical line from 0% to *min*% at $y = deadline$. In the example, the deadline of the *Parser*

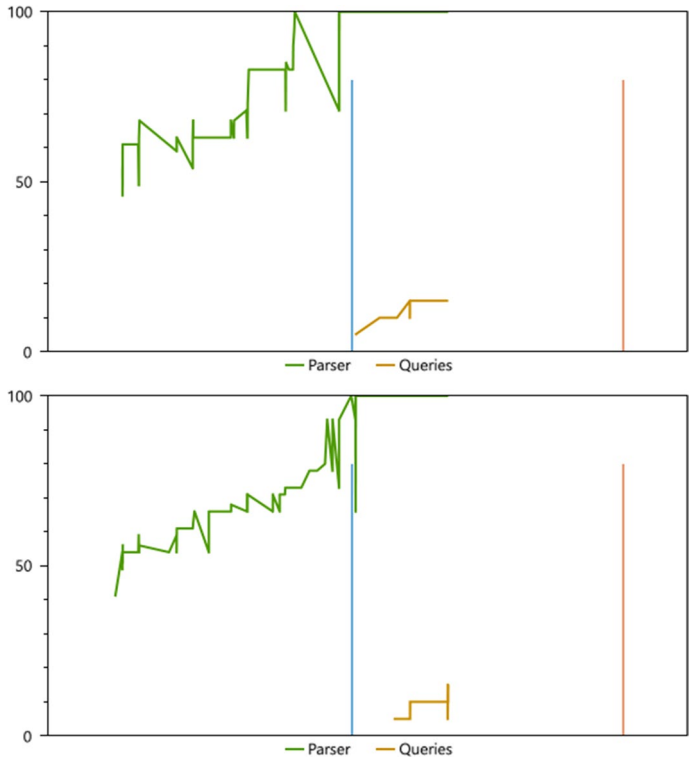

Fig. 9 Two sample plots showing the percentage of the assessment tests passed by each team in a real team-coding assignment. In this project, two deadlines have been defined: the first one (labeled *Parser*) in green, and the second one (labeled *Queries*) in orange

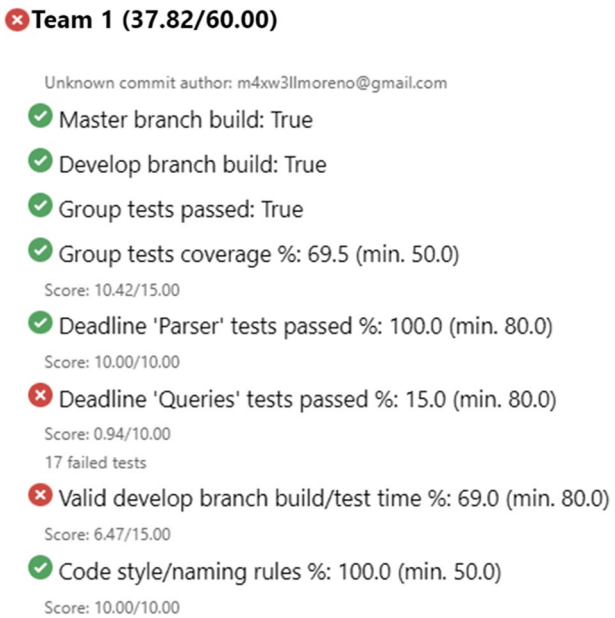

Fig. 10 Sample team report from a real team-coding assignment. The team name has been anonymized for data privacy reasons

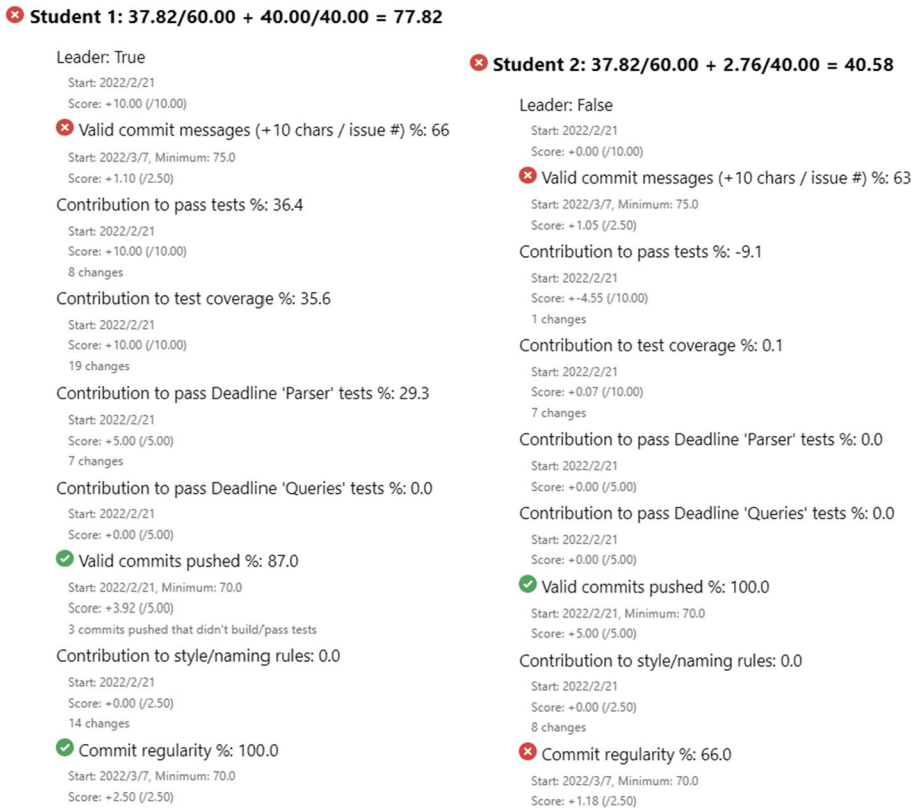

Fig. 11 Two sample individual reports from a real team-coding assignment. The student names have been anonymized for data privacy reasons

set of tests is represented as a blue line, whereas the deadline of the *Queries* set of tests is represented as a red line. The team goal is to make the green (or orange) line go over the vertical blue (red) line, which is a quite intuitive way to express progress with respect to a goal.

Qualitative results: student satisfaction and grades

The students enrolled in the 2021/22 course were asked to assess the degree of agreement as a numerical value from 1 to 5 to four different affirmative sentences regarding their opinion on the automated assessment method:

1. I was motivated by the automated assessment method to actively participate on the assignment.
2. My team was motivated by the automated assessment method to collaborate on the assignment.
3. The automated assessment method helped me decide how to orient my work and improve my grade.
4. I prefer automated assessment methods with continuous feedback than manual assessments with eventual feedback.

The average answer to *Sentence 1* was 4.1 (all answers were in range [3, 5]). This indicates that students felt positively motivated as individuals to contribute to the team effort. Students gave a higher score to *Sentence 2* by 4.3, which indicates they perceive a positive effect on the work of the team as a result of the assessment method. Regarding *Sentence 3*, the score was a bit lower (3.6). Our future work includes providing even more information in the reports so they can better identify ways to improve their performance. Finally, the agreement to *Sentence 4* achieved an score of 4.1. These results are encouraging and we conclude that students are satisfied with this assessment system and methodology.

The average grade obtained by the students (all grades are expressed out of 10) of the 2021/22 course was 7.51($\pm$2.92). This is a substantial improvement with respect to the scores given by manually assessing the assignments of the students of the two previous courses: 7.02($\pm$3.31) in 2019/20 and 6.93($\pm$3.18) in 2020/21. In all three courses, the same team of lecturers supervised the students and assessed the assignments.

We also conducted an internal manual grading of the 2021/22 assignments for comparison purposes. Our manual grading, using the same criteria we used in previous courses, produced an average grade was 7.37($\pm$3.23). This is 0.14 lower than the average grade calculated by the automatic process. This difference is almost negligible and may be in any case corrected by adjusting the weights and parameters of the assessment process. When compared to previous courses, the manual grading of this course's assignments gave considerably higher grades. Our hypothesis is that the automatic feedback system challenges the students and motivates them to work on the assignment.

Discussion and conclusions

Regarding the research questions in "Introduction", the incremental metric calculation method presented in "Incremental metric calculation" provides an answer to RQ1. We can distribute credit between team members by measuring performance metrics in each of the snapshot in the code repository and then, attributing a commit's author the differences in metrics with respect to the previous snapshot. This method is easily understood by the students and, although it is not completely accurate, it provides a way to organize work between members so that everyone gets credit for what they do.

"Assessment metrics" presents some novel performance metrics that aim to answer the second research question, RQ2, namely, Team Performance Metrics *TPM1*, *TPM2* and *TPM4*, and Individual Performance Metrics *IPM3*, *IPM4*, and *IPM5*. These metrics measure adherence to some of the practices involved in DevOps. By no means do they cover all the aspects of DevOps processes, but they offer an adequate start point for team-coding courses. They can be further extended and tailored to other courses with specific needs.

An important component of the assessment method is the use of assessment tests, hidden to the teams, that measure how many of the goals teams have reached. In order to run these tests, common interfaces need to be defined in the initial code handed to the students. Our approach requires that these interfaces are kept unchanged throughout the assignment. We also note here that special care must be taken when these tests are designed, because the assessment is based on the number of tests passed. This means that, for example, tests that use a lot of different functionalities will not pass until each

of those functionalities are correctly implemented. This makes the task of distributing credit between students harder, and the credit may be distributed in an unfair way to the student completing the last required functionality. We believe it is best to write small tests for specific functionalities, so that changes by a single member can lead to direct rewards that encourage to work further.

Conclusions

We have presented an automated assessment process for team-coding assignments. This method defines some Team Performance Metrics to measure properties of the software developed by each team, and also the correct use of DevOps techniques. It tracks the progress on each of the metrics by each group, and it also defines Individual Performance Metrics to distributes credit among team members of any change in Team Performance Metrics. These metrics are used in a periodically scheduled process to generate reports that provide students valuable feedback. This process also facilitates the process of assessing the assignments. Although this method is not intended to produce the final grade of each student, it provides very valuable information to the lecturers and we have used it as the main source of assessment in one subject. Additionally, we use other assessment methods to calculate the final grade: written conceptual tests to check their understanding of the development processes, and cross-evaluations.

Our future work will include using automatized code complexity metrics. This will allow us to measure other aspects we do not yet consider (i.e, readability of the code or cyclomatic complexity). We expect that using such metrics will encourage students not only to complete their tasks, but also to improve already working code.

Acknowledgements
Not applicable.

Author contributions
All authors read and approved the final manuscript.

Funding
Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Availability of data and materials
The software written by the authors as part of this work is publicly available in https://github.com/borjafdezgauna/GitRepoTracker. On the other hand, the code written by the students and their grades are not available due to data privacy reasons.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 13 December 2022 Accepted: 22 February 2023
Published online: 24 March 2023

References

Almeida, F., Simoes, J., & Lopes, S. (2022). Exploring the benefits of combining devops and agile. *Future Internet, 14*(2), 63. https://doi.org/10.3390/fi14020063

Assyne, N., Ghanbari, H., & Pulkkinen, M. (2022). The state of research on software engineering competencies: a systematic mapping study. *Journal of Systems and Software, 185*, 111183. https://doi.org/10.1016/j.jss.2021.111183

Britton, E., Simper, N., Leger, A., & Stephenson, J. (2017). Assessing teamwork in undergraduate education: a measurement tool to evaluate individual teamwork skills. *Assessment & Evaluation in Higher Education, 42*(3), 378–397. https://doi.org/10.1186/s41155-022-00207-1

Cai, Y. & Tsai, M. (2019). Improving programming education quality with automatic grading system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* 11937 LNCS:207–215. https://doi.org/10.1007/978-3-030-35343-8_22.

Clifton, C., Kaczmarczyk, L., & Mrozek, M. (2007). Subverting the fundamentals sequence: Using version control to enhance course management. *SIGCSE Bull, 39*(1), 86–90. https://doi.org/10.1145/1227504.1227344

Clune, J., Ramamurthy, V., Martins, R., & Acar, U. (2020). Program equivalence for assisted grading of functional programs. *Proceedings of the ACM on Programming Languages, 4*(OOPSLA). https://doi.org/10.1145/3428239.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *IEEE Computer, 27*(8), 44–49. https://doi.org/10.1109/2.303623

Cortes Rios, J., Embury, S., & Eraslan, S. (2022). A unifying framework for the systematic analysis of git workflows. *Information and Software Technology, 145*, 106811. https://doi.org/10.1016/j.infsof.2021.106811

De Prada, E., Mareque, M., & Pino-Juste, M. (2022). Teamwork skills in higher education: is university training contributing to their mastery? *Psicologia: Reflexao e Critica, 35*(5). https://doi.org/10.1016/j.ijme.2021.100538.

Gaona, E., Perez, C., Castro, W., Morales Castro, J. C., Sanchez Rodriguez, A., & Avila-Garcia, M. (2021). Automatic grading of programming assignments in moodle. pp. 161–167. https://doi.org/10.1109/CONISOFT52520.2021.00031.

Gonzalez-Carrillo, C., Calle-Restrepo, F., Ramirez-Echeverry, J., & Gonzalez, F. (2021). Automatic grading tool for jupyter notebooks in artificial intelligence courses. *Sustainability (Switzerland), 13*(21), 12050. https://doi.org/10.3390/su132112050

Gordillo, A. (2019). Effect of an instructor-centered tool for automatic assessment of programming assignments on students' perceptions and performance. *Sustainability (Switzerland), 11*(20), 5568. https://doi.org/10.3390/su11205568

Hamer, S., Lopez-Quesada, C., Martinez, A., & Jenkins, M. (2021). Using git metrics to measure students' and teams' code contributions in software development projects. *CLEI Eletronic Journal (CLEIej), 24*(2). https://doi.org/10.19153/cleiej.24.2.8.

Hegarty-E, K. & Mooney, D. (2021). Analysis of an automatic grading system within first year computer science programming modules. pp. 17–20. https://doi.org/10.1145/3437914.3437973.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pp. 30–39. https://doi.org/10.1109/QUATIC.2007.8.

Holck, J. & Jorgensen, N. (2012). Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, 40–53. https://doi.org/10.3127/ajis.v11i1.145.

Insa, D., Perez, S., Silva, J., & Tamarit, S. (2021). Semiautomatic generation and assessment of java exercises in engineering education. *Computer Applications in Engineering Education, 29*(5), 1034–1050. https://doi.org/10.1002/cae.22356

ISO, IEC,. (2001). *ISO/IEC 9126*. Software engineering—Product quality: ISO/IEC. https://doi.org/10.1016/j.ijme.2021.100538

Jurado, F. (2021). *Teacher assistance with static code analysis in programming practicals and project assignments.*https://doi.org/10.1109/SIIE53363.2021.9583635

Khan, M., Khan, A., Khan, F., Khan, M., & Whangbo, T. (2022). Critical challenges to adopt devops culture in software organizations: A systematic review. *IEEE Access, 10*, 14339–14349. https://doi.org/10.1109/ACCESS.2022.3145970

Le Minh, D. (2021). Model-based automatic grading of object-oriented programming assignments. *Computer Applications in Engineering Education*. https://doi.org/10.1002/cae.22464

Liu, X., Wang, S., Wang, P., & Wu, D. (2019). Automatic grading of programming assignments: An approach based on formal semantics. pp. 126–137. https://doi.org/10.1109/ICSE-SEET.2019.00022.

Macak, M., Kruzelova, D., Chren, S., & Buhnova, B. (2021). Using process mining for git log analysis of projects in a software development course. *Education and Information Technologies, 26*(5), 5939–5969. https://doi.org/10.1007/s10639-021-10564-6

Oman, R., & Hagemeister, J. R. (1994). Construction and testing of polynomials predicting software maintainability. *Journals of Systems and Software, 24*(3), 251–266. https://doi.org/10.1016/0164-1212(94)90067-1

Parihar, S., Das, R., Dadachanji, Z., Karkare, A., Singh, P., & Bhattacharya, A. (2017). Automatic grading and feedback using program repair for introductory programming courses. volume Part F128680, pp. 92–97. https://doi.org/10.1145/3059009.3059026.

Perez-Verdejo, J., Sanchez-Garcia, A., Ocharan-Hernandez, J., Mezura-E, M., & Cortes-Verdin, K. (2021). Requirements and github issues: An automated approach for quality requirements classification. *Programming and Computer Software, 47*(8), 704–721. https://doi.org/10.1134/S0361768821080193

Petkova, A. P., Domingo, M. A., & Lamm, E. (2021). Let's be frank: Individual and team-level predictors of improvement in student teamwork effectiveness following peer-evaluation feedback. *The International Journal of Management Education, 19*(3), 100538. https://doi.org/10.1016/j.ijme.2021.100538

Planas-Llado, A., Feliu, L., Arbat, G., Pujol, J., Sunol, J. J., Castro, F., & Marti, C. (2021). An analysis of teamwork based on self and peer evaluation in higher education. *Assessment & Evaluation in Higher Education, 46*(2), 191–207. https://doi.org/10.1080/02602938.2020.1763254

Rubinstein, A., Parzanchevski, N., & Tamarov, Y. (2019). In-depth feedback on programming assignments using pattern recognition and real-time hints. pp. 243–244. https://doi.org/10.1145/3304221.3325552.

Saidani, I., Ouni, A., & Mkaouer, M. (2022). Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering, 29*(1), 21. https://doi.org/10.1007/s10515-021-00319-5

Strandberg, P., Afzal, W., & Sundmark, D. (2022). Software test results exploration and visualization with continuous integration and nightly testing. *International Journal on Software Tools for Technology Transfer*. https://doi.org/10.1007/s10009-022-00647-1

Theunissen, T., van Heesch, U., & Avgeriou, P. (2022). A mapping study on documentation in continuous software development. *Information and Software Technology, 142, 10633*. https://doi.org/10.1016/j.infsof.2021.106733

von Wangenheim, C.G., Hauck, J.C.G., Demetrio, M.F., Pelle, R., da Cruz Alvez, N., Barbosa, H., Azevedo, L.F. (2018). Codemaster-automatic assessment and grading of app inventor and snap! programs. *Informatics in Education, 17*(1), 117–150. https://doi.org/10.15388/INFEDU.2018.08.

Wang, Y., Mantyla, M., Liu, Z., & Markkula, J. (2022). Test automation maturity improves product quality-quantitative study of open source projects using continuous integration. *Journal of Systems and Software, 188*, 11259. https://doi.org/10.1016/j.jss.2022.111259

Wunsche, B., Suselo, T., Van Der W, M., Chen, Z., Leung, K., Reilly, L., Shaw, L., Dimalen, D., & Lobb, R. (2018). Automatic assessment of opengl computer graphics assignments. pp. 81–86. https://doi.org/10.1145/3197091.3197112.

Youngtaek, K., Jaeyoung, K., Hyeon, J., Young-Ho, K., Hyunjoo, S., Bohyoung, K., & Jinwook, S. (2021). Githru: Visual analytics for understanding software development history through git metadata analysis. *IEEE Transactions on Visualization and Computer Graphics, 27*(2), 656–666. https://doi.org/10.1109/TVCG.2020.3030414

Publisher's Note