IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 11, 2020, accepted April 22, 2020, date of publication April 28, 2020, date of current version May 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2990980

Building a Comprehensive Automated Programming Assessment System

IGOR MEKTEROVIĆ, LJILJANA BRKIĆ, BORIS MILAŠINOVIĆ, (Member, IEEE), AND MIRTA BARANOVIĆ, (Member, IEEE)
Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
Corresponding author: Boris Milašinović (boris.milasinovic@fer.hr)

This work was supported by the European Regional Development Fund under Grant KK.01.1.1.01.0009 (DATACROSS).

ABSTRACT Automated Programming Assessment Systems (APAS) are used for overcoming problems associated with manually managed programming assignments, such as objective and efficient assessing in large classes and providing timely and helpful feedback. In this paper we survey the literature and software in this field and identify the set of necessary features that make APAS comprehensive – such that it can support all key stages in the assessment process. Put differently, comprehensive APAS is generic enough to meet the demands of ''any'' computer science course. Despite the vast number of publications, the choice of software turns out to be very limited. We contribute by developing Edgar, a comprehensive open-source APAS which, to the best of our knowledge, exceeds any other similar free and/or open-source tool. Edgar is the result of three years of development and usage in, for the time being, eight courses dealing with various programming languages and paradigms (C, Java, SQL, etc.). Edgar supports various text-based programming languages, multi-correct multiple-choice questions, provides rich exam logging and monitoring infrastructure to prevent potential fraudulent behaviour, and subsequent data analysis and visualization of students' scores, exams, question quality, etc. It can be deployed on all major operating systems and is written in a modular fashion so that it can be adjusted and scaled to a custom fit. We comment on the architecture and present data from real-world use-cases to support these claims. Edgar is in active use today (1000+ students per semester) and it is being constantly developed with new features.

INDEX TERMS APAS, computer science education, courseware, educational technology, software architecture, scalability.

I. INTRODUCTION

It's been nearly 60 years since Hollingsworth [1] reported the first use of automated program for code testing, but the reality is that programming assignments are still managed manually in most classrooms [2]. Teachers assess submitted code, performing compilation and testing or just visually scan the solutions. The fact that a single problem can be described with different algorithms and the same algorithm can be implemented in a number of different ways burdens the grading process when the grading is done manually. Well known problems of manual evaluation of programming assignments are the objectivity and consistency of the criteria as well as the quality and timeliness of the feedback received by the student. The lack of feedback can discourage students if they often fail and do not receive assistance to improve [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

On the other hand, CS schools all over the world, even the Ivy league faculties, are facing a shortage of staff, while the number of students increases. As Singer [4] states: *"the surge in student demand for computer science courses is far outstripping the supply of professors, as the tech industry snaps up talent."* The number of students enrolled in higher education programming courses is regularly counted in hundreds [5] and is often in opposition with the number of teachers and teaching assistants responsible for conducting the course. When the grading for the large class sizes is performed manually, the workload can grow to unmanageable extent. Shortage of staff paired with advantages and convenience that Automated Programming Assessment Systems (APAS) bring are the reasons why APASs are now more relevant than ever and will be used even more in the future.

Four years ago, we were faced with the challenge of introducing an APAS in the university environment with thousands of students per year. The very first courses on

TABLE 1. Edgar's usage statistics for the top five courses.

Course	Years used	No. of exams	No. of questions	No. of students
Databases	3	17,121	73,612	1,295
Object oriented programming	2	5,384	24,638	747
Programming in C	3	27,931	117,614	2,161
Software development	3	1,166	29,597	313
Advanced databases	3	961	3,004	254

which the system was to be used were the undergraduate course Databases and graduate course Advanced Databases, with, on average, 431 and 87 students enrolled respectively. We wanted to automate the assessment of programming assignments for the SQL query language and MongoDB's MapReduce. SQL query evaluation fundamentally differs from the evaluation of general-purpose code (such as C, Java, etc.) as one must compare obtained record sets in a parametrized fashion (e.g. whether the row ordering matters, whether column names matter, etc.). Additionally, some SQL statements create, alter or delete database objects such as tables, indexes, etc. and their correctness cannot be checked with a predefined output data. Following the database courses, the plan was to support other courses using interpreted or compiled programming languages like C or Java. From a more generic perspective, we wanted a standalone solution that can support all stages in the assessment process (see the Section III on APAS requirements). Beside these functional requirements, APAS should also be robust and scalable, able to support hundreds of students working in parallel with logging and monitoring facilities.

The problem is that, for those set of requirements, there has not been and there still isn't any publicly available APAS – only different systems targeting particular use cases.

As a solution, we created Edgar – an open-source APAS developed at University of Zagreb Faculty of Electrical Engineering and Computing (abbreviated FER) and presented in this paper. Besides SQL and MongoDB's MapReduce, Edgar supports any programming language that can be executed on a Linux machine (C, C#, C++, Java, Python, etc.) and meets the aforementioned requirements. The novelty of our approach is that it is designed from the ground up in a general and comprehensive way to support a wide range of users, that is, any programming language or paradigm, and accompanied with necessary modules for management (monitoring, logging, analysis) often neglected in the literature. Such generic approach calls for modular and scalable architecture that Edgar is founded on.

Edgar has been in use since the spring semester of 2016/2017, is still used today and it is going to be used in the foreseeable future. It has become an indispensable tool for programming courses at FER – Table 1 shows the usage statistics for top five courses with respect to the number of enrolled students.

We believe that automated and online assessment tools present a modern addition to the classic methods of programming skills assessment and that they will become a standard tool to CS teaching in the future. For instance, in our courses we typically combine several automated assessments (exams in a laboratory), several automated homework assignments, tutorials, adaptive exercises and two handwritten and human assessed exams per semester. All the while, we collect the underlying data (answers, logs about students' behaviour during automatic assessment etc.) which can be used in different ways, e.g. to improve the content and process, to profile students, predict success [5], etc. Although Edgar is presented here as APAS it is more than that – it has been extended with LMS features such as tutorials and adaptive exercises which will not be covered here for the sake of focus and clarity.

The rest of the paper is structured as follows: Section II comments on related work, Section III focuses on functional aspects – it defines the requirements of a comprehensive APAS and comments on Edgar's implementation. Section IV deals with non-functional aspects - Edgar's architecture and scalability. Section V compares our work with similar studies, discusses Edgar's educational aspects and comments on future work, followed by the conclusion.

II. RELATED WORK

Many APASs have been developed since the appearance of computer program assessment by automated assessment systems, such as the Rensselaer grader [1]. Their popularity has increased with the appearance of online coding contests [6] and challenges and Massive Open Online Courses (MOOCs) [7]. The benefits of using APAS have been widely examined and reported in scientific literature, as shown by the large number of published reviews of various facets of automated assessment tools, e.g. [8]–[16]. They are particularly well accepted in computing education [17], but rarely used outside the institutions in which they were developed, although lecturers in other institutions recognize their advantages. According to Roßling *et al.* [3] teachers are often too busy to find, learn, design exercises and integrate APAS into their courses, especially when the notation and methodology of the system do not precisely match those of the course. In [10] authors noted that (too) many new systems have been developed every year sharing common subset of features with a small number of new ones. Although the authors suggest that new features should be added to an existing system rather than developing own one, they recognize the plea for development of a new system due to existing systems (un)availability, lifespan, distribution (un)suitability, and lack of examples and explicit explanation how an existing system work. For instance, some systems and tools that have appeared in almost every literature review on automated assessment in computer programming and gained significant impact by number of citations (with more than 100 citations on Google Scholar) were either not updated in a decade (BOSS [17], [18]), not available (AutoLep [19]) or based on obsolete technology (CourseMarker [20]).

Amelung *et al.* [21] noted that components of an automatic assessment systems are usually strongly coupled making a

monolithic system impossible to extend and that assessment tools should be developed in service-oriented manner. Among the automatic assessment tools they reviewed, Amelung et al noted that only Moodle is extendable and open source. In addition to Amelung's research, two modular systems for automatic assessment should be noted: JACK and Web-CAT. Web-CAT is open-source and *"as of March, 2019, had been used across 39 universities"* [22] while JACK is proprietary and available only to German universities [23]. However, both JACK and Web-CAT lack features for our use-case (e.g. exam logging and monitoring, SQL evaluation, etc., see the following section). Additionally, some recent remarks from Web-CAT users point to drawbacks of its user interface and a feedback usefulness [24].

Thus, we identified Moodle as the only potentially viable option. Some courses at FER use Moodle and so we find it relevant to compare Moodle and CodeRunner plugin with Edgar. Moodle can run CodeRunner [25], a free and open-source question-type plug-in that lets teachers set questions where the answer is program code. CodeRunner is capable of evaluating programming assignments in Python, C, JavaScript, PHP and almost any other text-based programming language. CodeRunner can assess SQL assignments as well, but in an unsatisfactory way. SQL is assessed through its templating system and dynamic testing is administered: CodeRunner compares string/console representations of recordsets as expected input-output pairs. This is unnatural and inadequate for SQL: it is not possible to allow for different row and/or column order and consequently it is impossible to give good feedback; not to mention that it is closely bound to SQLite database and its string representation of a recordset. Should one change the database provider all questions (test-cases) would have to be changed! It is also not possible to test DDL constructs like `ALTER TABLE`, `CREATE INDEX`, etc., all of which is possible in Edgar. SQL simply has a different output data structure and demands custom treatment. Less importantly, MapReduce programming assignments in Mongo or any other DBMS are not supported in CodeRunner.

In both systems, in addition to programming assignments, other computer-graded question types like multiple choice or short answers are supported. The grading options, including penalizing fails on some test-cases and ignoring fails on others, as well as the format of the feedback presented to the student are pretty much alike in both systems. Even some limitations are similar, like the one that restricts the answer to a single compilation unit (segment of code). In conclusion, while they are comparable to a great extent, Edgar exceeds Moodle+CoreRunner capabilities when SQL and MapReduce evaluation is concerned. Also, Edgar provides better exam logging and monitoring, and data analysis facilities, as discussed in the following section.

III. COMPREHENSIVE APAS

A comprehensive APAS supports different stages in the assessment process in a typical course: from course administration, content authoring, exam conducting, logging and problem mitigation to subsequent data export and analysis. Comprehensive APAS should be flexible and extendable in order to support heterogeneous requirements in the computer science domain. Teachers should be empowered to independently administer the entire course through the user-friendly GUI and potentially export all the data collected. In order to classify an APAS as "comprehensive" we've identified necessary or minimal feature set. Some of these features are often neglected or not discussed in the literature, namely the first two rows in Fig. 1. This feature set is compatible with the Abello et al's requirements for e-assessment system (*"allow the delivery of assessment activities, the recording of responses, timely feedback, automatic grading, and weighted-average grade calculation"*) [26], Noonan's grading system phases (*"delivery, assessment, marking, return"*) [27], and extends the Wang et al's. conception of a web-based assessment and test analysis (WATA) system based on Triple A model (*"assembling, administering, appraising"*) [28].

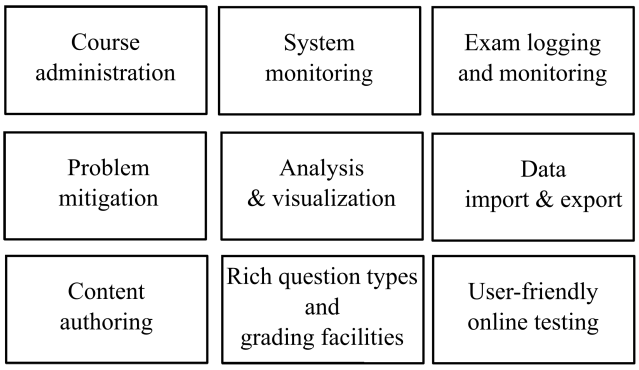

FIGURE 1. Minimal APAS feature set.

We comment on each feature in the following sections:

A. COURSE ADMINISTRATION

Examinations are administered in the context of courses and courses are enrolled by students in various academic years. As a rule, universities are already equipped with one or more information systems that store that information and APASs should have only a lightweight version of course/academic year enrolment administration which will simply bin the students to their respective courses and academic years. Analogously, there should be facilities for administering teachers and their course permissions. Edgar supports enrolling the students to a course via GUI which parses a tab-separated list of students typically procured from the university's information system. A related technical issue is user authentication which is also supposed to build upon the university's authentication facilities – consequently, a general purpose APAS should support rich authentication options. Edgar supports various authentication options, that will be elaborated in Section IV.

B. SYSTEM MONITORING

As a rule, all systems should be monitored, and APAS is no exception. Monitoring gives insight into system scalability

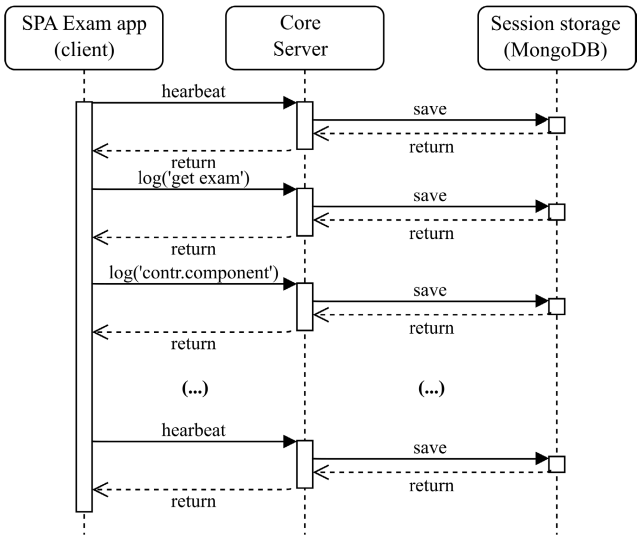

FIGURE 2. Student's application logging events during an exam.

and helps prevent future problems (e.g. disk shortage). We are monitoring Edgar through a well-known 3rd party software Zabbix [29] and have additionally developed a custom configurable monitoring daemon which dispatches notifications via email when configured conditions are met (e.g. average CPU usage over 5 minutes is over 50%, etc.)

C. EXAM LOGGING AND MONITORING

All actions a student undertakes on the exam page in the browser should be logged. This helps teachers conduct the exam, monitor students as they take the exam, and helps resolve any issues later, after the exam has been submitted. Surprisingly, this topic is neglected in literature; more so since it plays a very important role in the real-world examination process. Edgar provides a rich set of features for exam monitoring. During the exam, the client single page application logs all events that occur to the server via AJAX calls and also sends a heartbeat to indicate whether it is alive (Fig. 2 and Fig. 3). During the exam, teachers have a dashboard with overview of students currently running an exam with basic information – personal details, IP address, image, lost focus flags and a heartbeat. Fig. 3 shows exam monitoring dashboard. We group the students according to their IP ranges (each range in a different tab) corresponding to lab computers. Lost focus flags are useful in the context of multiple-choice questions where, in our case, it is typically forbidden to lose focus from the exam window, as that might indicate student browsing the web in another window to find the solution.

Teacher can even "stalk" the ongoing exam to see the detailed event log of the ongoing exam and current student answers. Event log is also visible later, post-submission, both to teacher and student. Event log has proven to be a very valuable asset for resolving potential misunderstandings, software bugs, and sometimes sadly, for detecting plagiarism among students, e.g. a student attempting wrong solution many times and then suddenly submitting a completely different - correct solution (which was acquired from another student).

D. PROBLEM MITIGATION

Exam logging and monitoring helps mitigate a number of issues caused by student, but sometimes teachers make mistakes when creating questions. Mistakes are normal in the course of this process, and APASs must provide support for error correction and manual points assignment. Edgar supports editing the already submitted exams by a teacher and a resubmission. Alternatively, it is possible to simply assign points to one or more students for a given question and thus override the inital grade.

E. ANALYSIS AND VISUALIZATION

Questions can also be "wrong" in a different sense – what is the point of a question that was correctly (or wrongly) answered by a vast majority of students? A comprehensive APAS must provide analytical capabilities to facilitate the content curation and evolution. Also, students should be able to see their scores visualized. Edgar provides several visualizations pertaining to exams, questions and students' behaviour during the exam. We present one visualisation from each category: a box plot in Fig. 4 is presented for all exams where score is not ignored.

When an exam is selected by clicking on the box, additional details are displayed: the timeline with exam statuses (in progress, started, submitted), exam pass percentage and score distribution percentage with average and median marks.

In the question category, we have developed a novel visualization to easily detect dubious questions (Fig. 5). It shows one bubble for each question group. The size of the bubble corresponds to the number of questions in the group. Question group is defined as a set of questions having the same:

- Average score percentage: average score percentage of a single question is calculated by averaging score percentage with one modification: negative scores are replaced with 0.
- Number of instances: number of times a question is taken

Abscissa shows the number of instances of a question group, and ordinate shows the average score percentage. For instance, the question group (bubble) in the upper left corner in Fig. 5 contains three questions having average score percentage 100% and number of instances 1. Obviously, the statistics are more significant as we move to the right and the number of instances increases. As we move to the right, questions groups that occupy the extreme regions (very green ones close to 100% and very red ones close to 0%) should be inspected as they apparently do not provide any discriminative power – they are either too easy or too hard. This is a very compact way of showing hundreds of questions in a single chart. Chart can be filtered by question types, authors and exams. Typically, we filter by the exam, as to analyse the

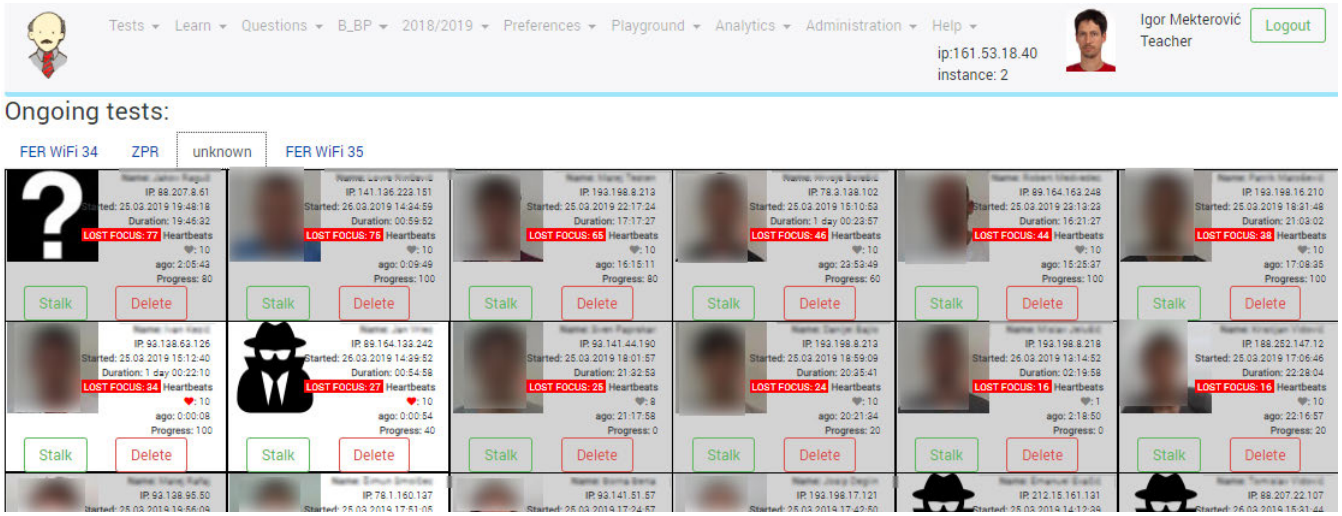

FIGURE 3. Exam monitoring view, students are grouped according to IP ranges, faces blurred on purpose.

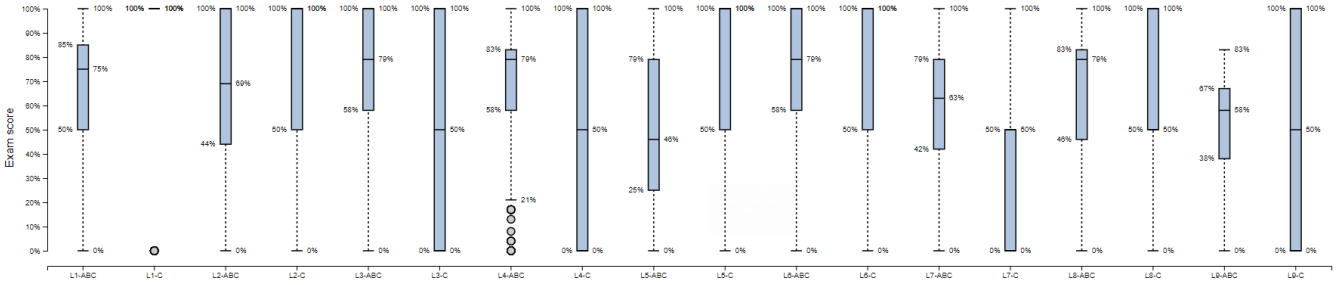

FIGURE 4. Exam score box plot.

questions from the recently finished exams and potentially take corrective actions. When clicking on the bubble, the right side of the screen shows the question details, with links to questions and additional per-question analytics. There are of course other ways of assessing questions - Edgar can also calculate question difficulty based on the Rasch model [30] of item response theory and Lord's two-parameter model [31] (used in Edgar's adaptive exercises).

Lastly, student behaviour plot shows the application events on the time scale. Fig. 6 shows student behaviour analytics for an exam that took place from 9AM to 7PM. Students were divided into groups and time slots for the two hour exam were allocated. This was a multiple-choice exam, and in our case ''Lost focus'' events shown in red raised concerns. The details are provided on the tooltip, and subsequently all such cases were inspected, and actions taken. Sometimes students lose focus for few seconds or even for less than a second and such cases were ignored.

F. DATA IMPORT AND EXPORT

APASs are usually just an addition to existing information systems and that is why it is important to enable data import and export to and from an APAS. For instance, in Edgar, students with their names, outer information system IDs and images can be imported and ultimately exported (in a CSV format) along with their scores from various exams. In our case, these scores are then combined with those of manually graded paper-written exams to form the final grade.

G. CONTENT AUTHORING

A support for the content design should be simple enough so that even a non-technical teacher can use it to create content and at the same time support a rich set of features: rich text formatting, images, tables and mathematical formulas. Question writing is often a group effort. That is why we suggest that APASs must have inbuilt support for question versioning and review process (e.g. teachers are paired in reviewing each other's questions). In Edgar, question text is written in Github flavoured markdown. Markdown is chosen because it is a simple, human readable format that can be efficiently searched using various full-text search facilities. Due to limitation of basic markdown syntax, Github's version of markdown is used to support tables and figures and MathJax library [32] is used for LaTeXstyle mathematical formulas. If a question that was already used by a student is saved via the edit form (e.g. to fix a bug in the question) Edgar will save

FIGURE 5. **Bulk question analytics.**

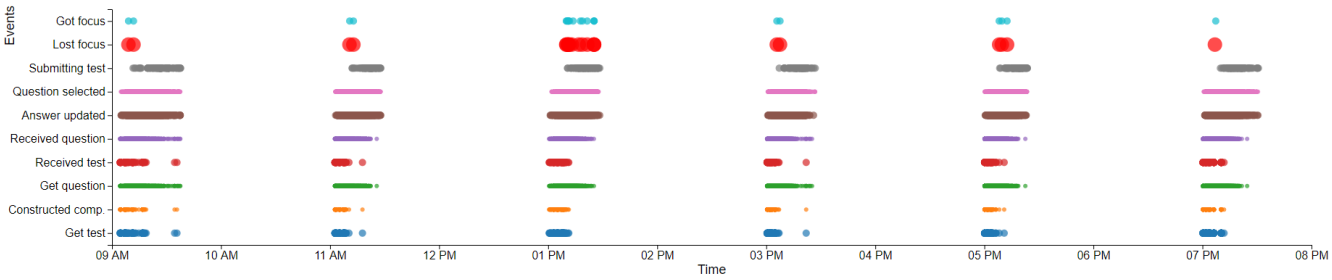

FIGURE 6. **Student behaviour analytics.**

changes to a new question while automatically deactivating the old question so it does not to appear in future exams. This keeps the history accurate, and past students' exams with erroneous question will still be linked to the original question. This behaviour can be overridden via "save in place" option which does not instantiate new question that could be useful for fixing (minor) text typos. Additionally, questions can have custom tags having meanings (e.g. "requires-attention" to mark a question to be checked) and list of reviewers. In our experience it is very important to structure a strict question writing/reviewing process and to assign responsibility for the question to author and reviewer. Questions are organized into a network of nodes of arbitrary types (e.g. unit, module, tutorial, etc.) and a question belongs to one or many nodes. In practice, we've encountered a dilemma whether to structure the questions and nodes thematically or organizationally. It appears that organizational structure is more used, but often both are used as shown in Fig. 7 and Fig. 8. Except in rare cases when the same unit, module or tutorial is used in several courses, nodes are typically structured hierarchically in a tree.

Once the questions are grouped in nodes it is possible to define an exam using those nodes. An exam can have multiple parts where one exam part corresponds to one node and additionally defines minimum and maximum questions to be generated at random from the assigned node. Building on the example in Fig. 7 we could, for instance, define an exam with three questions using two exam parts (EP):

- Exam1, N = 3
  - EP#1, node = *ints*, min = 1, max = 2
  - EP#2, node = *float*, min = 1, max = 2

Such definition will generate at least one and at most two questions from the ints node; likewise, for the float node, possible sets are: $\{Q1, Q2, Q3\}$, $\{Q1, Q2, Q4\}$, $\{Q1, Q3, Q4\}$ and $\{Q2, Q3, Q4\}$. Edgar provides GUI for all these operations (e.g. exam definition in Fig. 9) which allows teachers to autonomously create questions and exams. Additionally,

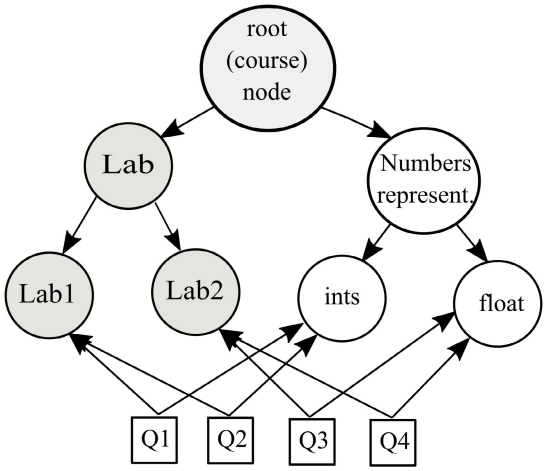

FIGURE 7. Parallel node structure - organizational (gray) and thematical (white).

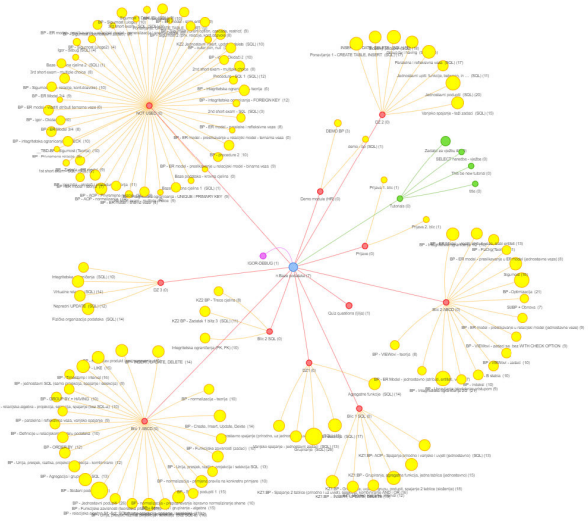

FIGURE 8. Real world node structure from the Database course. Bubble sizes reflect the number of questions. Bubble (course units) labels are not meant to be readable.

a grading model (defined in the following chapter) is attached to the exam parts. Fig. 9 shows an exam definition form with two questions and two exam parts – one being a coding question and the other a multiple-choice question. In this setup, students can achieve exam score ranging from −0.5 to 3.0.

H. RICH QUESTION TYPES AND GRADING FACILITIES

Most research on APAS focuses on issues in this category: various question types and programming language and various ways of evaluation (e.g. dynamic code analysis, static code analysis, etc.) and grading of submissions. A comprehensive APAS should support "all" programming languages or, in other words, be easily extendable to support new programming languages. Furthermore, languages should be decoupled from concepts being tested (e.g. entire compilation unit, part of the unit like function or a code line, metadata like existence of some public method) as much as possible. For instance, Moodle's CodeRunner uses templating system to test different concepts which results in many "question types" (i.e. templates) like *c_program*, *c_function*, *java_method*, *java_class*, etc. In our opinion, Edgar uses more elegant and equally powerful, *prefix-suffix* system which could be interpreted as ad-hoc templating. However, we should distinguish here two major categories of programming languages:

- "standalone" programming languages, like C, Java, Python, Haskell, etc. – languages that come with their own compiler or interpreter which is used to execute the code autonomously, in a sandbox
- "managed" programming languages, like SQL, HTML, CSS or MongoDB query language which are executed in some context, as a part of a bigger context

To the best of our knowledge, Edgar is the only APAS available that can handle both SQL and "standalone" programming languages (see comment on Moodle in Section II). Besides multi-correct multiple-choice questions and free text questions for gathering feedback, Edgar supports "any" "standalone" text-based programming language: adding a new language is just a matter of installing a language compiler and adding one database tuple of metadata (compiler invocation, etc.). As mentioned, "managed" languages demand custom treatment and Edgar's modular architecture abstracts this problem via various external code runners (further described in Section IV). Edgar ships with three external code runners: one that covers dozens of "standalone" languages, PostgreSQL code runner for SQL and JSON/MongoDB code runner. Table 2 provides a list of currently implemented question types and comparison methods. Currently Edgar supports only dynamic testing of "standalone" languages, while SQL and Mongo are tested in a similar vein – by executing correct and submitted queries and performing a parametrized result comparison.

Grading in Edgar is performed in two steps: first, the *correctness* of the submission is evaluated (1) and then the *correctness* is used with the assigned *grading model* (2) to calculate the *score* (3). Correctness is defined with the following formula:

$$correctness := max\left\{0\%, 100\% - \sum_{fail} PP(fail)\right\} \quad (1)$$

where PP is the *penalty_percentage* - the percentage of points **to deduct** (starting from 100%) if student missed the correct answer in the multi-choice question or failed a test-case in the code question. SQL and Mongo questions have a single test-case (i.e. recordset comparison) with PP = 100%. Consequently, the former can have correctness in the [0, 1] range, while later have binary correctness – 0 or 1. Grading model defines how to calculate question score from answer correctness and is defined as a triple of real numbers:

$$gm := (c, i, e) \quad (2)$$

where c, i, and e are values for correct, incorrect and empty answer respectively. Values for those numbers are custom and

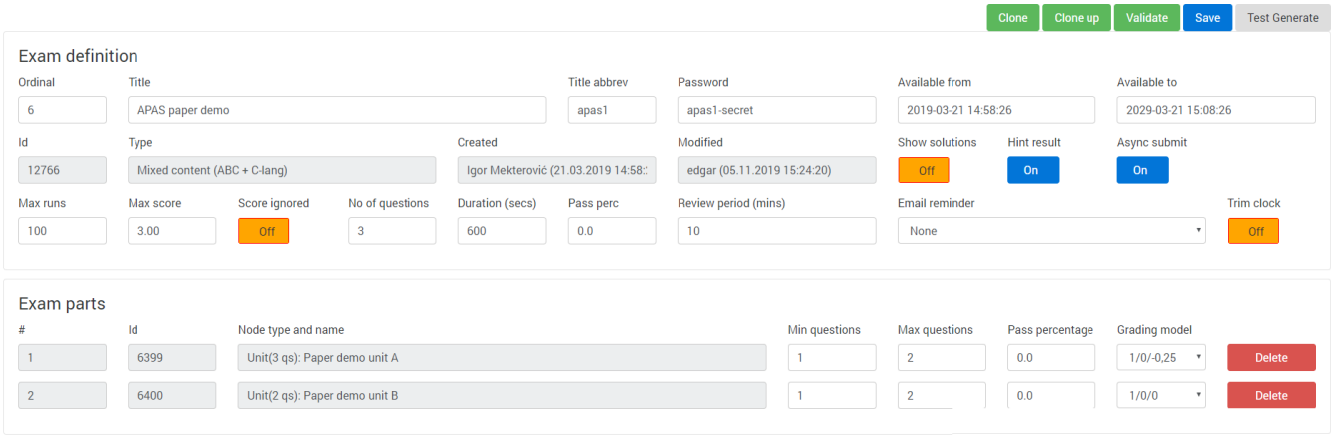

FIGURE 9. Exam definition form with N = 3 questions and two exam parts with different grading models.

TABLE 2. Currently implemented question types and the associated properties.

Question type	Comparison method	Comparison parameters
Multiple choice	trivial: comparison with stored correct answers;	
Free text	non-evaluated questions, used in questionnaires for gathering feedback	
SQL	questions evaluated against a database returning a record set (comparison of two record sets)	ignore tuple or columns order and/or column names
JSON	questions evaluated against some engine returning JSON (comparison of two objects)	deep or strict comparison
Code	programming language questions, e.g. Java, C, C#, C++, Python, that get compiled and executed in a sandbox environment (comparison of expected and actual standard output)	ignore case, trim whitespaces, use regular expression matching, use random input data

open to different grading strategies. We use the following guidelines: value for unanswered question is set to 0; value for incorrect score is set to 0 for coding questions (we do not penalize a failed coding attempt) and for multiple choice questions we set the incorrect score to negative value to discourage students from guessing. In case of partially correct answer, the answer score is calculated proportionally to their answer correctness and grading model.

$$score = \begin{cases} gm.e & (for\ empty\ answer) \\ correctness * (gm.c - gm.i) + gm.i \end{cases} \quad (3)$$

Additionally, the score can be further modified with two additional variables: N - number of code runs, and T – relative time left (starting from 1 and approaching 0 as the final value). Teacher can enter a custom formula to define the final value, e.g. $[S]*(-0.05*[N]+1.05)*(0.7*[T]+0.3)$ where S is base score described above. A detailed example of C and SQL question evaluation is provided in the Appendix.

I. USER-FRIENDLY ONLINE EXAMINATION

Unlike all others, this feature is student oriented. Students should be able to write exams in the laboratory, at home (homework) using different devices (desktop, mobile) and operating systems. With such requirements the only reasonable option is to implement an exam as a responsive web-application i.e. take an exam via internet browser. Exam should expose clean and simple user interface and provide appropriate, potentially immediate feedback to the submissions. Once an exam is submitted, a student must be able to review the exam having potential mistakes clearly marked. On the technical side, the exam taking process must be robust both in terms of scalability and resilience to errors (ranging from power outages to software bugs). Scalability is achieved through modular and horizontally scalable architecture. To facilitate error resilience exam state should be saved continuously. In Edgar, the part of the web application used to conduct the exams is written as a single page web application (SPA) using the Angular framework. Students start the exam by typing in the exam password given to them by the teacher. On password submission, various checks are employed: if the password is correct, if the exam is available at the present time, and if the student is enrolled in the course the exam belongs to. As the student progresses, the exam questions are loaded via calls and cached. This SPA approach has clear advantages for this use case over classic full-page refresh: the server is relieved of additional burden of serving the same questions multiple times and user experience is also better without the full-page loads. Edgar persists session data to MongoDB (see Section IV), so a student can close the browser or even restart a computer and continue writing the exam without losing the data: exam instance is already generated, and current exam answers are stored in Mongo. This is particularly convenient with the long-running exams. For instance, in some courses we have homework assignments for which students are allowed an entire week's time to submit.

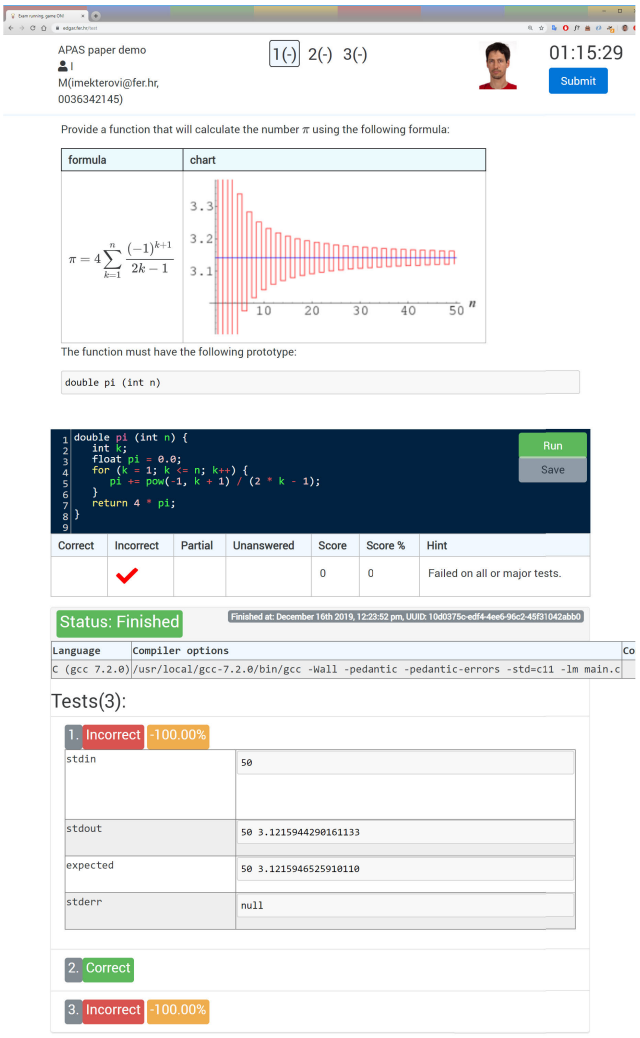

FIGURE 10. Example of an exam. The answer to the current C programming language question passed one of three test-cases (the first test-case is public).

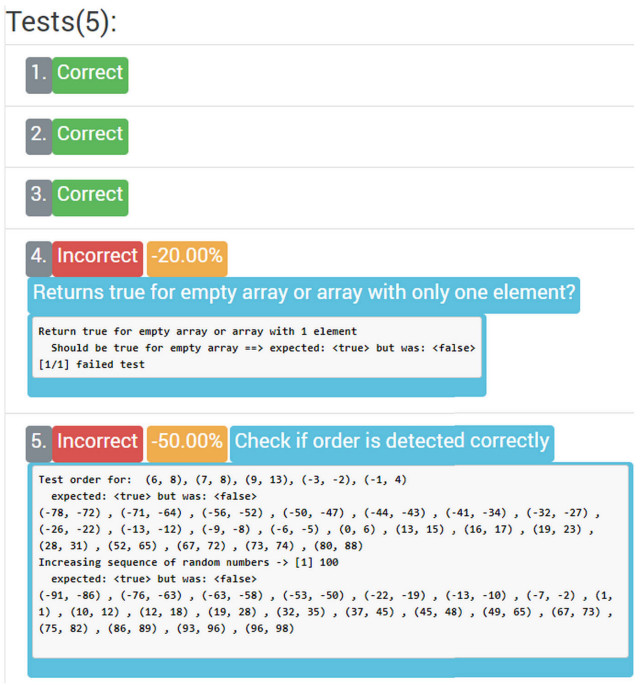

FIGURE 11. Captioned test-cases with detailed, programmatically generated, feedback.

Students then solve a part of the assignment on one day, take a break, and continue some other day.

Fig. 10 shows a C programming question asking a student to provide a function that calculates the approximate value of the number π. A student has provided a faulty solution, ran the solution and got immediate feedback. Students can run their solutions an arbitrary number of times before submitting the exam and Edgar is customizable with regards to what feedback it provides to students: feedback can range from very detailed to just stating whether the code compiles or not. Test-cases can be public or private - a public test-case displays all test-case data to the student and private displays only whether the test-case is correct or not. In the example in Fig. 10, Edgar provides feedback on how many test-cases are run, and whether students passed them (three test-cases in this case, the first test-case is public, one passed, each carrying a penalty percentage of 100%). For a more detailed feedback, test-cases can have captions and verbose outputs, e.g. Fig. 11 shows detailed feedback programmatically generated from teacher's code for two failed test-cases (for a different question, carrying 70% penalty in total). In SQL questions, feedback will provide hints as to what is different in the resulting data and expected dataset (e.g. *"uneven row count"*, *"student.first_name <> correct.first_name on row #33"*, etc.). With Mongo (JSON) questions, a detailed object difference will be shown (missing/excess properties, different values, etc.). Given the limited time and the unforgiving nature of automated assessment we typically provide rich feedback to our students.

When students submit an exam, they have a limited configurable time (e.g. 10 minutes) to review the exam. In the review phase, they can see all the data – what data was passed to the *stdin* and, if so configured, the correct answer (e.g. question shown on Fig. 12 is configured not to show correct answer). Students can also re-run both their code and the correct code. This is a feature useful for reclamations, as sometimes teachers provide wrong answers, or results are indeterministic due to e.g. using the random function, or sometimes numeric errors occur, or code behaves differently on different platforms etc. Fig. 12 shows the review exam view where all evaluation data are visible. The student incorrectly used the float type instead of double and failed two out of three test-cases due to differences in precision of those two types.

Exam results obtained during submission evaluation are typically identical to those obtained with re-run if the question wasn't changed in the meantime or functions for generating random numbers (e.g. rand()) were not used somewhere in the code or to generate *stdin*.

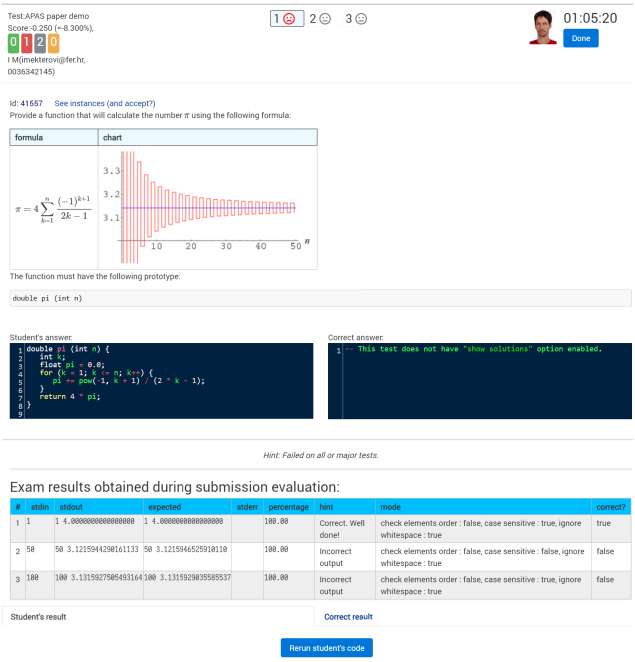

FIGURE 12. Exam review. The student can see saved evaluation data and can re-run their code. The student incorrectly used the float type instead of double and failed two out of three test-cases due to differences in precision of those two types).

J. ADDITIONAL FEATURES

Edgar has an accompanying cross-platform mobile application for Android and iOS platforms written in React Native suitable for use in scenarios where evaluations consist only of multiple-choice questions or where use of classroom clickers is considered. The mobile application is limited to multiple-choice questions because smart phones are not suitable for writing programming code. Apart from testing students in laboratory or at home, Edgar also supports lecture quizzes akin to Kahoot [33], AuResS [34] or Wooclap [35]. Lecture quiz is an exam with fixed questions and fixed questions order. Code assessment facilities of APAS lend themselves well to the construction of virtual learning environments. In Edgar, two such options exist - tutorials and adaptive exercises, though, for the time being, supporting only SQL and multiple-choice questions. Due to the current focus on APASs and space constraints, they will not be further discussed here.

IV. EDGAR'S ARCHITECTURE

Edgar features a modular and scalable architecture. Various parts of the system can be deployed across different servers and scale horizontally. Edgar is written in node.js and can run on all major operating systems. Fig. 13 shows Edgar's architecture. Components marked with "LB" are typically load balanced and 3rd party components are shown in gray. Modular APAS architectures have been previously discussed in literature, for instance, Edgar's architecture is in agreement with the concept and functions of frontend and backend components proposed in [21] with a minor exception that spooler (broker) is integrated in the core system (however, assessment workflow is the same).

Edgar comprises the following optional or obligatory modules, some of which are third party components, as stated in the following list:

- **Core system and persistance** (obligatory): the core system is implemented as web application. There can be an arbitrary number of instances hidden behind and load-balanced by Nginx. This is how the main application can gracefully scale horizontally. All that is needed to scale up the system is to start another node and add it to the Nginx load balancing configuration without any downtime. This is possible because, in the spirit of polyglot persistence, Edgar web application uses two different databases to persist data: PostgreSQL relational database as the main data store and MongoDB for transient session data and various event logs. Edgar departs from the default scenario where the main memory is used for session data, but rather stores sessions to the MongoDB. This incurs a tiny overhead in terms of speed but allows for stateless core system instances that can be dynamically adjusted since the state is stored in MongoDB. Also, it allows us to restart arbitrary nodes or the entire server without breaking students' sessions. For instance, if there is a power failure and students are writing the exam and the whole system (clients and servers) loses the power supply none of the current exams and unsubmitted answers will be lost, and after the power supply is restored - the exams can gracefully continue. To achieve that, answers to ongoing exam's questions are automatically saved to MongoDB each time a student runs the answer for evaluation, whether the answer is correct or not. Both MongoDB and PostgreSQL database placement are arbitrary. There are of course various ways to scale PostgreSQL and Mongo, but those are not of our making and they will not be discussed here.
- **Frontend** (obligatory): comprises of the SPA web-application used by students to take exams and the classic web-application for teachers and students for all other tasks described in the previous chapter. Both feature modern and responsive GUI.
- **Authentication** (obligatory): Edgar supports various authentication options. It is possible to authenticate using local authentication or via external identity service provider using SAML [36] or OAuth2 [37] protocol. It is possible to mix authentication modes even for a single course – for instance some students (and teachers) could be authenticated via Twitter and some via Google.
- **Nginx reverse proxy and load balancer** [38] (optional, 3rd party): it is a fairly common scenario to use Nginx as a front end for the node.js (and other web) applications. Nginx forwards request and performs load balancing for a cluster of node.js applications. Also, Nginx forces https protocol and works as a secure web server. From a

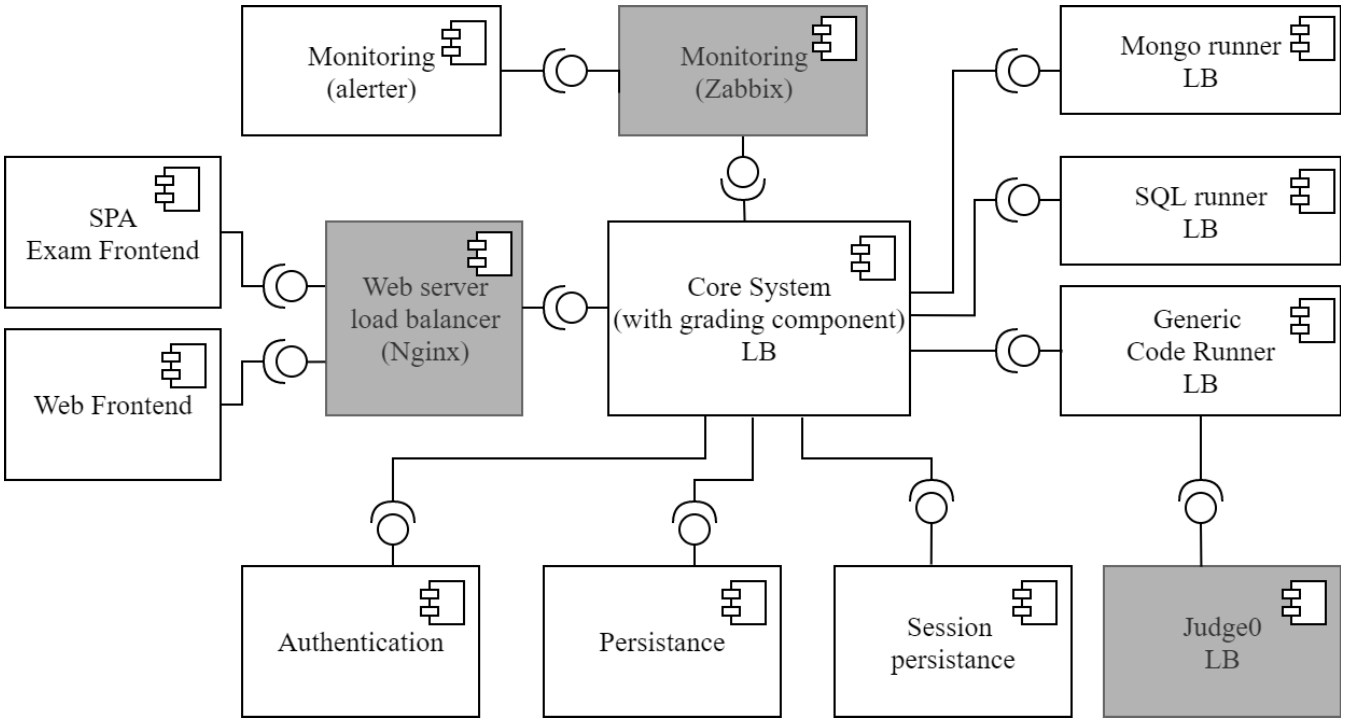

FIGURE 13. Edgar's modular architecture (3rd party components shown in gray).

security standpoint, such setup is also beneficial because it allows us to run all our web applications locally or in a demilitarized zone and expose only https port to the public via heavily used and tested Nginx web server.

- **Various runners** (optional): Edgar abstracts all "code running" by the use of so called "runners". Runners are independent web endpoints that accept requests to run some code and return a corresponding data structure (e.g. dataset, textual output). Runners for different programming languages are developed as separate projects within the Edgar group repository [39]. The following runners are currently available, covering most mainstream programming languages:
  - **SQL-runner**: PostgreSQL runner which accepts a SQL command, executes it against a configured PostgreSQL database using configured timeout and returns the record set or an error message. SQL-runner performs all statements within a transaction that is always rolled back. This provides a sandbox of sorts. This does not mean that only read-only `SELECT` statements are allowed – it is possible to execute `INSERT`, `DELETE`, `UPDATE`, or even DML statements like `ALTER TABLE` and `CREATE TABLE`, `INDEX`, etc. SQL-runner simply executes them, retrieves the required datasets, and then rolls back the transaction. This code could be easily cloned and modified to support any other RDBMS provider. SQL-runners are typically configured to run on the different copies of the same test database to avoid concurrency issues and increase performance. For instance, for our Database course with typically 400-500 enrolled students we run 5 SQL-runners each connected to the test databases db1, db2, ..., db5 having the same content and structure. Each SQL-runner instance is tied to just one database to leverage the connection-pooling performance benefits.
  - **mongo-runner**: MongoDB runner accepts MapReduce and find queries, executes them against a MongoDB database and returns the resulting JSON or an error. Mongo does not have a query language in the traditional sense but provides an object-oriented Javascript API to query documents. Therefore it is not possible to simply forward student's code (e.g `db.collection.find({id:123})` to Mongo's engine like it is possible with the SQL statements. Mongo-runner therefore parses the received statements and then uses the Mongo API to execute them. This is why not all statements are currently supported but only `find` and `mapReduce` which we use to test students in the Advanced Databases course.
  - **code-runner**: Code runner accepts code in "any" programming language (C, C#, C++, Java, Python, etc.) and an array of input strings, executes them, and returns the corresponding array of output and stderr strings, as well as rich metadata about the process. Code-runner relies on Judge0 API [40] for the code execution (described below). Supported

languages are all those that can be installed and executed on a Linux operating system.

- **Judge0 API** (optional, 3rd party): Judge0 API is a free and open-source Web API for executing and grading untrusted source code [40]. Judge0 API itself provides a wrapper for a well-known isolate [41] sandbox created to safely run untrusted executables. Judge0 also provides the adjoining infrastructure to receive programming code, queue the code execution tasks and execute them with a configurable number of workers using isolate, persist the results and return them to the client.
- **Monitoring software** (optional, 3rd party + own): Edgar uses a well-known open-source monitoring software: Zabbix [29]. Zabbix has its own MySQL database for storing monitoring data. Zabbix can monitor multiple servers by installing Zabbix agents on those servers. Building upon Zabbix, a custom *alerter* application was developed to send email notifications when configured thresholds are exceeded. It is possible to set thresholds on disk, CPU and memory in a given timeframe. For instance, one rule may be "send notifications if average value of used memory calculated over a time span of 10 mins is over the 50% threshold". Monitoring, in the context of overseeing students taking exams, is implemented in the core system.

This modular architecture is somewhat harder to install and maintain due to the number and variety of components, but it provides the flexibility and the ability to scale practically any part of the system to a custom fit. For instance, Edgar was used to test C programming code using GCC compiler on a course with 700+ students and worked fine with average execution time of C programs of approximately one second with typically 60 students working in parallel. Subsequently, the Object-oriented programming course with 700+ students using Java was started with the same infrastructure but had faced performance issues (Fig. 14).

Java compiler is slower than GCC, and the execution of programs is also far more memory and processor demanding since Java virtual machine must be started for each test-case to run the java bytecode (as opposed to machine code produced by GCC). However, we were able to easily scale the system. GCC setup used three Judge0 virtual machines, of which only one was kept (two of them were too slow) and additional three were added to a total of four machines (typically with 4GB of RAM and 4 cores) with number of workers set in the range [10, 15]. Fig. 15 shows execution times for the whole semester by the time of day. In the worst case the execution times did not surpass 25 seconds. If we consider the same plot for a single day (Fig. 16) it is more obvious that execution times are skewed to the end of each group's term: when the time runs out the students are prompted by TAs to submit the exam and then the usage peaks occur and submissions queue up for execution thus yielding longer execution times. The box and whiskers plot in Fig. 17 shows that execution times over 5 seconds are in fact outliers and that most of the exams get carried out below five seconds while server B appears fastest, and server C slowest.

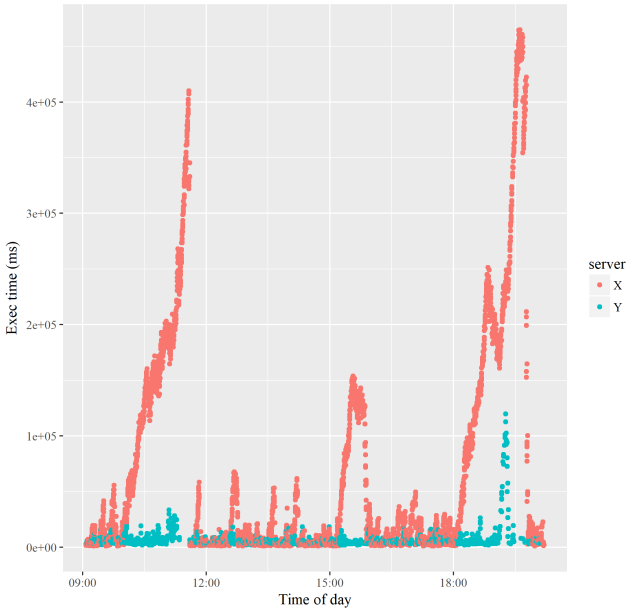

FIGURE 14. Performance issues on the March the 19rd (execution times over 400 seconds).

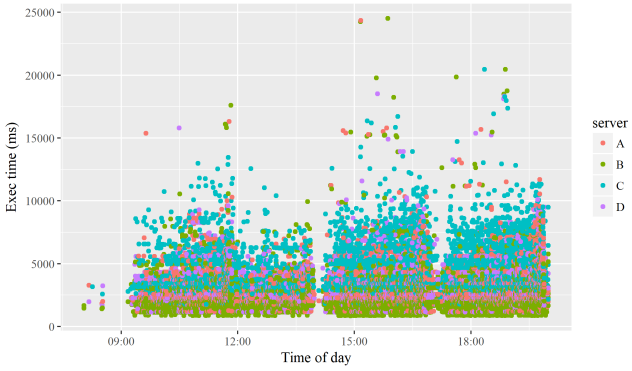

FIGURE 15. Code runner (Java) execution times by the time of day for whole semester.

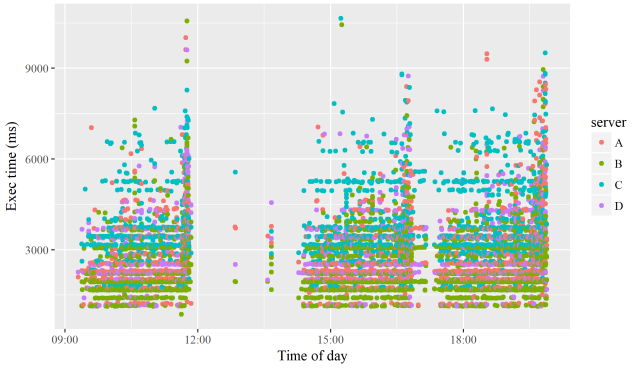

FIGURE 16. Code runner (Java) execution times by the time for 2019-05-16.

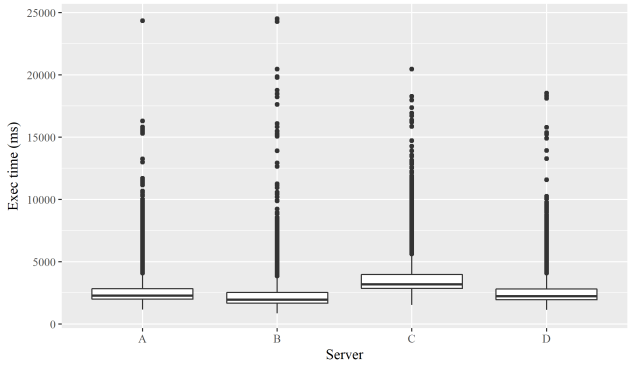

FIGURE 17. Execution times by server for the whole semester.

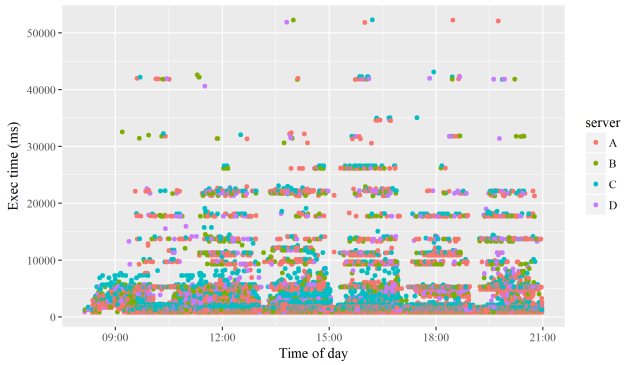

FIGURE 18. Distribution of execution by server for the winter semester(19/20)).

The average execution time of approximately 2.5 seconds is satisfactory. Again, one must have in mind that a typical Java task with e.g. 5 test-cases comprises of one *javac* compile, and five JVM runs where custom testing code is sometimes repeated tens of times. In the extreme, Judge0 code runners could be scaled so that every student ''has their own server'', however, in our case – four servers sufficed.

Subsequently, we have analyzed the execution times in the following semester (winter of 2019.) which features different set of courses (Programming in C and OOP) to confirm the scalability properties of our system. Figures 18 and 19 show the execution times for the entire winter semester.

Outliers in figures 18 and 19 are usually caused by students running infinite loops or highly unoptimized code. However, box and whiskers plot in Figure 19 shows that vast majority of tasks is executed in under 2 seconds, with overall mean (across all servers) of 1.61 seconds and median of 1.24 seconds. Execution times are even better than the semester before but this is because C programming language code (exams) dominates the winter semester, and significantly slower Java code dominates the spring semester.

V. DISCUSSION

In this section we provide comparison with related studies and software, comment on automated assessment advantages and challenges, and present future work.

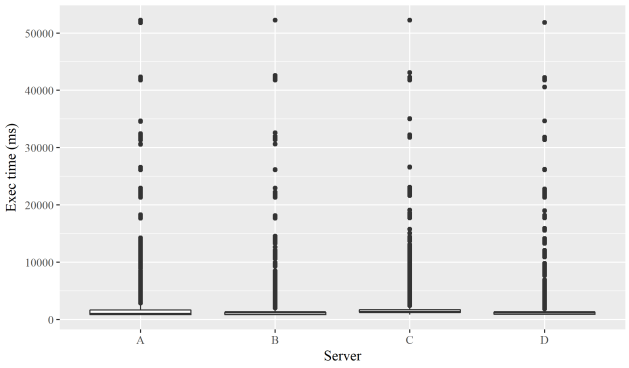

FIGURE 19. Execution by server for the winter semester(19/20).

A. COMPARISON WITH RELATED STUDIES

We have surveyed the existing APAS software and literature with regards to the features identified in Section III. The list of considered systems is mostly based on a recent systematic literature review [14]. Only systems supporting multiple languages within multiple paradigms (ACT Programming Tutor, Ceilidh, Checkpoint CourseMarker/CourseMaster, GAME (2, 2+), Marmoset, Pex4Fun, Submit!, Testovid/Svetovid, Moodle*, WebAssign, Web-CAT) were considered. This initial list was expanded with JACK as it is also a multi-paradigm system since it can cope with object oriented (Java, C++) languages and EPML/XML markup languages. Moodle is marked with an asterisk as it supports multiple languages only in combination with plugins (such as Virtual Programming Lab and CodeRunner), so it is evaluated with its plugins. To be able to compare the selected APASs in the context of the established features, relevant information must be available either through publications or through a publicly available version of software that can be tested freely. It turned out that for most of the 13 systems selected this is not the case, so the initial set, after thorough analysis, has been narrowed down to the five presented in Table 4. The reasons for excluding systems from further analysis are: (i) non-existent publication describing system features, (ii) the absence of a publicly available version of the software to test the system, and (iii) liveliness – an estimate whether the project has been abandoned judging by the publications and project website. Table 3 shows the initial list of systems which were filtered according to aforementioned criteria.

Columns F1-F9 in Table 4 correspond to the features presented in Section III while the last P/OS column indicates software availability (P stands for proprietary, and OS for open source solutions). The feature F2-System monitoring is omitted for it is typically implemented using 3rd party software and details are hard to find in both publications and demo versions of the system. Sign + indicates that the feature is supported in the corresponding system, regardless of the level. Furthermore, for every system there was at least one feature that we could not find data to resolve whether it was

TABLE 3. Initial list of 13 APASs with exclusion criteria.

APAS	Recent publication		Demo website URI	Incl. in comparison	Reason for exclusion
	Year	Ref			
ACT Programming Tutor	2001	[42]		No	(i) and (ii)
Ceilidh	1995	[43]		No	(i) and (ii)
Checkpoint	2015	[44]		Yes	
CourseMarker/CourseMaster	2009	[45]		No	(i) and (ii)
GAME (2, 2+)	2009	[46]		No	(i) and (ii)
JACK	2016	[47]	https://jack-demo.s3.uni-due.de/jack2/stud/Start.jsf	Yes	
Marmoset	2006	[48]	https://marmoset.cs.umd.edu/	No	(i) and (iii) only initial page is active, other links are broken
Pex4Func	2016	[49]	https://www.microsoft.com/en-us/research/project/pex4fun/	No	(i) and (iii) all usefull links are broken and project seems to be shut down
Submit!	2002	[50]	http://web.science.mq.edu.au/ apce/	No	(i) and (iii) broken link from the last publication, no other URIs
Testovid/Svetovid	2017	[51]		Yes	
Moodle*	2020		https://moodle.org/demo/ https://vpl.dis.ulpgc.es/	Yes	
WebAssign	2004	[52]	https://online-uebungssystem.fernuni-hagen.de/	No	(i) and (ii) Can't be tested online - limited to selected institutions
Web-CAT	2014	[53]	http://web-cat.org/	Yes	

TABLE 4. Feature comparison matrix for selected systems from Table 3. Columns: F1-Course administration; F3-Exam logging and monitoring; F4-Problem mitigation; F5-Analysis & visualisation; F6-Data import & export; F7-Content authoring; F8-Rich question types and grading facilities; F9-User friendly online testing; P/OS-proprietary or open source.

System	F1	F3	F4	F5	F6	F7	F8	F9	P/OS
Checkpoint	+	?	?	+	?	+	?	+	P
Testovid/Svetovid	+	+	?	?	?	+	+	+	P
Moodle*	+	+	?	+	+	+	+	+	OS
Web-CAT	+	+	+	?	+	+	+	+	OS
JACK	+	?	?	+	+	?	+	+	P
Edgar	+	+	+	+	+	+	+	+	OS

supported or not, so we used "?" (meaning "No info") to denote such cases.

In conclusion, despite the vast number of publications in the field and a large body of potential users, the choice of APAS software for a university looking to introduce automated assessment is very limited. It boils down to just few systems that are still lacking in certain areas.

B. APAS ADVANTAGES AND CHALLENGES

Use of an APAS brings many advantages. A study in [54] showed that human graders can significantly differ when scoring partially correct solutions. Automatic evaluations based on well-defined criteria can fix human graders' errors and even increase students' score [55]. Multi-paradigm support in Edgar has enabled use of APAS in several courses allowing students to have higher number of assignments, which we find essential for mastering programming concepts. On the other hand, APASs also have open issues, mostly related to feedback appropriateness, submission policies and enabling unwanted students' behaviour. Opinions about unlimited submissions vary. Pieterse [56] states that unlimited submission is essential if assessment is done for formative reasons. Auvinen [57] makes a point on undesired and harmful habits of exploiting submission feedbacks to do trial-and-error strategy. In order to prevent students from misusing the system by gauging test cases, Edgar enables creation of test cases with random input data. However, that might not dissuade the students that Karavirta *et al.* call *iterators* [58] – those that persistently iterate in a trial-and-error fashion. To address such issues, Edgar provides a support to limit or penalize repeated submissions, but it is left to the teachers to define whether they want to apply it or not.

In [59] it is stated that binary feedback can motivate students to cheat and Edgar addresses this problem by allowing teachers to create granular scoring with several test cases using various grading models and penalty percentages. A study by Falkner *et al.* [60] shows that granularity can increase effectiveness.

Hao *et al.* [61] define and test the effects of three types of feedback: *"What's wrong"* – feedback in which only test case pass or fail is revealed, *"Gap"* that displays differences between actual and expected output, and *"Hint"* showing hint how to fix the problems. They conclude that *Gap* feedback type increase student' performance compared to *What's wrong feedback*, but they do not find significant improvement using *Hint* over *Gap* feedback type. The effects of submission policy and feedback verbosity was tested in [62] and the authors conclude that they "cannot declare any of the feedback styles clearly superior to the other".

Edgar does not enforce a specific feedback style and directly supports *What's wrong* and *Gap feedback*. How to use test case descriptions and verbose output is described in Section III-I showing how simple hints from the teacher can be shown to a student. A more sophisticated hint feedback is

language and paradigm dependent and could be implemented as an additional plugin. For instance, previous submissions could be analysed in order to categorize errors aiming to provide hint e.g. for specific error [63], error category [64] or hint based on similarity of student's program to elements containing teaching comments and hint [65]. Edgar does not yet support such advanced scenarios, but it provides facilities to show various types of feedback including the arbitrary code that will generate feedback based on the student's submissions. Feedback generation is an interesting topic in this area where automated systems still fall short of human generated feedback [64].

C. FUTURE WORK

Edgar is being actively developed past the basic APAS features to include certain LMS features. APASs provide convenient foundation for learning programming and Edgar already features tutorials and adaptive exercises. Tutorials intertwine content and code and enable students to e.g. execute SQL code and receive feedback as they progress through the tutorial steps. Adaptive exercises categorize questions into three difficulty categories and use configurable models to adaptively present questions to the students based on students' performance. In both APAS and LMS categories, future development includes additional evaluation features like static code analysis and code style assessment, additional learning scenarios and assisted coding (rich feedback), data visualization and configurable plagiarism detection since standard methods yield too many false positives for short coding assignments often used in our courses. Plagiarism detection should also consider the time when the code is written and reconstruct the dissemination path of the plagiarised code. In addition to that, gamification, peer assessment and competitive programming modules are already in development. From a more technical standpoint, we are planning to implement real-time logging (log aggregation) and to dockerize all components to facilitate the management of the entire ecosystem.

D. PRACTICAL CONSIDERATIONS

Edgar is released under the MIT licence and is free for use for all interested parties. Edgar's source code and installation instructions can be found at Edgar's GitLab site [39]. The site contains link to demo showcasing student's perspective, e.g. it is possible to write an exam with multiple-choice questions, SQL, C and Java questions.

VI. CONCLUSION

In computer science education field, there is an ever-growing demand for automated assessment of programming assignments in different programming languages and technologies. In this paper we survey the software and literature to define nine feature categories that qualify an APAS as comprehensive – such that would support all key stages in the assessment process. Our objective was to introduce APAS to large university level courses and having found no systems that meet these requirements while being freely available (Moodle being the closest) we decided to develop our own. Here, we present Edgar - a novel, state-of-the-art, automated program assessment system capable of assessing programming assignments written in arbitrary programming language (SQL, Java, C, C#, etc.) as well as multi-correct multiple-choice questions. It features a flexible exam and question definition model, rich exam logging and monitoring facilities and data analysis and visualization features. Edgar is built using modern, open-source, cross-platform technologies with modular architecture so that it can be customized and scaled at will. We present its architecture and scalability properties using real-world data. Unlike most other systems that target a particular course, language, use case, or aspect of programming assessment, Edgar's contribution lies in the generic, comprehensive approach to the APAS development. With its rich feature set, Edgar surpasses, to the best of our knowledge, any other publicly available system for testing programming assignments. As such, we believe it will be of great interest to a wide audience of institutions practicing computer science education. Before this public release, Edgar has been in use at University of Zagreb Faculty of Electrical Engineering and Computing for three years and is going to be used in the future to an increasing extent.

APPENDIX
PROGRAMMING QUESTIONS ASSESSMENT EXAMPLES

Programming questions, regardless of type (SQL, Java, C, ...), are defined with three code snippets: obligatory *source* and optional *prefix* and *suffix*, and (except for the SQL) an arbitrary number of test-cases with penalty percentages. Students provide code corresponding to the source and the complete program is constructed by concatenating the *prefix*, *source* and *suffix*. This way, arbitrary code snippets can be tested (typically functions), but teachers can creatively use these code sections to support various scenarios; e.g. in Object oriented programming course teachers are testing the class structure (existence, names and visibility of methods and members) using reflection and use test-case inputs to select set of unit tests contained inside *suffix*! For SQL questions the use of *prefix*+source+suffix scheme allows to test not only `SELECT` statements but also record updates, deletions, and even DML expressions like `ALTER TABLE` or `CREATE INDEX` by querying the system catalogue in *suffix* part. Execution of the assembled code is delegated to an external runner (as described in Section IV) to be run in an isolated environment. Results are then returned to Edgar to be assessed and graded. Edgar features different data structures for different programming languages (Table 2) with corresponding comparison parameters used to assess the solutions. Here we present SQL and C assessments. SQL assessment is illustrated with the example in Table 5 using comparison parameters shown in Table 6. In this example, `INSERT` statements in the *prefix* are used to temporarily change the database's content with rows that were not visible to student during the exam. This is typically

TABLE 5. SQL DELETE statement question definition and student's answer.

Question: Delete all teachers with a letter 'w' in the last name.		Student's code

prefix	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (100, 'Ann', 'Aww');`
	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (101, 'Mary', 'Johnson');`	

source	`DELETE FROM teacher WHERE ln LIKE '%w%';`	`DELETE FROM teacher` `WHERE ln LIKE 'w';`
suffix	`SELECT * FROM teacher;`	
sql_alt_presentation_query	`SELECT * FROM teacher WHERE ln LIKE '%w%';`	

TABLE 6. Evaluating correctness for the example in Table 5.

Assembled code to execute:		Compare and present	Correctness

Teacher	`BEGIN WORK;`
	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (100, 'Ann', 'Aww');`
	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (101, 'Mary', 'Johnson');`
	`DELETE FROM teacher WHERE ln LIKE '%w%';`
	`SELECT * FROM teacher; -- get recordset RST`
	`ROLLBACK WORK;`
		Compare RST and RSS using options:
		Check tuple order
		1. STRICT: both column names and order must match.
		2. SQL: column order must match (column names ignored).
		3. RELALG: column names must match (column order ignored).
		4. PERMISSIVE: try 3 (to match by names); if not - try 2 (use column order).
		Present RSP to student.
			Binary: 0% or 100% whether `RST == RSS`

Student	`BEGIN WORK;`
	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (100, 'Ann', 'Aww');`
	`INSERT INTO teacher (id, fn, ln)`
	`VALUES (101, 'Mary', 'Johnson');`
	`DELETE FROM teacher WHERE ln LIKE 'w';`
	`SELECT * FROM teacher; -- get recordset RSS`
	`SELECT * FROM teacher`
	`WHERE ln LIKE '%w%'; -- get recordset RSP`
	`ROLLBACK WORK;`		

used to prevent students from hard-coding the solution (e.g. `SELECT 1, 'first row' UNION SELECT 2, 'second row' ...`). When the complete SQL code is assembled for the teacher and the student (Table 6), record sets presenting correct solution (denoted RST - *RecordSetTeacher*) and student's solution (denoted RSS - *RecordSetStudent*) are acquired and subsequently compared. Since in this scenario, student's solution is erroneous, RSS will contain both rows inserted with statements from the *prefix* part, while RST will not contain row with id = 100. The student will be presented with a RSP (*RecordSetPresentation*) rather than RST as it will be easier to spot and correct error based on a smaller data set. Note that unlike code questions or multiple-choice questions the correctness of SQL questions is binary – 0% or 100%.

Examples of code questions in C language are shown in Table 7 and Table 8. Similar approach is used for other programming languages like Java, Python, etc. Assembled student's code is compiled (once) and executed (three times in this example as we have three test-cases) with corresponding test-case data redirected to the standard input. Output of the program is compared with the expected output and, in the case of mismatch, the penalty percentage is applied. Comparison is done either by:

- comparing fixed expected output with student's actual output (test-cases 1 and 2 in Table 8)
- comparing student's program output with the teacher's program output (test-case 3 in Table 8). This approach is used when tests-cases with randomly generated input values are used (e.g. Edgar can generate range of numbers, letters, ...). In this case, *source* part in question definition is obligatory, though it is always a good practice to provide the source part i.e. the solution.

During comparison of outputs, depending on selected options, leading and trailing whitespaces can be trimmed, case can be ignored, or even regular expression matching can be employed. Table 8 shows how correctness is calculated in this example. Shaded cells in a row are compared for equality. Penalty percentages in this example are 100% for each test, as each test failure suggest that student's solution is completely wrong and should be treated as incorrect. Often, tests consist of several test with 100% penalty and some test with minor penalty (e.g. 10-30%) for minor mistakes like improperly handling boundary conditions.

TABLE 7. Code question definition and student's answer.

Question	Using C programming language, write a function `int add(int, int)` that adds two integers.					**Student's answer**
prefix	`#include <stdio.h>`					

source	`int add(int a, int b) {`
	`  return a + b;`
	`}`
						`int add(int a, int b) {`
						`  return a * b; //error here`
						`}`

suffix	`int main (void) {`
	`  int a, b;`
	`  scanf("%d", &a);`
	`  scanf("%d", &b);`
	`  printf("%d", add(a, b));`
	`  return 0;`
	`}`					

Test-cases(3)	**ordinal**	**type**	**PP**	**stdin**	**expected stdout**	
	#1	fixed	100%	0 0	0	
	#2	fixed	100%	0 0	0	
	#3	random	100%	*randomly generated according to certain rules*	*matches teacher's code output*	

TABLE 8. Evaluating correctness for the example in Table 7. Shaded cells within a row are compared.

Test-cases	**Test stdin**	**Test std-out**	**Correct code stdout**	**Student's code stdout**	**PP**	**To deduct**
#1	0 0	0	- *(not executed)*	0	100%	-
#2	0 1	1	- *(not executed)*	0	100%	100%
#3	12 10	-	22	120	100%	100%
					TOTAL PP	200%
					CORRECTNESS	0%

JSON questions are evaluated in a similar way as SQL questions, also with binary outcome. When comparing objects that are results of teachers and students code, JSON comparison can do shallow (deep) and strict comparison of objects.

REFERENCES

[1] J. Hollingsworth, "Automatic graders for programming classes," *Commun. ACM*, vol. 3, no. 10, pp. 528–529, Oct. 1960, doi: 10.1145/367415.367422.

[2] X. Bai, A. Ola, S. Akkaladevi, and Y. Cui, "Enhancing the learning process in programming courses through an automated feedback and assignment management system," *Issues Inf. Syst.*, vol. 17, no. 3, pp. 165–175, 2016. [Online]. Available: http://www.iacis.org/iis/2016/3_iis_2016_165-175.pdf

[3] G. Rößling, M. Joy, A. Moreno, A. Radenski, L. Malmi, A. Kerren, T. Naps, R. J. Ross, M. Clancy, A. Korhonen, R. Oechsle, and J. Á. V. Iturbide, "Enhancing learning management systems to better support computer science education," *ACM SIGCSE Bull.*, vol. 40, no. 4, pp. 142–166, Nov. 2008, doi: 10.1145/1473195.1473239.

[4] N. Singer. (Jan. 2019). *The Hard Part of Computer Science? Getting Into Class*. [Online]. Available: https://www.nytimes.com/2019/01/24/technology/computer-science-courses-college.html

[5] I. Mekterović and L. Brkić, "Setting up automated programming assessment system for higher education database course," *Int. J. Edu. Learn. Syst.*, vol. 2, pp. 287–294, Jan. 2017.

[6] J. P. Leal and F. Silva, "Mooshak: A Web-based multi-site programming contest system," *Softw., Pract. Exper.*, vol. 33, no. 6, pp. 567–581, 2003, doi: 10.1002/spe.522.

[7] M. Szabó and K. Nehéz, "Grading java code submissions in MeMOOC," in *Proc. 32nd MultiScience MicroCAD Int. Sci. Conf.*, 2018. [Online]. Available: https://www.uni-miskolc.hu/~microcad/publikaciok/2018/c2/C2_SzaboMartin.pdf, doi: 10.26649/musci.2018.026.

[8] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Comput. Sci. Edu.*, vol. 15, no. 2, pp. 83–102, Jun. 2005, doi: 10.1080/08993400500150747.

[9] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, vol. 5, no. 3, pp. 1–4, 2005, doi: 10.1145/1163405.1163409.

[10] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proc. 10th Koli Calling Int. Conf. Comput. Edu. Res. (Koli Calling)*, 2010, pp. 86–93, doi: 10.1145/1930464.1930480.

[11] J. L. F. Aleman, "Automated assessment in a programming tools course," *IEEE Trans. Educ.*, vol. 54, no. 4, pp. 576–581, Nov. 2011, doi: 10.1109/te.2010.2098442.

[12] J. C. Caiza and J. M. Del Alamo, "Programming assignments automatic grading: Review of tools and implementations," in *Proc. 7th Int. Technol., Edu. Develop. Conf.*, 2013, pp. 5691–5700. [Online]. Available: http://oa.upm.es/25765/1/INVE_MEM_2013_160449.pdf

[13] D. M. Souza, K. R. Felizardo, and E. F. Barbosa, "A systematic literature review of assessment tools for programming assignments," in *Proc. IEEE 29th Int. Conf. Softw. Eng. Edu. Training (CSEET)*, Apr. 2016, pp. 147–156, doi: 10.1109/cseet.2016.48.

[14] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Edu.*, vol. 19, no. 1, pp. 1–43, Jan. 2019, doi: 10.1145/3231711.

[15] J. Liebenberg and V. Pieterse, "Investigating the feasibility of automatic assessment of programming tasks," *Inf. Technol. Educ. Innov. Pract.*, vol. 17, pp. 201–223, 2018. [Online]. Available: http://www.jite.org/documents/Vol17/JITEv17IIPp201-223Liebenberg4853.pdf, doi: 10.28945/4150.

[16] N. Stanger, "Semi-automated assessment of SQL schemas via database unit testing," in *Proc. 26th Int. Conf. Comput. Edu. (ICCE), Asia–Pacific Soc. Comput. Educ. (APSCE)*, J. C. Yang, M. Chang, L.-H. Wong, and M. M. T. Rodrigo, Eds., 2018, pp. 237–246.

[17] M. Luck and M. Joy, "Automatic submission in an evolutionary approach to computer science teaching," *Comput. Edu.*, vol. 25, no. 3, pp. 105–111, Nov. 1995, doi: 10.1016/0360-1315(95)00056-9.

[18] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *J. Educ. Resour. Comput.*, vol. 5, no. 3, pp. 2–es, Sep. 2005, doi: 10.1145/1163405.1163407.

[19] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, "Ability-training-oriented automated assessment in introductory programming course," *Comput. Edu.*, vol. 56, no. 1, pp. 220–226, Jan. 2011, doi: 10.1016/j.compedu.2010.08.003.

[20] C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas, "The CourseMarker CBA system: Improvements over Ceilidh," *Edu. Inf. Technol.*, vol. 8, no. 3, pp. 287–304, 2003, doi: 10.1023/A:1026364126982.
[21] M. Amelung, K. Krieger, and D. Rösner, "E-assessment as a service," *IEEE Trans. Learn. Technol.*, vol. 4, no. 2, pp. 162–174, Jun. 2011, doi: 10.1109/TLT.2010.24.
[22] (Dec. 2019). *Web-CAT: What is Web-CAT?* Accessed: May 1, 2020. [Online]. Available: https://web-cat.org/projects/Web-CAT
[23] M. Striewe, "Der grader JACK," in *Automatisierte Bewertung der Programmierausbildung*, O. J. Bott, P. Fricke, U. Priss, and M. Striewe, Eds. Bradford, U.K.: The Science and Information (SAI) Organization, 2017, pp. 143–159. [Online]. Available: https://thesai.org/Publications/ViewPaper?Volume=10&Issue=3&Code=ijacsa&SerialNo=28
[24] H. Aldriye, A. Alkhalaf, and M. Alkhalaf, "Automated grading systems for programming assignments: A literature review," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 3, pp. 215–221, 2019, doi: 10.14569/ijacsa.2019.0100328.
[25] R. Lobb and J. Harlow, "Coderunner," *ACM Inroads*, vol. 7, no. 1, pp. 47–51, Feb. 2016, doi: 10.1145/2810041.
[26] A. Abelló, X. Burgués, M. J. Casany, C. Martín, C. Quer, M. E. Rodríguez, O. Romero, and T. Urpí, "A software tool for E-assessment of relational database skills," *Int. J. Eng. Edu.*, vol. 32, vol. 3, pp. 1289–1312, 2016.
[27] R. E. Noonan, "The back end of a grading system," in *Proc. 37th SIGCSE Tech. Symp. Comput. Sci. Edu. (SIGCSE)*. Houston, TX, USA: ACM, 2006, pp. 56–60, doi: 10.1145/1121341.1121360.
[28] T. H. Wang, K. H. Wang, W. L. Wang, S. C. Huang, and S. Y. Chen, "Web-based assessment and test analyses (WATA) system: Development and evaluation," *J. Comput. Assist. Learn.*, vol. 20, no. 1, pp. 59–71, Feb. 2004, doi: 10.1111/j.1365-2729.2004.00066.x.
[29] (Jun. 2019). *Zabbix: The Enterprise-Class Open Source Network Monitoring Solution*. Accessed: May 1, 2020. [Online]. Available: https://www.zabbix.com/
[30] G. Rasch, *Probabilistic Models for Some Intelligence and Attainment Tests*. Copenhagen, Denmark: Danmarks Paedagogiske Institute, 1960.
[31] F. M. Lord and M. R. Novick, *Statistical Theories of Mental Test Scores*. Charlotte, NC, USA: Information Age Pub, 2008.
[32] (Jun. 2019). *MathJax|Beautiful Math in All Browsers*. Accessed: May 1, 2020. [Online]. Available: https://www.mathjax.org/
[33] (Jun. 2019). *Kahoot!|Learning Games|Make Learning Awesome!* Accessed: May 1, 2020. [Online]. Available: https://kahoot.com/
[34] M. Jagar, J. Petrović, and P. Pale, "AuResS: The audience response system," in *Proc. ELMAR*, Sep. 2012, pp. 171–174.
[35] (Jun. 2019). *Wooclap—An Interactive Platform That Makes Learning Awesome*. Accessed: May 1, 2020. [Online]. Available: https://www.wooclap.com/
[36] OASIS. (Sep. 2019). *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. Accessed: May 1, 2020. [Online]. Available: https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf
[37] IETF OAuth Working Group. (Jun. 2019). *OAuth 2.0 Specification*. Accessed: May 1, 2020. [Online]. Available: https://oauth.net/2/
[38] (Jun. 2019). *NGINX|High Performance Load Balancer, Web Server, and Reverse Proxy*. Accessed: May 1, 2020. [Online]. Available: https://nginx.org/
[39] I. Mekterović. (Jun. 2019). *Edgar Project Group*. Accessed: May 1, 2020. [Online]. Available: https://gitlab.com/edgar-group
[40] H. Z. Došilović. (Jun. 2019). *Judge0*. Accessed: May 1, 2020. [Online]. Available: https://www.judge0.com/
[41] M. Mareš and B. Blackham, "A New Contest Sandbox," *Olympiads Informat.*, vol. 6, pp. 100–109, Jan. 2012. [Online]. Available: https://mj.ucw.cz/papers/isolate.pdf
[42] A. T. Corbett and J. R. Anderson, "Locus of feedback control in computer-based tutoring," in *Proc. SIGCHI Conf. Human Factors Comput. Syst. (CHI)*. New York, New York, USA: ACM, 2001, pp. 245–252, doi: 10.1145/365024.365111.
[43] S. D. Benford, E. K. Burke, E. Foxley, and C. A. Higgins, "The ceilidh system for the automatic grading of students on programming courses," in *Proc. 33rd Annu. Southeast Regional Conf. (ACM-SE)*, 1995, pp. 176–182, doi: 10.1145/1122018.1122050.
[44] J. English and T. English, "Experiences of using automated assessment in computer science courses," *J. Inf. Technol. Educ., Innov. Pract.*, vol. 14, pp. 237–254, Oct. 2015, doi: 10.28945/2304.
[45] C. A. Higgins, B. Bligh, P. Symeonidis, and A. Tsintsifas, "Authoring diagram-based CBA with CourseMarker," *Comput. Edu.*, vol. 52, no. 4, pp. 749–761, May 2009, doi: 10.1016/j.compedu.2008.11.019.
[46] R. Matloobi, M. Blumenstein, and S. Green, "Extensions to generic automated marking environment: Game-2+," in *Proc. Interact. Comput. Aided Learn. Conf. (ICL)*, 2009, vol. 1, no. 8, pp. 1069–1076.
[47] M. Striewe, "An architecture for modular grading and feedback generation for complex exercises," *Sci. Comput. Program.*, vol. 129, pp. 35–47, Nov. 2016, doi: 10.1016/j.scico.2016.02.009.
[48] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses," *ACM SIGCSE Bull.*, vol. 38, no. 3, pp. 13–17, Jun. 2006, doi: 10.1145/1140123.1140131.
[49] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *Proc. 38th Int. Conf. Softw. Eng. Companion (ICSE)*. New York, NY, USA: ACM, 2016, pp. 501–510, doi: 10.1145/2889160.2889204.
[50] Y. Pisan, D. Richards, A. Sloane, H. Koncek, and S. Mitchell, "Submit! A Web-based system for automatic program critiquing," in *Proc. 5th Australas. Conf. Comput. Edu. (ACE)*, vol. 20, 2003, pp. 59–68.
[51] I. Pribela and D. Pracner, "A temporal file system for student's assignments in the system svetovid," in *Proc. 6th Workshop Softw. Qual., Anal., Monitor., Improvement, Appl. (SQAMIA)*, Z. Budimac, Ed., Belgrade, Serbia, 2017, pp. 12:1–12:08.
[52] C. Beierle, M. Kulas, and M. Widera, "Partial specifications of program properties," in *Proc. 1st Int. Workshop Teach. Log. Program. (TeachLP)*, no. 12. Linköping, Sweden: Linköping Univ., 2004, pp. 18–34.
[53] S. H. Edwards, "Work-in-progress: Program grading and feedback generation with Web-CAT," in *Proc. 1st ACM Conf. Learn. Scale Conf. (LS)*. New York, NY, USA: ACM, 2014, pp. 215–216, doi: 10.1145/2556325.2567888.
[54] I. Albluwi, "A closer look at the differences between graders in introductory computer science exams," *IEEE Trans. Educ.*, vol. 61, no. 3, pp. 253–260, Aug. 2018, doi: 10.1109/te.2018.2805706.
[55] D. Insa and J. Silva, "Automatic assessment of java code," *Comput. Lang., Syst. Struct.*, vol. 53, pp. 59–72, Sep. 2018, doi: 10.1016/j.cl.2018.01.004.
[56] V. Pieterse, "Automated assessment of programming assignments," in *Proc. 3rd Comput. Sci. Educ. Res. Conf. Comput. Sci. Educ. Res.*, vol. 3, Apr. 2013, pp. 45–56, doi: 10.1145/1559755.1559763.
[57] T. Auvinen, "Harmful study habits in online learning environments with automatic assessment," in *Proc. Int. Conf. Learn. Teach. Comput. Eng.*, Apr. 2015, pp. 50–57, doi: 10.1109/latice.2015.31.
[58] V. Karavirta, A. Korhonen, and L. Malmi, "On the use of resubmissions in automatic assessment systems," *Comput. Sci. Edu.*, vol. 16, no. 3, pp. 229–240, Feb. 2007, doi: 10.1080/08993400600912426.
[59] A. Kyrilov and D. C. Noelle, "Binary instant feedback on programming exercises can reduce student engagement and promote cheating," in *Proc. 15th Koli Calling Conf. Comput. Edu. Res. (Koli Calling)*, vols. 19–22, 2015, pp. 122–126, doi: 10.1145/2828959.2828968.
[60] N. Falkner, R. Vivian, D. Piper, and K. Falkner, "Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units," in *Proc. 45th ACM Tech. Symp. Comput. Sci. Edu. (SIGCSE)*, May 2014, pp. 9–14, doi: 10.1145/2538862.2538896.
[61] Q. Hao, J. P. Wilson, C. Ottaway, N. Iriumi, K. Arakawa, and D. H. Smith, "Investigating the essential of meaningful automated formative feedback for programming assignments," in *Proc. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VL/HCC)*, Oct. 2019, pp. 151–155, doi: 10.1109/vlhcc.2019.8818922.
[62] A. Annamaa, R. Suviste, and V. Vene, "Comparing different styles of automated feedback for programming exercises," in *Proc. ACM Int. Conf.*, 2017, pp. 183–184, doi: 10.1145/3141880.3141909.
[63] D. Insa and J. Silva, "Computer assisted self-assessment of programming code: A report on university students experience and opinion," in *Proc. 15th Int. Conf. Inf. Technol. Higher Edu. Training (ITHET)*, Sep. 2016, pp. 1–3, doi: 10.1109/ithet.2016.7760727.
[64] A. Kyrilov and D. Noelle, "Do students need detailed feedback on programming exercises and can automated assessment systems provide it?" *J. Comput. Sci. Colleges*, vol. 31, no. 4, pp. 115–121, 2016.
[65] P. Silva, E. Costa, and J. R. de Araújo, "An adaptive approach to provide feedback for students in programming problem solving," in *Intelligent Tutoring Systems* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11528. Cham, Switzerland: Springer, Jun. 2019, pp. 14–23. [Online]. Available: [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-030-22244-4_3, doi: 10.1007/978-3-030-22244-4_3.

IGOR MEKTEROVIĆ received the Ph.D. degree from the University of Zagreb, in 2008.

He is an Associate Professor with the Faculty of Electrical Engineering and Computing, Department of Applied Computing, University of Zagreb. His research interests are in the areas of computer science education, databases, data warehouses, Web development, and bioinformatics. He is a member of the Croatian Centre of Research Excellence for Data Science.

LJILJANA BRKIĆ received the M.Sc. and Ph.D. degrees in computer science, in 2004 and 2011, respectively.

She is an Associate Professor with the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb. She is currently a Main Coordinator of the student mobility with the Faculty of Electrical Engineering and Computing. Her research interests include data warehouses, business intelligence, information systems, and programming paradigms.

BORIS MILAŠINOVIĆ (Member, IEEE) received the bachelor's degree from the Faculty of Science, Department of Mathematics, University of Zagreb, in 2001, and the M.Sc. and Ph.D. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2006 and 2010, respectively.

He is an Associate Professor with the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb. His main research interests include software development methodologies and workflow management. He has been a member of the Editorial Board of *Computer Science and Information Systems* journal, since 2018, and a program committee member of several international conferences.

MIRTA BARANOVIĆ (Member, IEEE) is a Full Professor in computer science with the Faculty of Electrical Engineering and Computing, University of Zagreb. She has published more than 80 articles in journals and conference proceedings. Her research interests include databases, information systems, data warehouses, data lakes, semantic Web, geospatial databases, and data streams.

Dr. Baranović served as the Honorary President of the European Federation of National Maintenance Societies (EFNMS). She also served as a member of the Board of Directors of the EFNMS. She has been a member of the IEEE Computer Society and the IEEE Women in Engineering, since 2004. She is currently a member of the Croatian Centre of Research Excellence for Data Science. She has received several awards, including the Silver Medal Josip Lončar for an outstanding doctoral thesis, in 1997, the IBM International Informix Users Group Award for exceptional service and dedication to the Informix Community, in 2012, and the Golden Medal Josip Lončar for significant improvement of teaching, connecting with the industry, and contributing to the international exchange programmes, in 2014, and she was honoured with the Award of the IEEE Croatia Section for outstanding contribution to engineering education, in 2015.