Automatic Evaluation of Student Software Solutions in a Virtualized Environment

M. Fabijanić, G. Đambić, B. Skračić, and M. Kolarić
Algebra University College/Software Engineering, Zagreb, Croatia
mario.fabijanic@algebra.hr

***Abstract* – Universities are using Automated Programming Assessment Systems (APAS) to minimize problems that emerge by manually managing student software solutions: subjectivity, inefficiency in case of many solutions, and lack of fast and rich feedback for the student. Lately, handling code executions in a secure way has become a requirement. Researchers addressed this topic and proposed generic security models: user-level restrictions, process-level restrictions, and virtualization. In the case of virtualization, commonly suggested solutions are virtual machines and containers. With all the benefits of virtual machines, but less resource-demanding, containers are becoming widely used last years. This paper proposes a newly developed assessment model and an information system and proposes a solution for the automatic evaluation of student software solutions in a virtualized environment. It analyzes the key parts of the student solution source codes and commands against a combination of the test case input data, using the pattern-finding method. This method was used in the context of finding the required lexical structures. The proposed system is used during the exam evaluation of the Programming course at the College of Algebra, and the relational database courses, for evaluating SQL solutions in combination with the previously developed system at the College of Algebra.**

Keywords – APAS; computer science education; educational technology; automatic evaluation; assessment; virtualized environment; partial marks; configurable assessment process

I. Introduction

With the increase in the popularity of computer science, as well as university and professional studies in the field of computing, the enrollment trend at the corresponding higher education institutions has also increased. In correlation with this positive trend, the number of students who take exams and other forms of testing skills and knowledge of programming each semester has also increased. Teachers, teaching assistants, and other staff members spend a lot of time manually and repetitively correcting student solutions due to the volume of tests and the number of students that take them.

However, the problem is not new, the first solution to this problem was created about 60 years ago [1]. In the meantime, the requirements changed, and the solutions for automatic correction of the program code met the needs.

This paper describes a software solution that enables students to submit solutions to exam tasks, reduces the time consumption of exam correctors, and minimizes the possibility of human error during evaluation and scoring. For security reasons, reading and executing submitted user files should be considered untrusted and executed within a virtualized environment. The goal is to isolate the processes that occur when compiling and executing the code from the environment of the operating system that serves the service itself. Functional and non-functional requirements, architecture, and system domain entities are defined, as well as an analysis of the functionality of each system component. In the end, an example of the application of the proposed solution is shown.

The implemented service satisfies two groups of requirements: code execution requirements and solution evaluation requirements.

A. *Code Execution Requirements*

Code execution is a prerequisite for a successful solution evaluation process. The expected system capabilities are:

- translation of the original program code
- execution of compiled program code while passing input data
- stopping the process after a defined time limit.

The main purpose is to translate and execute the source code submitted as a solution to a specific task. The source code is translated using a compiler, and only if the code can be translated into an executable, the program moves to the execution phase. After executing, it is necessary to read the stream of standard output (standard output) and the stream of standard error (standard error). The compilation or execution of the program must be stopped after a certain time limit to save processor and memory resources and to ensure the smooth operation of the system.

Non-functional requirements are not related to examples of use but to system characteristics, and in the context of code execution they are the following [2], [3]:

- error tolerance
- compilation and execution within a protected environment
- agnosticism towards programming languages
- ease of setup and configuration.

When compiling the source code and executing the program, numerous errors are possible, and some of the causes of these errors [2], [3] are the impossibility of compiling the source code, infinite compilation time, exceptions, and infinite loops. Source code with syntax error cannot be compiled, therefore the program ends with the transfer of the cause of the error. If it is an infinite translation time, the translator does not report an error, but the resources of the environment in which the program is translated are increasingly occupied.

During execution, the program may generate an exception (for example, accessing a non-existent array index) that is passed to the standard error stream, which needs to be read and the error passed. Like the case of infinite compile time, continuous execution of a program caused by faulty logic within the code does not raise an error but consumes resources of the environment in which the program is executed.

Due to the variety of software transfer formats, the submitted source code received by the system for security reasons should be considered untrusted, malicious, and a potential cause of the errors. Moreover, the code execution unit could be used within different exam correction systems, so it is important to separate the process of compiling and executing the program from the specifics of the programming language and related tools. This condition will be satisfied by the correct implementation of a protected virtualized environment.

II. Methodology

Evaluation of student program solutions consists of two separate tasks: the evaluation of execution results based on defined test cases, and static analysis of the source code, i.e., checking the existence of certain lexical structures.

Different methods can be applied during the automatic evaluation [4]: unit tests, sketching synthesis and error statistical modeling, peer-to-peer feedback, test cases with random input data, and pattern matching.

A. Test cases methods

To develop a software solution evaluation service, a combination of the test case method with random input data and the pattern-finding method was chosen [5]. Each test case contains predefined input and expected output data. The pattern-finding method was used in the context of finding the required lexical structures. The examiner defines the test-ordered pair of input-output data in the format of the standard input that will be used during the execution of the program and the standard output that will be compared with the actual output of the executed program. Based on the coincidence of these two data, the number of points will be awarded.

B. Static analysis

Static analysis of the source code is a check of syntactic rules. Each rule is defined as the presence of specific lexical constructions and the number of occurrences within the source code. For each task, the examiner defines a set of rules related to a specific request. Any non-compliance with the rules is punished with a defined point penalty, which is deducted from the final sum of points for that task.

C. Service requirements for automated evaluation of software solutions

There are several basic requirements of the service for the automated evaluation of software solutions:

- The examiner defines the test tasks: question text, name, test cases, grading rules, and the test taker receives feedback on the number of points earned and the reason for possibly deducted points.
- The examiner defines rules and test cases for each task.
- The examinee transfers his solution to the evaluation service in the form of a compressed file. The service creates a directory structure that meets the project pattern. During the exam, the examinee can upload his current solution several times to check the evaluation results of the currently solved tasks. At the end of the exam, the user confirms that his solution application is final and receives a complete detailed analysis of the scoring of each task.

D. Proposed solution architecture

The software solution includes a server web service that communicates with a database and a protected virtualized environment. The client communicates with the server by sending HTTP (Hypertext Transfer Protocol) requests according to the defined routes of the server web service.

REST (Representational State Transfer) is a program architecture model that contains a certain set of restrictions [6]. A system that implements REST is called RESTful and is based on the client-server concept. The RESTful system does not store states (stateless) and is characterized by uniform accessibility to every resource through HTTP requests. The backend service passes data from the request component to the evaluation component for code execution. The code execution component, communicating with the virtualized environment, translates the source code and if the translation is successful, executes the program and monitors the success of the implementation of these two phases. The server service stores all assessment requests in a relational database, which also stores data on defined exams, tasks, test cases, and rules.

E. Virtualized environment

There are more tools and techniques available to create a protected isolated environment. The most popular techniques are creating virtual machines and creating containers. The main difference between them is the hypervisor used by virtual computers. The hypervisor manages virtualized resources and guest operating systems, thus creating virtual machines with running complete operating systems and associated processes. Unlike virtual machines, starting a container does not require the existence of a hypervisor but a container engine installed as part of the host operating system.

Using containers with the necessary libraries within the container [7] makes it possible to start application processes independently of the host operating system. Starting containers is less time-consuming than starting virtual machines [6], [7] thanks to the lack of a hypervisor and the absence of starting the entire virtual operating system.

To ensure the processing of submitted files without consequences for the host computer, during the operation of the service, it is necessary to repeatedly destroy and recreate the virtualized environment. For ease of management, a protected virtualized environment is implemented using containers. Large community support, ease of installation and configuration, and many libraries for integration with programming languages and working environments [8], [9], [10], [11], [12], [13] to implement a virtualized environment for the code execution component, Docker was chosen. Each container is started with the help of a Docker image, a file that contains instructions for creating a Docker container and at the same time represents the basis for the file system that will be located inside the container. Each image has multiple layers that speed up the construction process. The Docker image used to start the virtualized environment in which the program code will be compiled and executed is g++:4.9, which contains the GNU C++ compiler g++.

After successfully starting a container using the docker start command, the container is ready to receive commands. If the execution time of the program inside the container exceeds the given time limit, the docker stop command will stop the container and its processes. After stopping the container, the container is deleted with the docker rm command so that it does not occupy resources unnecessarily (Figure 2).

F. *Implementation of code evaluation services*

In the data layer, a database is used to store relevant data about exams and assessment results. PostgreSQL was chosen as an open-source tool with strong community support and many libraries available for integration with a variety of programming languages and frameworks.

One exam can have one or more tasks. Each task has one or more test cases and one or more rules. For test cases, a text is defined that represents the standard input and the text of the standard output that will be compared with the actual output of the program. One or more rules can be defined for each task. Each rule contains a textual identifier, a description, and a regular expression of the check, the number of points that are deducted if the specific rule is not satisfied, and the number of occurrences of the lexical construct described by the rule. During the evaluation, a record is created that contains information about the year, semester, and exam, as well as information that indicates the status of the submission of the solution. An individual exam task contains the evaluation status waiting, which is true if the task has been fully evaluated. Each evaluation creates records of the result of checking test cases and rules.

To implement the business layer, the Go programming language and the Echo framework were used, which allows the creation of a REST web server service with all the necessary functionalities, such as controllers for accepting requests and defining processing methods, authentication filters, and logging management [8], [9], [10], [11]. Web service consists of two main packages: Executor and Marker. Inside the Executor package is the executor.Executor structure and associated methods for executing code. Each of these methods serves as a wrapper for calls made by the Docker agent. Some of the important methods are: ContainerList, ContainerStart, CopyToContainer, ContainerExecAttach, ContainerKill, and ContainerRemove [8], [9], [10], [11]. The Marker package defines structures and methods for evaluating individual tasks. The verification of test cases is carried out by calling methods for compiling and executing code from the Executor package, while the verification of rules is carried out by checking the existence of, certain lexical constructions in the source code using regular expressions. After the evaluation is complete, the results are stored using the Store package method.

Two values must be sent within the body of the initial POST request at the /submission path: exam_id, which represents the integer value of the universal identifier of the exam stored in the database, and project_zip, which is a compressed file of the MS Visual Studio Solution folder and related projects. If some value is missing, an HTTP response with the status code 400 Bad request is returned to the user. If the exam with the submitted identifier is not found, the status code 404 Not found is returned. If the request received the correct parameters, the compressed file is extracted, and the extracted files are stored on the server's file system. A directory is then created whose keys are the names of the tasks, and whose values are a list of files associated with that task, which includes a file with the extension cpp and optional files with the extension txt and csv used in the tasks.

The first step is to create a record that is the result of the assessment, and the user is returned an HTTP response with the status code 201 Created and the universal identifier of the result of the submission of the solution, submission_result_id. The method is not terminated after creating the HTTP response but continues with the evaluation. First, by calling the CompileCheck method, the possibility of translating the source code of the solution is checked. If the source code cannot be translated, the evaluation stops, and 0 points are awarded.

This is followed by verifying the test cases by executing them with the passed input data and verifying the expected and actual output data. When starting execution, as well as when compiling source code, a timer is started. Using the lexical construct select of the Go programming language, the process is stopped if the counter ends before the program is executed or compiled.

Rule checking is performed by finding patterns in the text of the source code using regular expressions [14], [15], e.g., to find the definition of a for loop, the regular expression is used: `[^a-zA-Z0-9_]for [ \t\n]*\(`. The search is for matching text that contains for loop definition.

The evaluation process can take a long time, so the system evaluates solutions simultaneously using the goroutines of the Go programming language. Each goroutine is active while the assessment is in progress and ends with the recording of the obtained points. The client application continuously send a GET request to the /submission/:id path to display the grading status and current scoring results. This process can be stopped when the waiting attribute of the JSON response body is set to false (Figure 1).

The initial POST request to the path /submissions returns the HTTP response that contains submission identifier. The submission_result_id attribute is a universal identifier of the entity that represents the result of the evaluation of the solution, and it must be sent as a parameter when calling the GET request according to the path /submission/:id where id is the value of the submission_result_id attribute (Figure 3).

Next, the GET request to the specified path returns a JSON response in which the waiting attribute is specified for each task, which indicates the status of the evaluation of the solution:

```
{"id": 101,
  "created_at": "2022-05-
18T22:58:12.814289+02:00",
  "exam": {
    "id": 1,
    "name": "Midterm Exam 1"},
  "final": false,
  "task_results": [{
      "id": 62,
      "waiting": true,
      "compiles": true,
      "scored_points": 0,
      "message": "",
      "task_id": 1,
      "test_case_results": [{
            "id": 41,
            "passed": true}],
  "test_cases_passed": 1,
  "rule_results": [{
    "id": 22,
    "satisfied": false,
    "points_affected": 0.5}],
  "rules_satisfied": 0}]}
```

III. RESULTS

The basic application of the presented solution is in the automated correction of exams from the course Programming. The course contains six learning outcomes; therefore, it is essential that the system can support the definition of rules and conditions that the student needs to meet the learning outcomes.

The conditions are:

- defining different types of variables
- defining for, while and do while loops
- defining methods with different signatures
- defining and using structures
- defining and using pointers and references

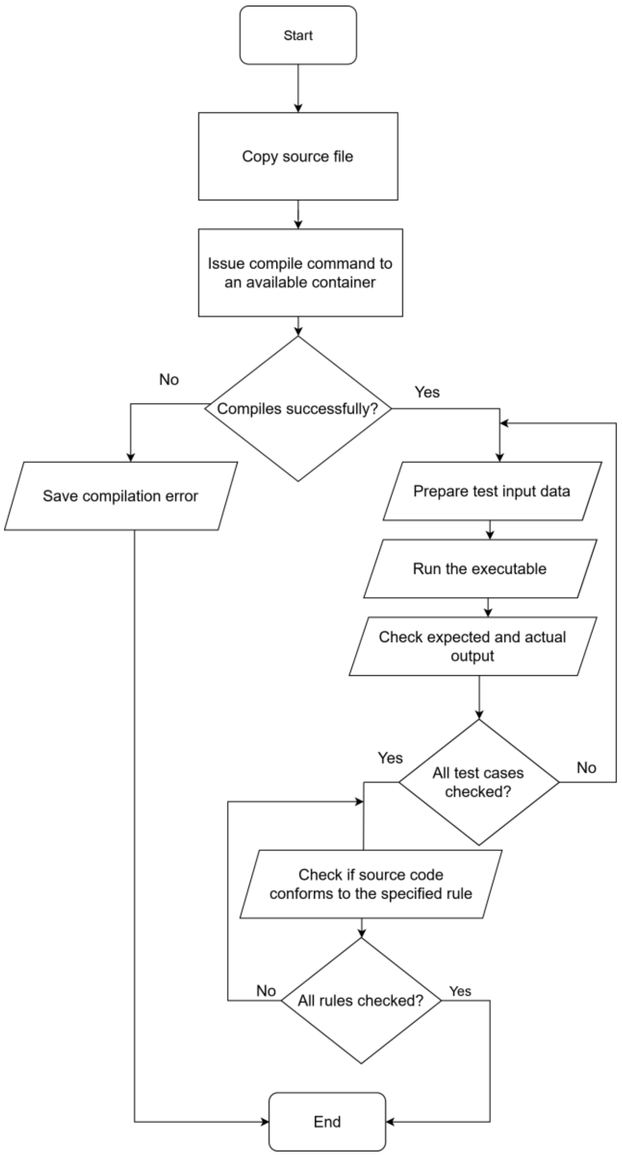


Figure 1. Solution evaluation process

- defining and using ifstream and ofstream file management types.

A. *Example of use for assessing program solutions*

During the exam, each student creates his own program solutions based on the assigned tasks. The following tasks were selected as an example of using the proposed solution:

1. It is necessary to ask the user to enter two whole numbers and print their sum. (2 points)
2. It is necessary for the user to load the word and print "yes" if it is a word, and print "no" otherwise. (3 points)
3. Ask the user to enter the height of an isosceles triangle and to print the corresponding triangle using the star symbol "*". An example of a printout (4 points):

```
   *
  ***
 *****
*******
```

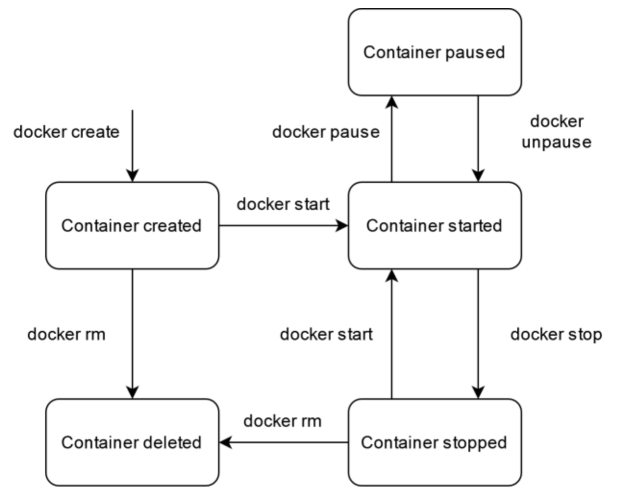


Figure 2. Docker container life cycle

Students use the MS Visual Studio 2022 Integrated Development Environment and select the Empty project form, thus defining the project that will contain the source code of the solution to the first task. In addition to the project, the student also defines a solution that represents a group of projects connected to one exam solution. The next step is to create projects for each remaining task.

Students can check the current solution while solving the exam tasks. The folder created by the project should be compressed and transferred to the evaluation system as such. After finishing the evaluation, an overview of the scoring details is available (Figure 4 for the first task). The source code of the solution is translated, and all test cases and rules are satisfied, and the solution of the task achieves the maximum number of points. Figure 5 shows the scoring details of the second task. Although the solution is translatable and satisfies all the rules for the existence of lexical constructions, the source code contains a logic error that results in only one test case being satisfied. The source code of the solution of the third task cannot be translated and the task achieves 0 points. The program execution procedure and the checking of rules and test cases were not carried out (Figure 6).

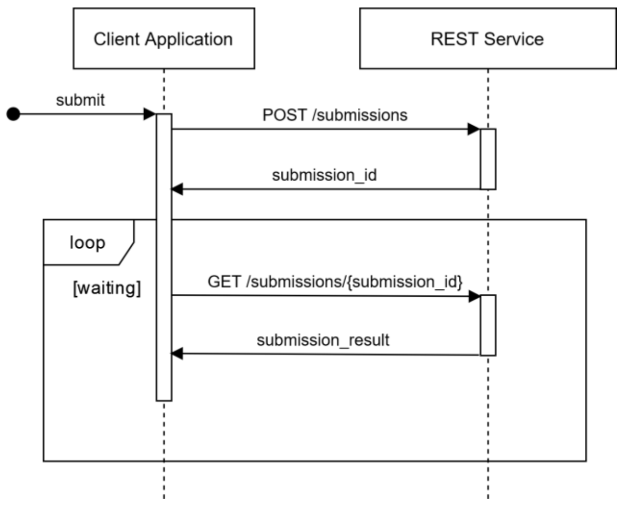
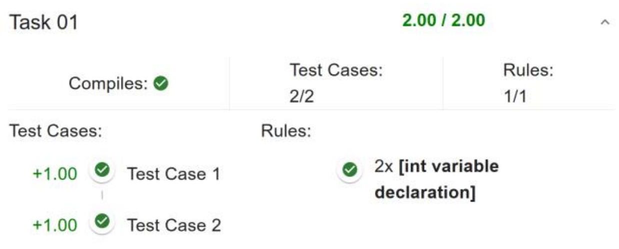


Figure 3. Communication between the client application and the server service



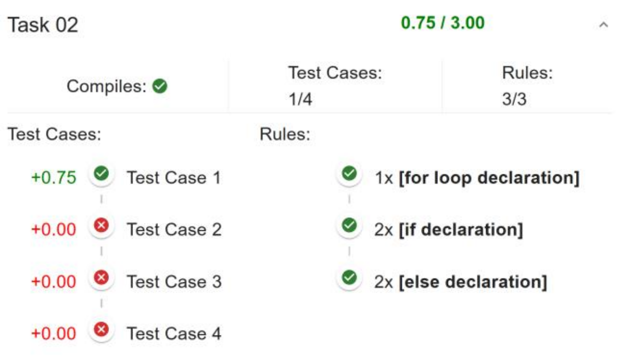
Task 01 2.00 / 2.00

Compiles: ✓	Test Cases: 2/2	Rules: 1/1

Test Cases:
+1.00 ✓ Test Case 1
+1.00 ✓ Test Case 2

Rules:
✓ 2x **[int variable declaration]**

Figure 4. First task evaluation results



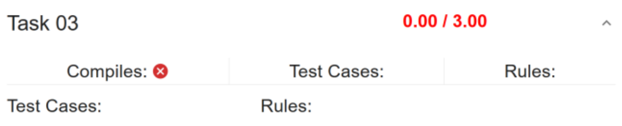
Task 02 0.75 / 3.00

Compiles: ✓	Test Cases: 1/4	Rules: 3/3

Test Cases:
+0.75 ✓ Test Case 1
+0.00 ✗ Test Case 2
+0.00 ✗ Test Case 3
+0.00 ✗ Test Case 4

Rules:
✓ 1x **[for loop declaration]**
✓ 2x **[if declaration]**
✓ 2x **[else declaration]**

Figure 5. Second task evaluation results



Task 03 0.00 / 3.00

Compiles: ✗	Test Cases:	Rules:

Test Cases:

Rules:

Figure 6. Third task evaluation results

B. *Example of use with a system for grading sql solutions*

Analogous to the process of evaluating C++ program solutions, it is possible to evaluate SQL solutions using the system for evaluating SQL solutions previously developed at the College of Algebra [16]. Parameters marked with the value of the Content-Type header set to multipart/form are submitted to the system via an HTTP request on the path /submissions: *config* with the task settings, and *sql* with the SQL file of the student's solution.

Task settings define rules that indicate the presence of certain lexical constructions of SQL queries, such as keywords SELECT, FROM and ORDER BY, as well as the course of evaluation if a certain rule is not respected. The SQL file contains the solutions of several tasks that are separated within the source code by the comment mark --LOXTY, where X is the sequence number of the learning outcome, and Y is the sequence number of the task.

After successfully receiving the request, the SQL solution evaluation system performs a static analysis of the code as well as checking whether the SQL query returns the necessary rows defined by the task. A database container is used when executing SQL queries. If the specified container is not started, a command is given to start the container, within which an SQL script is then executed to create a database filled with the initial data needed to test the student's solution.

The JSON response body for the HTTP POST request on the path /submissons contains a universal identifier of the solution that needs to be sent during each subsequent HTTP GET request on the path /submissions/:id. The response to the specified HTTP request contains the number of points achieved for each task, as well as feedback on the reasons for possibly deducted points.

C. Disadvantages of the proposed system

Naive manual testing of the system revealed the shortcomings of the developed solution. The name of each task within the student's solution must explicitly match the defined task name within the exam. The output data in the standard output stream must explicitly match the expected output data.

An improvement to such an approach would be to implement metrics when checking task names and output data. An example is the Levenshtein distance [8], [17]. When checking the test cases, it would be necessary to define the permissible value of the Levenshtein distance between two strings of characters to avoid deducting points due to random errors. For example, it is possible to define a Levenshtein distance of 1, if one wants to ignore a misspelled letter in the solution.

Static analysis of the presence of lexical constructions is performed using regular expressions, which are not an optimal tool when it comes to performance. Due to their limitations, regular expressions are not the most suitable tool for finding specific lexical constructions [14], [15]. The development of a parser that creates an abstract syntax tree from the given source code, and whose search is more precise than using regular expressions would improve proposed system.

The presented solution has a monolithic architecture, and two microservices should be developed instead: for code execution and for evaluating the solution. With such an approach, the system would enable a larger set of functional requirements for different test tasks. Such an agnostic architecture would more simply implement a system for evaluating software solutions of other programming languages within a common virtualized environment.

IV. Conclusion

The developed grading system allows examiners to automate grading. The key parts of the code and commands are analyzed, and the use of the developed system is described along with the corresponding textual description and graphical representation of the user interface. The proposed architecture enables the development of different types of clients that can call the developed system. After analyzing the examples of the developed system use, shortcomings were observed, and possible system upgrades were suggested.

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] H. Z. Dosilovic and I. Mekterovic, “Robust and Scalable Online Code Execution System,” in 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), Sep. 2020, pp. 1627–1632, doi: 10.23919/MIPRO48935.2020.9245310.

[3] I. Mekterovic, L. Brkic, B. Milasinovic, and M. Baranovic, “Building a comprehensive automated programming assessment system,” IEEE Access, vol. 8, pp. 81154–81172, 2020, doi: 10.1109/ACCESS.2020.2990980.

[4] 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.

[5] A. Z. Javed, P. A. Strooper, and G. N. Watson, “Automated generation of test cases using model-driven architecture,” Proc. - Int. Conf. Softw. Eng., 2007, doi: 10.1109/AST.2007.2.

[6] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2013.

[7] D. Merkel, “Docker: lightweight Linux containers for consistent development and deployment,” Linux Journal, vol. 2014, no. 239. p. 2, 2014, Accessed: Jan. 26, 2023. [Online]. Available: https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf.

[8] T. Combe, A. Martin, and R. Di Pietro, “To Docker or Not to Docker: A Security Perspective,” IEEE Cloud Comput., vol. 3, no. 5, pp. 54–62, 2016, doi: 10.1109/MCC.2016.100.

[9] R. Zhang, A. M. Zhong, B. Dong, F. Tian, and R. Li, Container-VM-PM Architecture: A Novel Architecture for Docker Container Placement, vol. 10967 LNCS. Springer International Publishing, 2018.

[10] T. Bui, “Analysis of Docker Security,” 2015, [Online]. Available: http://arxiv.org/abs/1501.02967.

[11] A. M. Potdar, D. G. Narayan, S. Kengond, and M. M. Mulla, “Performance Evaluation of Docker Container and Virtual Machine,” Procedia Comput. Sci., vol. 171, no. 2019, pp. 1419–1428, 2020, doi: 10.1016/j.procs.2020.04.152.

[12] P. Gkikopoulos, V. Schiavoni, and J. Spillner, “Analysis and Improvement of Heterogeneous Hardware Support in Docker Images,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2021, vol. 12718 LNCS, pp. 125–142, doi: 10.1007/978-3-030-78198-9_9.

[13] N. Vermeir, “Application Architecture,” Introd. .NET 6, pp. 259–273, 2022, doi: 10.1007/978-1-4842-7319-7_9.

[14] P. Bille and M. Farach-Colton, “Fast and compact regular expression matching,” Theor. Comput. Sci., vol. 409, no. 3, pp. 486–496, Dec. 2008, doi: 10.1016/J.TCS.2008.08.042.

[15] A. Backurs and P. Indyk, “Which Regular Expression Patterns Are Hard to Match?,” Proc. - Annu. IEEE Symp. Found. Comput. Sci. FOCS, vol. 2016-December, pp. 457–466, Dec. 2016, doi: 10.1109/FOCS.2016.56.

[16] M. Fabijanic, G. Dambic, and B. Fulanovic, “A novel system for automatic, configurable and partial assessment of student SQL queries,” 2020 43rd Int. Conv. Information, Commun. Electron. Technol. MIPRO 2020 - Proc., pp. 832–837, 2020, doi: 10.23919/MIPRO48935.2020.9245264.

[17] L. Yujian and L. Bo, “A normalized Levenshtein distance metric,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 29, no. 6, pp. 1091–1095, 2007, doi: 10.1109/TPAMI.2007.1078.