CodeShuffler 2.0: Advancing Paper-Based Assessments in the Age of Generative AI

Shawn Spitzel, University of Connecticut
Prof. Hasan Baig, University of Connecticut

Dr. Hasan Baig currently serves as an Assistant Professor at the University of Connecticut's Stamford Campus, USA. Prior to joining the Computer Science and Engineering (CSE) department, Dr. Baig was a Postdoctoral Research Fellow in the Mendes Research Group at UConn Health, USA, where he focused on developing cloud-based computational tools for systems biology.

Dr. Baig is award winning educationist who has developed numerous EdTech tools for students and instructors. Other than educational technology, his research spans a diverse range of interdisciplinary areas, including Genetic Design Automation (GDA), Cloud Computing, Embedded Systems, Computer Architectures, Fault-Tolerant Systems, and Human-Machine Interactions.

For any questions related to this work, please reach out at https://www.hasanbaig.com/contact/

CodeShuffler 2.0: Advancing Paper-Based Assessments in the Age of Generative AI

I. Abstract

The widespread adoption of large language models (LLMs), such as ChatGPT, has shown great promise in increasing student workflow and productivity, but comes with significant drawbacks in maintaining academic integrity. These models come with the ability to generate and provide detailed solutions for programming problems in any language, and complicate the process of fair assessment via automated platforms. To address this, many curricula have introduced paper-based coding exams to minimize outside help from LLMs. While more accurate, these exams create grading and logistical challenges for instructors, especially in large courses. To address this trade-off, prior work introduced CodeShuffler, an open-source framework that converts paper-based coding assessments into structured, multiple-choice formats by shuffling code lines, thereby enabling efficient and scalable grading while preserving assessment integrity. Building on this previous work, this paper presents CodeShuffler v2, an evolution of the original CodeShuffler system designed to improve grading fidelity, scalability, and instructor usability. The new version introduces a structurally aware partial-credit scoring algorithm that more accurately reflects intermediate student understanding, as well as multi-versioned exam generation with the introduction of ExamShuffler. Moreover, the system transitions from a command-line tool to a fully integrated graphical application, streamlining exam creation, configuration, and export. Collectively, CodeShuffler v2 aims to serve as a practical framework for fair, scalable, and instructor-friendly paper-based coding assessments.

II. Introduction

Computer science is a field that has seen rapid growth in recent years. The number of bachelor's degrees earned in computer and information sciences more than doubled over the past decade, rising from 51,696 (2013–2014) to 112,720 (2022–2023), even as total degree production in other majors declined [1]. With this immense growth comes an increased workload for instructors, namely with proctoring and grading assessments, especially in introductory programming courses where high enrollment and diverse student backgrounds make it difficult to achieve reliable assessment. Traditionally, this balance has been achieved through a combination of take-home assignments and auto-graded, computer-based coding platforms, which substantially reduce grading effort for instructors. In a study from Gordon et. al, classroom data indicated that the adoption of cloud-based auto-graders substantially reduced manual grading time by 9 hours/week on average, and scaled to thousands of students and instructors, which showed great promise for future automation [2].

However, the widespread adoption of large language models (LLMs), such as ChatGPT [3], has fundamentally altered this assessment landscape. These models are capable of generating high-quality code in virtually any programming language, often accompanied by detailed explanations and documentation. While this offers clear benefits for both learning and productivity, it simultaneously complicates instructors'

ability to assess student understanding. High-quality code generation is making it increasingly difficult to distinguish student submissions from those generated by AI, rendering grades less reflective of individual competence and undermining the validity of assessments.

In response to these challenges, many instructors have begun to reconsider the structure and delivery of programming assessments. However, addressing this need demands not only pedagogical adjustments but also the tooling and infrastructure necessary to support efficient exam creation, grading, and deployment at scale. This paper aims to provide this need by presenting a refined and scalable assessment framework designed explicitly around the practical constraints faced by instructors.

III. Background and Related Work

A variety of approaches have been explored to address the challenges of assessing programming knowledge at scale. Auto-graded coding platforms like ZyBooks [4] have become widely adopted due to their ability to reduce instructor workload and provide rapid feedback. These systems typically evaluate program correctness based on program output or predefined unit tests, which allows for efficient grading within large classes [4].

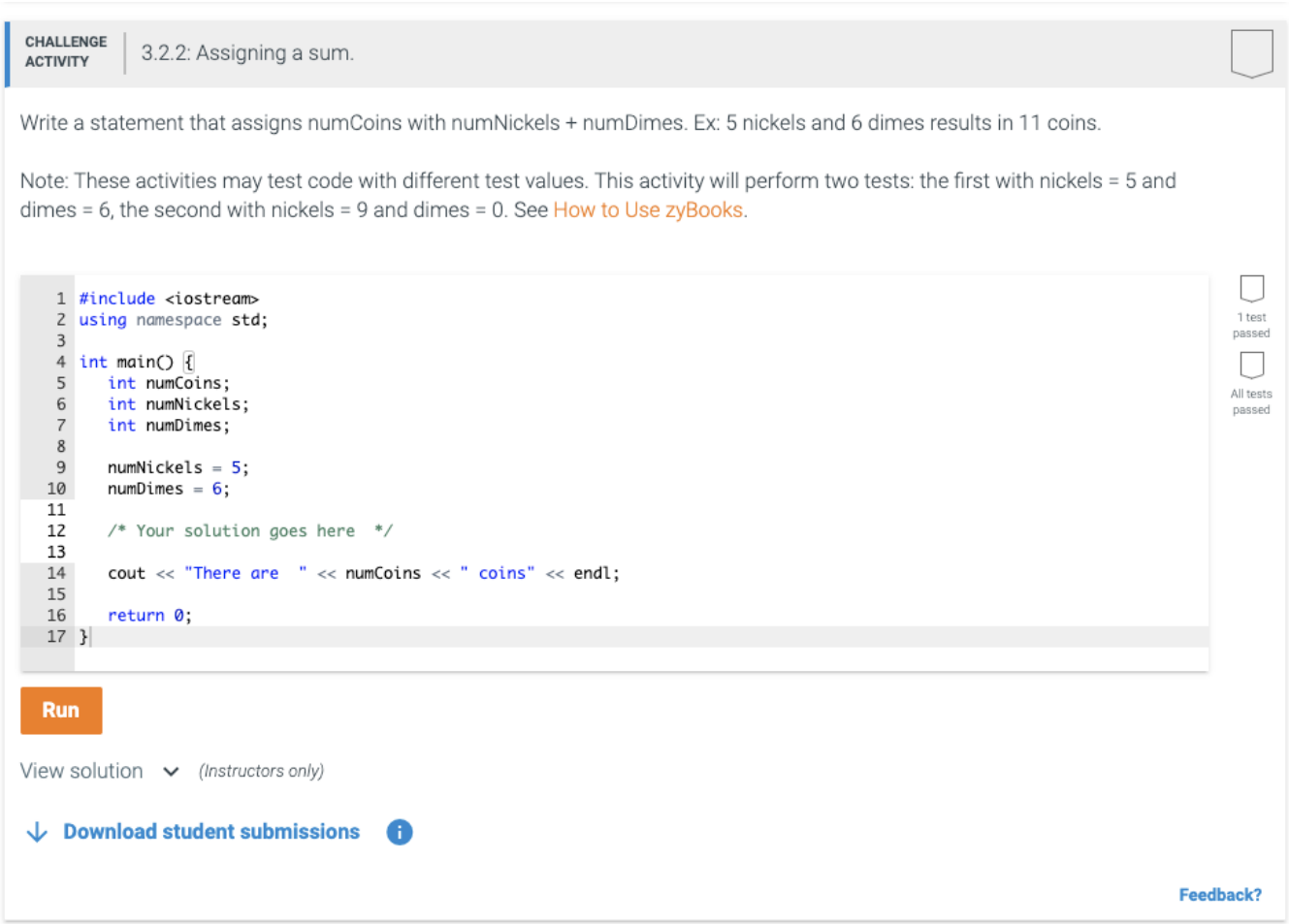

Fig. 1. An example of a ZyBooks programming assignment [4].

However, such platforms primarily assess final outputs rather than students' reasoning processes or understanding of code structure, and have become increasingly vulnerable to AI-generated submissions in the presence of LLM-based code assistants.

To mitigate these concerns, paper-based coding assessments have re-emerged as an alternative means of evaluation. Paper-based exams emphasize students' ability to reason about program structure, control flow, and logical sequencing without access to external tools. Despite their advantages, paper-based coding exams introduce significant logistical challenges for instructors. Manual grading is time-consuming and error-prone, particularly in large-enrollment courses, often resulting in delayed feedback and increased instructor workload. Additionally, instructors must invest substantial effort in generating unique code snippets across exam versions to maintain academic integrity between class sections. Prior work introduced the CodeShuffler system [5] , which addressed this trade-off by introducing an automated framework that transforms code snippets into shuffled, multiple-choice reconstruction tasks. This approach aimed to preserve the granular assessment qualities offered by standard paper-based coding assessments while mitigating manual intervention. Classroom deployments demonstrated the feasibility of this approach and highlighted its potential to reduce grading burden while maintaining meaningful assessment [5].

While CodeShuffler [5] established the viability of structured paper-based coding assessments, it also revealed several limitations that constrained broader adoption.

CodeShuffler is an open-source tool that was introduced as a proof-of-concept system to address the growing difficulty of conducting fair paper-based coding assessments in the presence of AI-assisted code generation [5]. By transforming code snippets into shuffled multiple-choice question options, the system demonstrated that paper-based coding exams could be structured in a way that preserved assessment integrity while significantly reducing manual grading effort. Classroom deployments showed that this approach enabled instructors to evaluate students' understanding of basic programming principles, and also highlighted the disparity between paper-based and computer-based assessments [5].

Despite its effectiveness in addressing core assessment challenges, the original CodeShuffler implementation, also called CodeShuffler v1, possessed several limitations that inhibited broader application and scalability. One such limitation was its reliance on binary grading. Student responses were evaluated as either fully correct or incorrect, providing no mechanism to capture partially correct reasoning or intermediate levels of understanding. This all-or-nothing scoring failed to reflect the nuanced, structurally dependent nature of programming, where partial correctness often indicates meaningful conceptual progress. As a result, instructors seeking more flexible evaluation were required to manually review submissions for partial-credit grading, reintroducing grading overhead. Secondly, CodeShuffler v1 lacked built-in support for automated multi-version exam generation. Instructors seeking to distribute multiple equivalent versions of an exam were required to manually rerun the tool and manage versioning externally. This process again lacked scalability and introduced additional labor, since distributing distinct assessments is a necessity for maintaining academic integrity.

Most notably, the original system was implemented exclusively as a command-line utility. While functional, this design imposed an apparent usability barrier for instructors without prior experience in programming or terminal-based workflows. Common tasks, such as configuring shuffle parameters, inserting exam headers or footers, and exporting finalized artifacts, all required manual configuration and limited the application's accessibility.

These limitations highlight the gap between a functional prototype and a production-ready assessment

system, and while the original CodeShuffler established the viability of automated paper-based coding assessments, it did not fully address the practical implications to be faced when deploying such assessments at scale. These shortcomings motivated the design of CodeShuffler v2, which aims to address these limitations through a more comprehensive system architecture.

IV. CodeShuffler v2

CodeShuffler v2 is designed as an end-to-end system for creating, managing, and grading paper-based coding assessments at scale. Unlike its predecessor, which primarily functioned as a code transformation utility, CodeShuffler v2 owns the exam-generation process from start to finish. At a high level, the system operates as a pipeline that begins with instructor-authored source material and ends with deployable exams with answer keys, shuffled questions, and unique versioning. Figure 2 illustrates the overall system architecture and the flow of data between components.

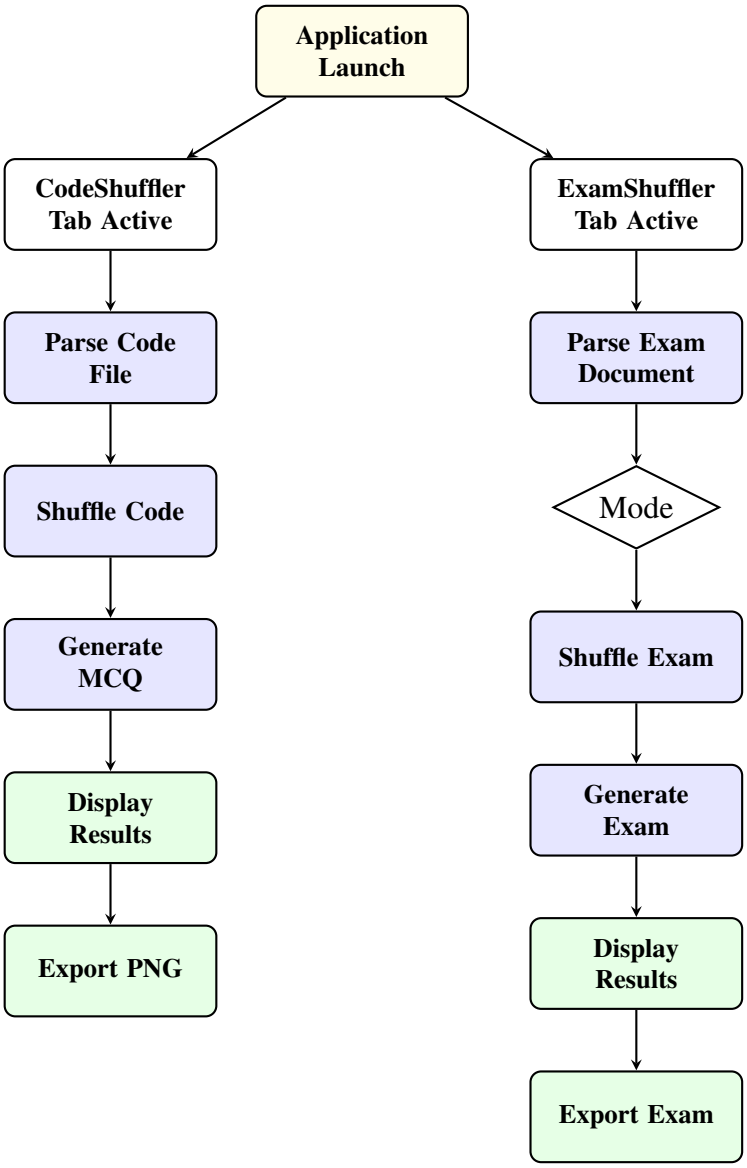

Fig. 2. High-level overview of the CodeShuffler and ExamShuffler systems

The architecture is organized around two distinct but complementary workflows, each addressing different assessment scenarios. The CodeShuffler workflow transforms individual code snippets into shuffled multiple-choice questions with structured partial credit, producing PNG images suitable for embedding in paper exams. The ExamShuffler workflow generates multiple versions of complete exams by systematically shuffling question and answer order within existing exam documents, producing DOCX files ready for distribution. Both workflows share common underlying components, including a parser for extracting structured data, a shuffler for applying randomization, and export utilities for generating outputs, but operate independently and are accessed through separate interfaces within the graphical application.

A. Architectural Overview: CodeShuffler

The CodeShuffler pipeline consists of four sequential processing stages, including parsing, shuffling, answer generation, and image rendering. Each stage operates on the output of the previous stage.

The CodeShuffler parser accepts a source code file in any programming language, structured with two components: the correct implementation, and a hashmap of correct lines mapped to semantically plausible incorrect variants. The parser extracts these components, separating them and storing them. The parsed output is then passed into the shuffler, which uses a randomization algorithm to reorder lines and assign numeric prefixes. Importantly, both correct lines and their incorrect variants are included in the shuffled output. This creates a pool of lines from which students must select the correct subset in the correct order.

From this pool of lines, the correct answer is generated by tracing the positions of correct lines through the shuffle, and applies the optional partial-credit algorithm by generating intermediate answer options. Random distractors are generated to fill any remaining multiple-choice slots. The result is an answer set with a correct answer, partial credit options, and incorrect answer options. The shuffled code is finally rendered as a PNG image with adaptive dimensions, configured through user settings.

B. Architectural Overview: ExamShuffler

The ExamShuffler pipeline also consists of four, albeit slightly different, sequential processing stages. These stages include parsing, shuffling, preview rendering, and document regeneration.

The ExamShuffler parser accepts a Microsoft Word (.docx) document formatted as a standard exam, and extracts structured question data by identifying question boundaries, extracting question text and answer choices, detecting embedded code blocks, and tracking correct answer positions. The result is a structured representation that separates questions, answers, embedded code snippets, and formatting. From here, the instructor can decide to either shuffle exam questions, shuffle exam answers, or shuffle both. Question shuffling applies a random permutation to the question sequence, completely randomizing the ordering of questions. Answer shuffling applies independent random permutations to the answer choices within each question, reordering the position of each choice while maintaining the original question order. Combined shuffling applies both operations sequentially, and is preferred for achieving maximum variation between versions.

Lastly, ExamShuffler reconstructs a new exam document from the shuffled question data. Document headers and footers from the original template are preserved, ensuring that institutional branding and course identifiers are maintained across all generated versions.

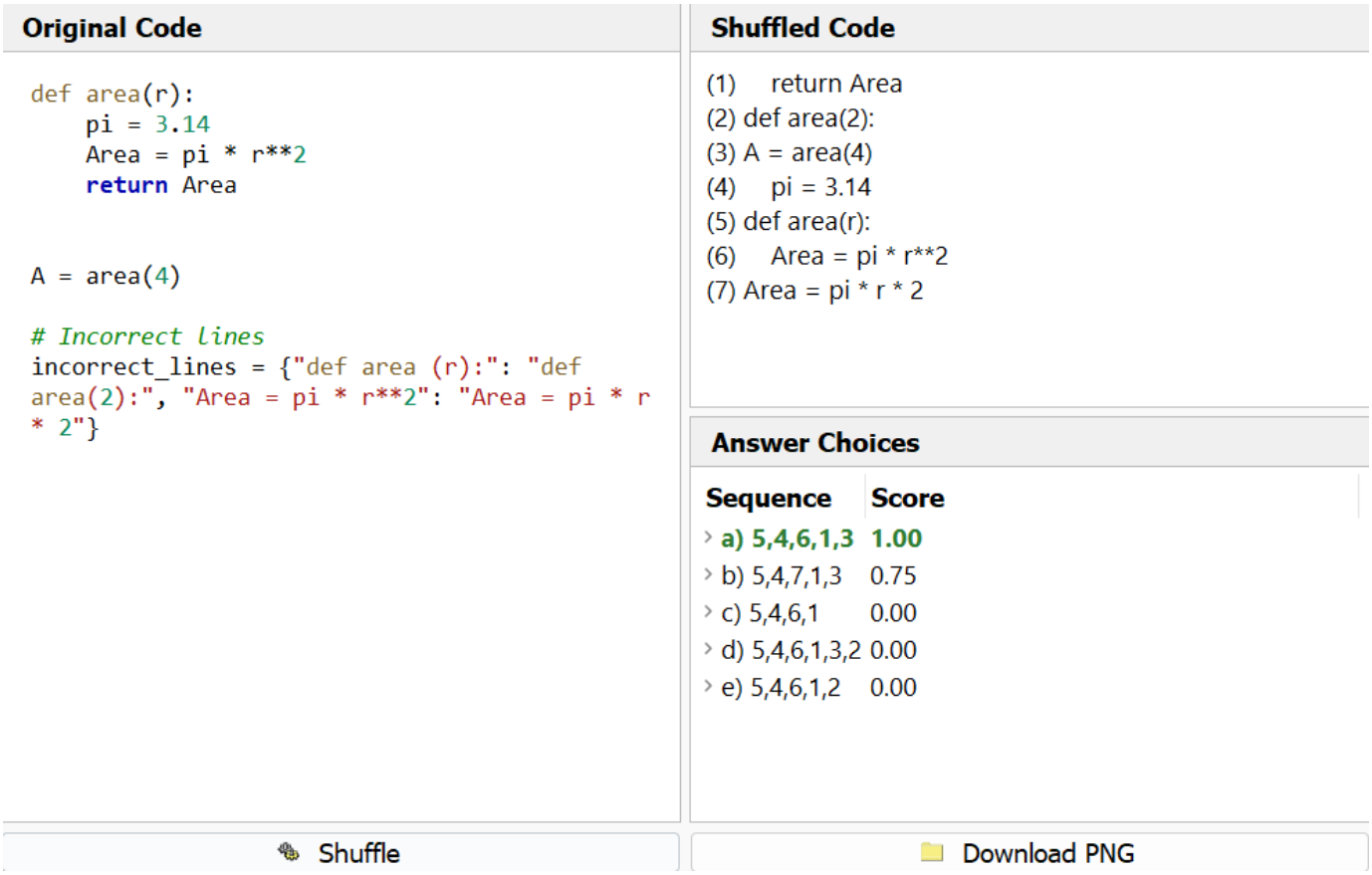

Fig. 3. A full-size screenshot of the CodeShuffler tab, complete with syntax highlighting, shuffled output preview, and answer choices

V. Architectural Overview: CodeShuffler

The following sections provide technical details on the implementation of the primary subsystems introduced in the previous section. The focus will be on algorithmic design, implementation decisions, and the technical rationale for specific architectural choices. This section in particular focuses on the implementation and design behind the CodeShuffler system.

CodeShuffler starts by taking the correct implementation after separating it from incorrect variant mappings, where lines are extracted, randomized using the Fisher-Yates shuffle algorithm [6], and assigned numeric prefixes. Critically, when a line ℓ_j has an incorrect variant ℓ (ℓ_j), both are included in the shuffled output. Students must identify which lines belong in the solution and in what order, adding complexity beyond basic reordering.

The correct answer is computed by following the positions of correct lines through the shuffling process. Let $C = \{\ell_1, \ell_2, \ldots, \ell_n\}$ denote the correct implementation and S the shuffled sequence. The algorithm iterates through C locating each line ℓ_1 in S and recording its position. The correct answer is the sequence of these positions in original order.

After shuffled options are computed, partial-credit options are brought into the mix to help saturate the answer set. In the original CodeShuffler system, binary grading was the only evaluation method available for shuffled questions. CodeShuffler v2 introduces a new partial-credit scoring algorithm to change this.

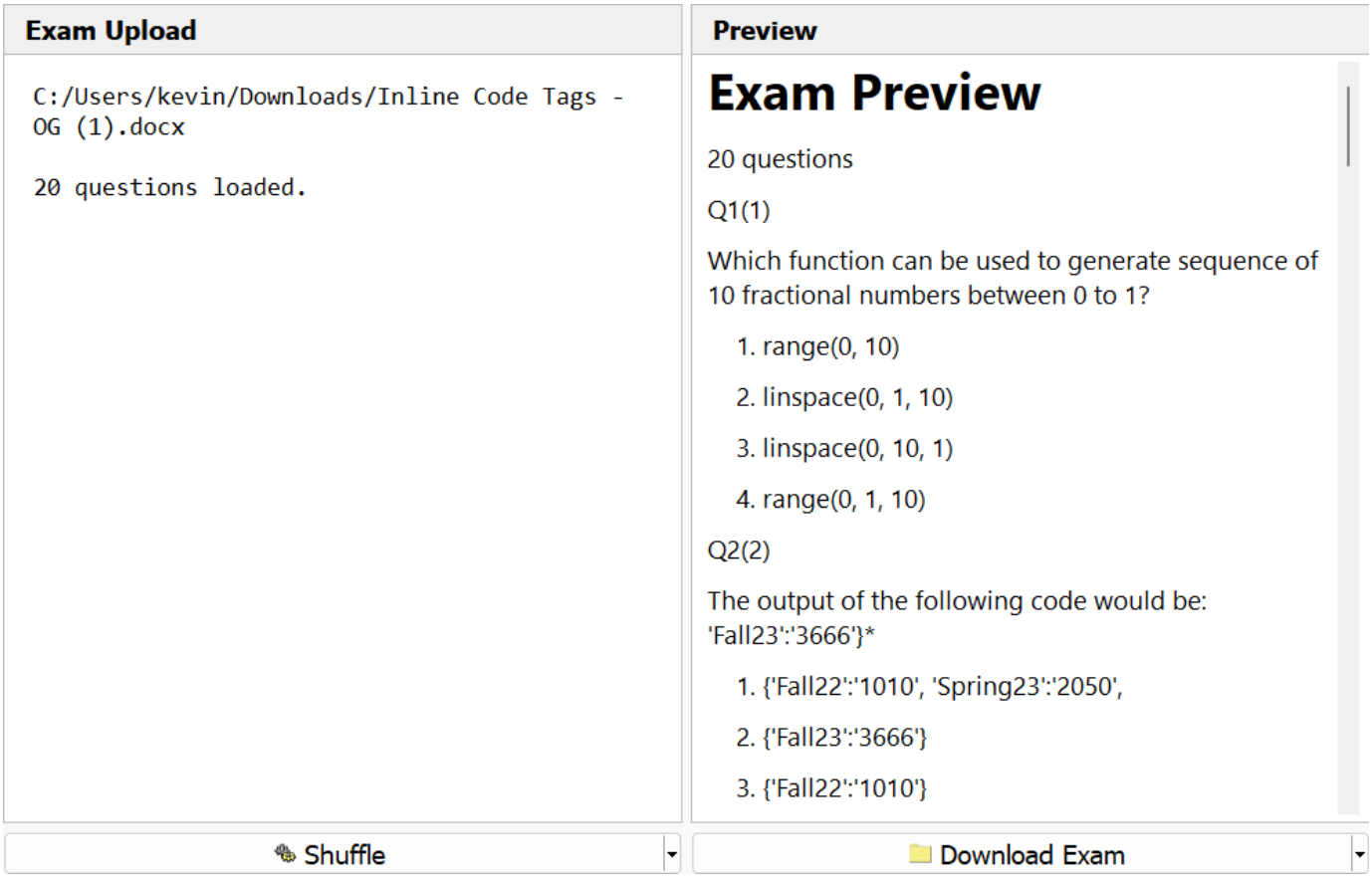

Fig. 4. A full-size screenshot of the ExamShuffler tab, consisting of file drop and preview components.

Consider a code snippet C consisting of n lines $\{\ell_1, \ell_2, \ldots, \ell_n\}$ that must appear in a specific order to execute correctly. A shuffling operation produces a permutation $\sigma \in S_n$ of these lines, yielding a reordered sequence in which each line is assigned a numeric prefix. The correct answer A is the sequence of indices that restores the original order:

$$A = \left[\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(n)\right].$$

Binary grading evaluates a student response R as correct if and only if $R = A$, assigning full credit to exact matches and zero credit to all other responses. This provides no differentiation between responses that differ minimally from A and responses that bear no structural relationship to the correct sequence. The partial-credit algorithm addresses this limitation by generating intermediate answer choices

$$\{A_1, A_2, \ldots, A_k\},$$

where each A_i differs from A by exactly i line swaps.

The key challenge is ensuring that swaps reflect semantically meaningful errors rather than arbitrary perturbations. CodeShuffler achieves this by allowing instructors to create their own mappings of correct lines to plausible incorrect alternatives. For example, consider the following maximum-finding function:

```
def max_num(nums):
    max_val = nums[0]
    for n in nums:
```

```
        if n > max_val:
            max_val = n
    return max_val
```

An instructor may define an incorrect-variant mapping I^* such that $I * (\texttt{max\_val = nums[0]}) = \texttt{max\_val = 0}$, reflecting a common initialization error. Similarly, $I * (\texttt{return max\_val}) = \texttt{return n}$ captures the mistake of returning the loop variable rather than the accumulator.

These mappings enable the generation of partial-credit options that reflect realistic student misconceptions while constraining the space of incorrect answers to those that remain logically plausible. Partial-credit options are generated by systematically swapping the positions of correct lines with the positions of their corresponding incorrect variants.

For each line ℓ_j that has an associated incorrect variant $I * (\ell_j)$, the following procedure is applied:

1) Identify the positions of ℓ_j and $I * (\ell_j)$ in the shuffled sequence.
2) Locate the position of ℓ_j within the student answer ordering A.
3) Swap this position with the position corresponding to $I * (\ell_j)$.
4) Add the resulting sequence to the partial-answer bank.

Instructors may specify the number of partial-credit options by defining a number of swaps, in which the credit given for these options are dependent on. Let s denote the number of swaps required to transform an incorrect option into the correct ordering. The awarded credit is computed as

$$C(s) = \max\big(1 - 0.25\,s,\ 0\big),$$

such that each additional swap incurs a linear 0.25 decrement in credit, taking the maximum value between it and 0.

VI. Architectural Overview: ExamShuffler

The architecture for ExamShuffler is slightly more involved, as semantically parsing exam documents is nontrivial. Upon document upload, the parser attempts to convert .docx files into nested list structures, then applies regex-based pattern matching to identify question boundaries and components.

Questions are detected by lines beginning with numeric or bullet markers. For each detected question, the system scans subsequent lines for answer choice prefixes, extracts choices as a list, and records labels. If an answer choice ends with an asterisk, it is flagged as correct and the asterisk is stripped. Code blocks within questions are delimited by `<code>... </code>` tags and extracted separately for special styling in regenerated documents. This provides a mapping of question numbers to dictionaries containing question text, code blocks, answer choices, and correct answer indices, which is later used to recreate the exam from scratch. An example is provided below.

1. *For the following class, the parameter names in the initializer/constructor method are:*

```
<code>
class Test:
    def __init__(self, a, b):
        ...
```

```
5
6       def construct(self, c, d):
7           ...
8
9       def initialize(self, e, f):
10          ...
11
12      def clear(self, g, h):
13          ...
14  </code>
```

1) a, b*
2) c, d
3) e, f
4) g, h

Once parsing has completed, the next step in the process is shuffling. The system supports three independent shuffling modes, giving users the option to shuffle questions, answers, or both.. Question shuffling applies a random permutation to the question sequence, reordering presentation while preserving all content. Answer shuffling applies independent random permutations to answer choices within each question. Combined shuffling applies both operations sequentially, producing maximum variation. Since indices are tracked within each question independently, question reordering does not affect answer key correctness. The combinatorial capacity provided is more than enough for deploying unique exam versions, as for n questions with m answer choices each, combined shuffling yields n! × (m!)pow(n) distinct versions.

Once exam data is stored and shuffled, ExamShuffler then attempts to regenerate the exam from scratch using the updated exam structure. This is done by iterating through the updated exam structure and appending each question, along with its corresponding answers, onto a new exam document. Once completed, users have the ability to export and print for use.

VII. Evaluation

We evaluate CodeShuffler v2 as a software system intended to support scalable, paper-based coding assessments. As such, our evaluation focuses on system-level properties, including determinism, robustness under randomized generation, invariant preservation, and correctness across multi-version exam generation. The evaluation considers the CodeShuffler and ExamShuffler independently. Each system was subjected to repeated randomized test runs using both fixed and varying seeds. Fixed seeds were used to test determinism and reproducibility, while varying seeds were used to assess shuffle diversity and distributional behavior. All metrics reported are aggregated across 1000 independent runs. We report the following system-level metrics:

- **Collision Rate**: The percentage of shuffle operations that produce duplicate outputs. Lower values indicate greater effective randomness and higher exam diversity.
- **Seed Reproducibility**: The percentage of runs in which identical random seeds produce identical outputs. This value should be exactly 100% for deterministic systems.
- **Invariant Pass Rate**: The percentage of runs in which system invariants are preserved, including correct answer tracking, absence of data loss, and consistency of grading metadata.

- **Position Uniformity**: A statistical measure of whether shuffled items appear uniformly across positions.
- **Correctness Rate**: The percentage of runs in which regenerated exams maintain complete data integrity.

Across these metrics, Table 1 showcases the results of CodeShuffler and ExamShuffler:

TABLE I
EVALUATION RESULTS FOR CODESHUFFLER V2 COMPONENTS

Metric	CodeShuffler	ExamShuffler
Collision Rate	9.16%	0.18%
Seed Reproducibility	100.00%	100.00%
Invariant Pass Rate	98.81%	100.00%
Position Uniformity	91.77%	N/A
Correctness Rate	N/A	100.00%

Both subsystems achieved perfect seed reproducibility, confirming that CodeShuffler v2 behaves deterministically under fixed seeds. This property is essential for assessments, as it enables reproducible exam generation and consistent re-grading without much manual overhead. Shuffle quality is characterized by collision rate and position uniformity. ExamShuffler achieved a collision rate of 0.175%, indicating strong output diversity even under repeated generation. CodeShuffler exhibited a higher collision rate of 9.16%, which is expected considering code snippets are considerably shorter than exams, and under identical iterations will converge to identical outputs more often. Position uniformity for CodeShuffler reached 91.77%, an expected outcome given that code snippets contain structural dependencies that must be preserved during shuffling to ensure syntactic correctness.

Invariant preservation represents a non-negotiable requirement for assessment systems, of which ExamShuffler maintained a 100%, along with correctness rate. This finding confirms that multi-version exam generation does not compromise grading integrity or alignment. CodeShuffler achieved an invariant pass rate of 98.81%. While high, this result indicates the presence of rare failure cases. Overall, this evaluation demonstrates that CodeShuffler v2 provides determinism, correctness, and robustness suitable for deployment in large instructional settings.

VIII. Limitations and Future Work

While CodeShuffler v2 substantially improves the automation of paper-based coding assessments, several limitations remain that constrain its current applicability.

First, ExamShuffler does not explicitly model dependencies between questions. When shuffling exams, questions are treated as independent units with no relation to one another. As a result, two questions that rely on shared context or sequential reasoning may be reordered in a way that disrupts instructor intent. Instructors must therefore ensure that dependent questions are either grouped manually or excluded from question-level shuffling. While this design choice simplifies the system and guarantees correctness, it limits ExamShuffler's ability to reason about semantic relationships between questions.

Second, much of the workflow in CodeShuffler relies on manual instructor input. Instructors must explicitly define mappings between correct lines and plausible incorrect variants in order to generate semantically

meaningful partial-credit options. While this design prioritizes control and prevents the generation of nonsensical distractors, it forces manual input during exam authoring. This trade-off reflects a deliberate design choice favoring instructor oversight over fully automated inference, but it may limit adoption in settings where rapid content generation is required.

Due to this, CodeShuffler's notion of semantic correctness is limited to instructor-defined invariants. The system does not perform static analysis of generated code, and simply aims to preserve syntactic structure. As a result, some logically invalid but syntactically valid configurations may still pass shuffling unless explicitly changed by instructors.

Future work may address these limitations by extending the system's semantic awareness and automation capabilities. One direction is to introduce dependency annotations or grouping constraints within ExamShuffler, allowing instructors to specify question ordering requirements. Another promising avenue is the integration of generative AI techniques to assist with partial-credit option generation by proposing candidate incorrect variants or swaps that instructors can review and approve.

IX. Conclusion

In this work, we examined the challenges introduced by large language models in programming education, particularly their impact on grading validity, assessment integrity, and instructor workload. As generative AI tools become increasingly capable of producing correct and well-structured code, traditional take-home and auto-graded programming assessments are no longer sufficient for evaluating individual student understanding. Paper-based coding assessments have re-emerged as a viable alternative, but introduce substantial grading challenges, especially in large-enrollment courses.

To address this trade-off, we presented CodeShuffler v2, an extension of the original CodeShuffler framework designed to automate the creation of paper-based coding assessments. The proposed system is applicable across programming languages and course contexts, enabling instructors to assess foundational programming skills without relying on external tools. The result is an application that is robust, scalable, and intuitive for non-technical instructors.

Alongside the introduction of ExamShuffler, which enables scalable multi-version exam generation, CodeShuffler v2 provides a practical framework for administering fair and robust paper-based coding assessments in the era of generative AI.

X. Supplementary Materials

CodeShuffler and the assessment template are open-source and available for instructors to easily use and benefit from them. Both can be downloaded from the GitHub repository at https://github.com/hasanbaig/CodeShuffler.

XI. Acknowledgement

During the preparation of this work the author(s) used the ChatGPT tool to improve the language of the manuscript. After using this tool/service, the author reviewed and edited the content as needed and take full responsibility for the content of the published article.

This work was also supported by the University of Connecticut Office of the Vice President for Research (OVPR) Scholarship Facilitation Funds Award.

References

[1] National Student Clearinghouse Research Center, "Computer science has highest increase in bachelor's earners," https://www.studentclearinghouse.org/nscblog/computer-science-has-highest-increase-in-bachelors-earners/, 2024, accessed: 2026-01-20.

[2] C. L. Gordon, R. Lysecky, and F. Vahid, "The rise of program auto-grading in introductory cs courses: A case study of zylabs," in *2021 ASEE Virtual Annual Conference Content Access*. Virtual Conference: ASEE Conferences, 2021.

[3] OpenAI, "Chatgpt," https://openai.com/chatgpt, 2023, accessed: 2026-01-16.

[4] a. W. b. zyBooks, "zybooks interactive learning platform," https://www.zybooks.com/, 2025, accessed: 2025-01-19.

[5] H. Baig and P. Bradford, "Paper-based coding exams: Overcoming coding assessments and grading challenges in the era of ai large language models," in *Proceedings of the IEEE Frontiers in Education Conference (FIE)*, 2024.

[6] R. A. Fisher and F. Yates, "Statistical tables for biological, agricultural and medical research," 1938.