1828
AKADÉMIAI KIADÓ

Template-based question generation from code using static code analysis

Jawad Alshboul* and Erika Baksáné Varga

Faculty of Mechanical Engineering and Informatics, Institute of Information Science, University of Miskolc, Miskolc-Egyetemvaros, Hungary

Received: May 16, 2025 • Revised manuscript received: June 3, 2025 • Accepted: June 4, 2025
Published online: October 3, 2025

Pollack Periodica • An International Journal for Engineering and Information Sciences

21 (2026) 2, 71–79

DOI: 10.1556/606.2025.01413

ORIGINAL RESEARCH PAPER

ABSTRACT

This study presents a multilingual system for automatic question generation from source code in Python, C++, Java, and C. Using static code analysis, and template-based methods; it extracts code structure and generates questions aligned with Bloom's Taxonomy. A dataset of 456 questions from 19 algorithms and three code complexity levels was used. Current systems are monolingual; this approach handles four programming languages with a unified framework. The system was evaluated using several metrics, including the overall quality score. The automatic evaluation shows that C achieved a slightly higher overall quality score of 0.59, while the other languages scored 0.57. Human evaluation complements the automated evaluation, providing educators' perspective on question quality.

KEYWORDS

programming language assessment, multilingual code parsing, Bloom's taxonomy

1. INTRODUCTION

The manual creation of programming exercises remains time-consuming for educators, often taking hours to ensure questions align with specific learning objectives and code complexity levels [1]. This challenge intensifies in multilingual educational settings where instructors must simultaneously maintain question banks for multiple programming languages. Recent advances in static analysis frameworks and attribute grammar systems have laid the technical foundation for Automatic Question Generation (AQG) tools that parse code structures, extract semantic elements, and populate pedagogical templates [2, 3]. Traditional AQG systems relied heavily on template-based approaches that limited question diversity and contextual relevance [4]. Integrating Abstract Syntax Tree (AST) analysis with reference attribute grammars has enabled more sophisticated code element extraction, particularly for object-oriented languages like Java and C++ [5–7]. These technological advancements coincide with growing pedagogical demands for personalized learning pathways and competency-based assessment frameworks in computer science education [8]. Cross-language question generation introduces unique parsing challenges due to varying syntax rules and programming paradigms. There is no agreed-upon or standard evaluation metric for code-based question generation for educational purposes. The current few systems deal with one programming language (monolingual) without fully automated evaluation [1, 4]. As a result, this research's main added value is dealing with multilingual code-based question generation and automating the evaluation process.

This study presents a multilingual code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes. It focuses on code-based question generation using static code analysis. It offers pattern-based algorithm detection, structural analysis, and question templates. Pattern-based algorithm detection is performed through regex patterns. Structural analysis examines functions, loops, conditionals, and variables to generate relevant questions. Question templates involve predefined

*Corresponding author.
E-mail: alshboul.jawad@student.uni-miskolc.hu

templates for different code elements to create varied questions. The research objectives of this study are:

1. Developing a multilingual code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes (code-based question generation);
2. Developing an approach for automatically evaluating the proposed system based on a set of evaluation criteria through experiments on a real-world dataset to demonstrate its effectiveness in generating questions from program codes.

2. LITERATURE REVIEW

Early AQG systems focused primarily on Natural Language Processing (NLP) techniques applied to programming textbooks and documentation [9]. Most recent research focuses on generating questions from text, while some research focuses on generating questions from visuals or images [4]. The article [4] presents a hybrid approach to generate questions from Python program codes. However, there is a limitation concerning relying solely on the QuestGen artificial intelligence framework, which might lead to poorly phrased questions due to its nature and main purpose of focusing on generating questions from text.

Integrating of Bloom's Taxonomy into AQG frameworks marked a significant advancement in aligning educational technology with pedagogical objectives. This integration enables the generation of assessment items systematically mapped to cognitive skill levels, ensuring that instruction and evaluation are pedagogically sound and targeted to desired learning outcomes. Recent AQG systems utilize Bloom's Taxonomy to classify and generate questions that target specific cognitive levels, from basic recall (remembering) to higher-order thinking skills like learners' cognitive development and support differentiated instruction [10]. Static code analysis is employed across various domains, particularly in compiler design and security [11]. Static code analysis is used to automate checking student programming assignments. It verifies the correctness of student programming assignments concerning assignment instructions [12].

Most evaluations were conducted based on human evaluators, as it is illustrated by article [4], which relied solely on human evaluators in evaluating the generated questions. The quality assessment of machine-generated questions requires robust metrics beyond traditional automated scoring. Bloom's Taxonomy is a foundational framework for categorizing questions based on cognitive complexity [13]. It encompasses remembering, understanding, applying, analyzing, evaluating, and creating [13–15]. This taxonomy helps assess the cognitive skills that the questions aim to consider. Bloom's Taxonomy is used to classify educational learning objectives into levels of complexity and specificity. The following are the six levels from the simplest to the most complex:

1. *Remembering*: This is the basic level where learners must recall facts and concepts;
2. *Understanding*: Learners demonstrate comprehension by explaining ideas or concepts, summarizing information, and interpreting meaning;
3. *Applying*: It involves using knowledge in new situations to solve problems or complete tasks, demonstrating practical understanding;
4. *Analyzing*: Learners break down information into parts to understand its structure. They can differentiate between facts and inferences and identify relationships among various components;
5. *Evaluating*: Learners make judgments based on criteria and standards. They can critique ideas, assess the validity of arguments, and provide justification for their opinions;
6. *Creating*: This is the highest level of Bloom's Taxonomy, where learners combine elements to form a coherent or functional whole. They can design new products, propose solutions, or generate original ideas.

These levels are essential for educators to design assessments and questions that target various cognitive skills, ensuring a comprehensive evaluation of student learning. In the context of AQG, understanding these levels is crucial for creating questions that effectively assess students' knowledge and cognitive abilities.

3. METHODOLOGY

The four programming languages were chosen based on the up-to-date The Importance Of Being Earnest (TIOBE) Index, which indicates the popularity of programming languages. Python, C++, Java, and C are the most popular programming languages worldwide according to the TIOBE Index as of May 2025 [16]. While the paper [13] primarily focuses on general educational applications, it is important to note that modern adaptations of Bloom's Taxonomy can be tailored to specific domains, like programming. This adaptation allows for evaluating cognitive tasks unique to programming education, ensuring that the generated questions are relevant and effective for learners in that field. As a result, the methodology in the current research adopts Bloom's Taxonomy evaluation levels: remembering, understanding, applying, analyzing, evaluating, and creating.

Figure 1 shows the proposed methodology for a code-based multi-language question generator. The methodology behind involves several interconnected components that analyze code snippets and generate relevant questions. A detailed explanation of the methodology follows.

3.1. Language-specific parsing

The foundation of the system is a modular parsing that handles multiple programming languages:

1. *Language detection*: The system first identifies the programming language of the input code using heuristic

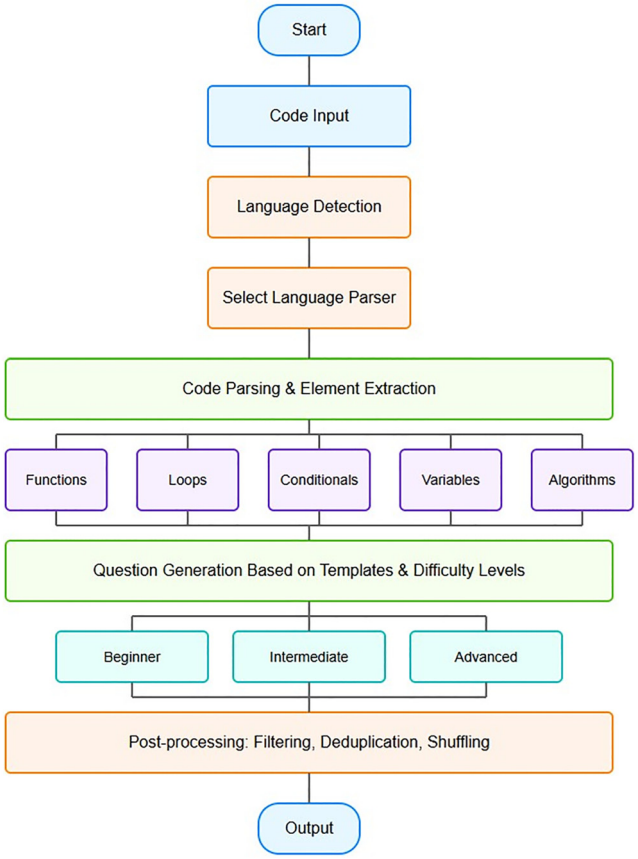

Fig. 1. Methodology for multilingual code-based question generation

pattern matching. This detection is based on language-specific keywords, syntax patterns, and structures;
2. *Language-specific parsers*: Each supported language (Python, Java, C++, and C) has a dedicated parser that implements the common code parser interface. This enables polymorphic handling of different languages while accounting for their unique characteristics;
3. *Python parser implementation*: For Python, the system leverages the AST module to perform deep structural analysis of the code. This provides detailed information about functions, loops, conditionals, and variables;
4. *Other language parsers*: For Java, C++, and C, the system implements regex-based parsers that identify key structural elements despite the lack of native AST support in Python for these languages.

3.2. Code element extraction

After parsing, the system extracts various structural elements from the code:

1. *Function analysis*: The system extracts information about functions, including their names, parameters; return statements, and recursion patterns;
2. *Loop detection*: The system identifies different types of loops (for/while) and extracts information about their variables and conditions;
3. *Conditional statement analysis*: For conditional statements (if/else), the system extracts conditions, identifies branch patterns, and determines nesting levels;
4. *Variable tracking*: The system extracts variables, their data types (when possible), initialization values, and tracks their modifications throughout the code;
5. *Algorithm identification*: Using a dictionary of algorithm-specific regex patterns, the system identifies common algorithms implemented in the code (e.g., binary search, sorting algorithms, and graph traversals).

3.3. Template-based question generation

The question generation process uses templates customized for different code elements and difficulty levels, as a sample is shown in Code 1:

1. *Difficulty stratification*: Questions are categorized into three difficulty levels - beginner, intermediate, and advanced - aligned with increasing cognitive complexity;
2. Element-specific templates: Each code element type (functions, loops, conditionals, variables, algorithms) has specific question templates designed to test understanding at different levels;
3. *Dynamic template parameters*: The system dynamically fills in template parameters with specific code elements. For example, function parameter examples are generated based on parameter names using heuristic rules.

Code 1. Sample of templates used for code-based question generation

```
'loop': { DifficultyLevel.BEGINNER: [
  "What is the purpose of the {type} loop on line {line_num}?",
  "How many times will the {type} loop on line {line_num} execute
  with typical input?", "What happens in each iteration of the
  {type} loop on line {line_num}?"],
```

3.4. Cognitive science-based question design

The templates are designed based on principles from cognitive science and educational theory, as it is shown in Code 1:

1. Bloom's Taxonomy alignment:
   a) Beginner questions focus on remembering and understanding (e.g., "What is the purpose of function X?");
   b) Intermediate questions target applying and analyzing (e.g., "Trace the execution of function X with inputs Y");
   c) Advanced questions emphasize evaluating and creating (e.g., "How could you optimize function X?");
2. Contextual relevance: Questions directly reference specific code elements, line numbers, and variable names from the input code to create contextually relevant assessments;
3. Balanced coverage: The system distributes questions across different code elements to ensure a comprehensive assessment of the code snippet.

3.5. Question post-processing

After generating candidate questions, the system applies several post-processing steps:

1. *De-duplication*: Eliminates duplicate or highly similar questions to ensure variety;
2. *Shuffling*: Randomizes the order of questions to prevent predictable patterns;
3. *Limiting*: Controls the number of questions to prevent overwhelming the user, while maintaining a balance of difficulty levels;
4. *Fallback strategies*: If specific elements cannot be extracted (e.g., due to parsing errors), the system falls back to more general questions about the code.

3.6. Evaluation approach

The methodology includes an evaluation approach to assess the quality of the generated questions. The evaluation of the proposed system is designed around a set of defined criteria. The methodology involves a structured approach to assess the quality of the generated questions across several key dimensions:

1. *Bloom's Taxonomy*: The Bloom's Taxonomy cognitive level distribution is computed by using Bloom's Taxonomy alignment to assess cognitive level distribution (remembering, understanding, applying, analyzing, evaluating, and creating);
2. *Difficulty distribution*: The questions are analyzed across three difficulty levels (Beginner, Intermediate, Advanced) for four programming languages: C, C++, Java, and Python;
3. *Linguistic complexity*: This dimension combines word count, sentence count, Flesch-Kincaid Grade Level, and average sentence length. All values are normalized to a 0-1 scale, with sentence length capped at 25 words and grade level capped at 10. The final score is computed using the formula:

$$Linguistic\ Complexity = \begin{cases} 0.6 \cdot Normalized\ Grade\ Level \\ +0.4 \cdot Normalized\ Sentence\ Length; \end{cases} \quad (1)$$

4. *Code coverage*: Measures how comprehensively the generated questions address different code components. The score is calculated as:

$$Code\ Coverage = \begin{cases} 0.4 \cdot Variables\ Coverage \\ +0.6 \cdot Functions\ Coverage; \end{cases} \quad (2)$$

5. *Precision*: Defined as the ratio of relevant or correct questions to the total number of questions generated by the system;
6. *Recall*: Assesses the system's ability to generate all relevant or expected questions, using code coverage as a proxy indicator for recall;
7. *Novelty*: Measures the originality of the generated questions using the formula:

$$Novelty = \begin{cases} 0.4 \cdot Bloom\ Score + 0.3 \cdot Code\ Elements \\ +0.3 \cdot Advanced\ Question\ Types; \end{cases} \quad (3)$$

8. *Educational alignment*: Evaluates how well the questions align with predefined learning objectives. The score is computed as:

$$Educational\ Alignment = \begin{cases} 0.7 \cdot Expected\ Bloom\ Match \\ +0.3 \cdot Expected\ Linguistic\ Complexity\ Match; \end{cases} \quad (4)$$

9. *Cognitive diversity*: Captures the diversity of cognitive skills involved in answering the questions. The formula used is:

$$Cognitive\ Diversity = 0.4 \cdot Bloom\ Score/6 + 0.6 \cdot Entropy, \quad (5)$$

$$Entropy = -\sum p \cdot \log(p)/\log(6), \quad (6)$$

and p denotes the proportion of questions at each Bloom's level. The weighted values are flexible and open to future refinement. For instance, future researchers might introduce additional variables, such as the density of technical terms, to further improve linguistic complexity estimation;
10. *Question quality score by language and difficulty*: The score is calculated through a multi-step process. First, computing eight different quality metrics for each question (linguistic complexity, code coverage, Bloom's distribution, precision, recall, novelty, educational alignment, and cognitive diversity). Second, combining these metrics with predetermined weights. Third, aggregating the scores by programming language and difficulty level;
11. *Quality score by code complexity*: The score is calculated through a multi-step process. First, computing eight different quality metrics for each question (linguistic complexity, code coverage, Bloom's distribution, precision, recall, novelty, educational alignment, and cognitive diversity). Second, combining these metrics with predetermined weights. Third, aggregating the scores by language and code complexity (simple, moderate, or complex).

4. RESULTS

The system analyzes code structure using language-specific parsers and generates questions at varying difficulty levels. The 114 questions for each programming language are evaluated based on 19 different algorithms and across three complexity levels (simple, moderate, and complex). The dataset of code snippets used is available on GitHub [17]. There are six generated questions for each algorithm in each programming language: two for beginners, two for intermediates, and two for advanced learners. The total number of generated questions is 456. Established educational assessment metrics, outlined in section 3.6 of the

Table 1. A sample transformation from code to question

Original Code	Template	Generated Question
def calculate_area (radius): return 3.14·radius·radius	"What does the {function_name} function calculate using {parameter}?"	"What does the calculate_area function calculate using radius?"
class Student: def __init__(self, name, age): self.name = name self.age = age	"What attributes does the {class_name} class initialize?"	"What attributes does the Student class initialize?"
try: result = x/y except ZeroDivisionError: result = 0	"What happens in this code when {error_type} occurs?"	"What happens in this code when ZeroDivisionError occurs?"

methodology, were used to evaluate the generated questions. The algorithms used are listed based on their fundamental categories:

1. Sorting Algorithms (Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort);
2. Searching Algorithms (Binary Search, Linear Search, and Knuth-Morris-Pratt);
3. Graph Traversal Algorithms (Depth-First Search, Breadth-First Search, and Topological Sort);
4. Shortest Path Algorithms (Dijkstra's, Floyd-Warshall, and A* Search);
5. Minimum Spanning Tree Algorithms (Kruskal's and Prim's);
6. Optimization & Problem-Solving Approaches (Dynamic Programming, Greedy, and Huffman Coding).

For the collected and prepared dataset, the following attributes are included:

1. *Functions, Loops, Conditionals, and Variables*: Each attribute is binary - 0 means the feature is not present in the code snippet, while 1 indicates it is present. All selected code examples include at least one instance of each of these four elements;
2. *Lines*: This attribute captures the length of the code, measured by the number of lines in each snippet;
3. *Complexity*: This is a categorical attribute with three levels - simple, moderate, and complex - reflecting the overall complexity of the code;
4. *Generated Questions*: The questions are primarily designed to require explanatory answers rather than simple yes/no or multiple-choice responses (open-ended questions). This field contains six automatically difficulty-tiered generated questions based on the input code: two aimed at beginner-level learners, two at intermediate level, and two at advanced level.

Table 1 shows a sample transformation from code to question.

Figure 2 shows the distribution of question difficulty levels (Advanced, Intermediate, and Beginner) across four programming languages: C, C++, Java, and Python. The proportions of difficulty levels are identical across all four languages. There is no noticeable skew toward a particular difficulty level for any specific language. In short, the difficulty level distribution is very evenly balanced across these languages. This deliberate balance ensures that one-third of the questions target each difficulty level, providing a well-rounded assessment experience.

Figure 3 presents Bloom's Taxonomy coverage. Each question was first analyzed to detect its cognitive level using keyword matching, with the level determined based on the highest number of keyword matches from Bloom's taxonomy. These levels were then mapped to numeric values (1–6) and normalized to a 0-1 scale for further analysis. For example, the system calculated the percentage of questions falling under each level, resulting in distributions of 16% for "Remember" and 8% for "Create". The generated questions

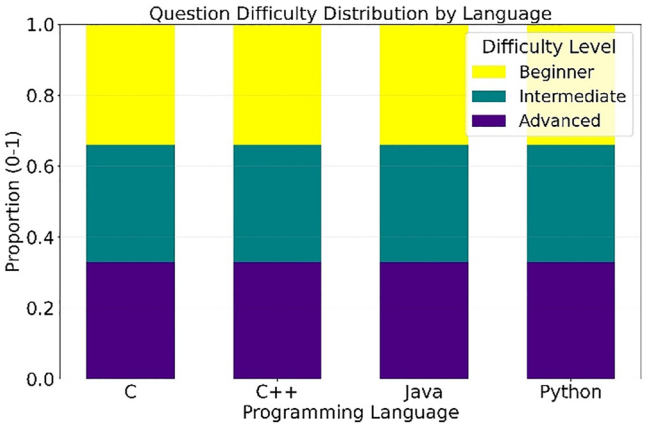

Fig. 2. Question difficulty distribution by language

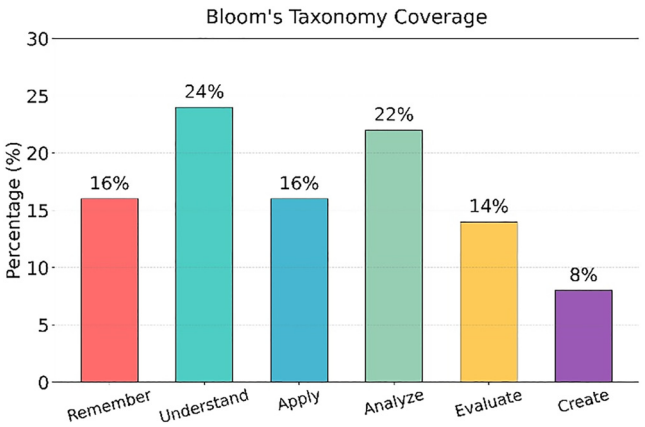

Fig. 3. Bloom's Taxonomy coverage across cognitive levels

demonstrated good coverage across cognitive levels, with a distribution of Remember: 16%, Understand: 24%, Apply: 16%, Analyze: 22%, Evaluate: 14%, and Create: 8%. This distribution indicates a balanced approach, with room for improvement in higher-order thinking (Create level).

Figure 4 reveals the question quality score by language and difficulty level. The scores shown in this visualization were calculated through a multi-step process. The overall quality scores cluster around the 0.55–0.60 range, indicating fairly consistent quality across difficulty levels and languages. It looks like beginner questions are generally better crafted or better received - maybe because they are simpler and easier to generate and validate.

Figure 5 focuses on the question quality score by language and code complexity. Across the board, none of the complexity levels dominate quality scores universally, which suggests that the quality of a question is not strictly tied to how simple or complex the code is.

Figure 6 shows the linguistic complexity of different programming languages across three difficulty levels. Linguistic complexity tends to increase with difficulty. Basic text metrics - word and sentence counts - were computed for each question to analyze sentence structure and length. Readability metrics, including Flesch-Kincaid Grade Level, were generated using the Textstat library to assess how readable and educationally appropriate the questions were. Average sentence length was also calculated to evaluate syntactic complexity. All metrics were normalized to a 0-1 scale for comparability, with sentence length capped at 25 words and grade level at 10. A final complexity score was derived using a weighted formula: 0.6 times the normalized grade plus 0.4 times the normalized sentence length. Scores were then aggregated by difficulty - Beginner, Intermediate, and Advanced - to analyze patterns across tiers.

Figure 7 shows that the average question diversity score varied by language, from 0.63 for C to 0.55 for C++. These scores were calculated using a structured, multi-step process with Shannon entropy to measure how evenly questions were distributed across templates and types. Unlike cognitive diversity, which reflects Bloom's taxonomy levels, this metric captures template usage patterns across algorithms and averages them per language. Scores were normalized to a 0-1 scale for cross-language comparison. C elicited the most diverse questions (0.63), followed by Java (0.59), Python (0.57), and C++ (0.55). This variation may stem from structural differences - C's lower-level constructs likely

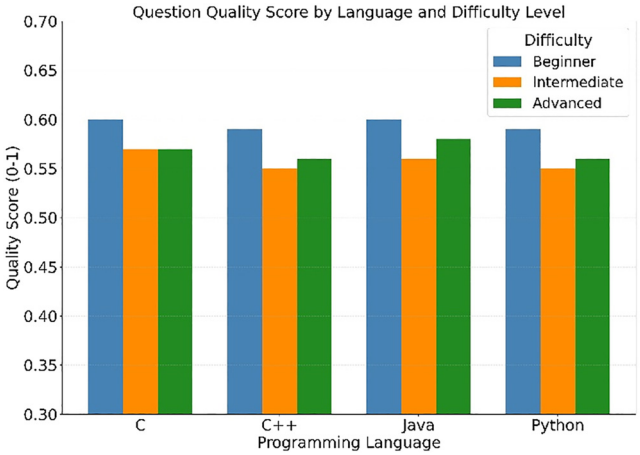

Fig. 4. Question quality score by language and difficulty level

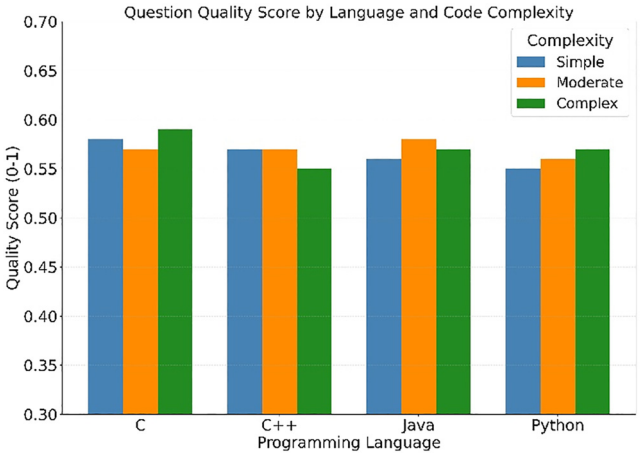

Fig. 5. Question quality score by language and code complexity

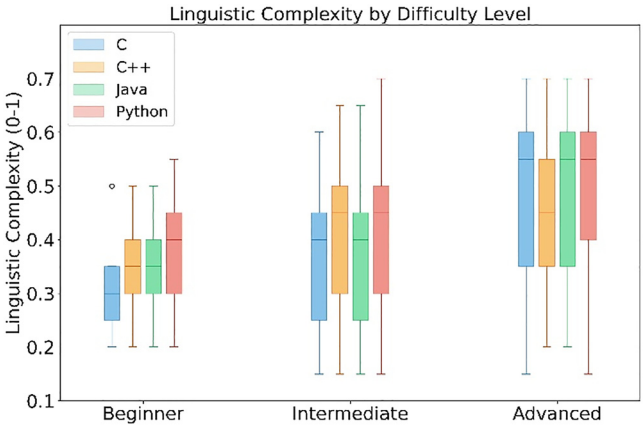

Fig. 6. Linguistic complexity by difficulty level

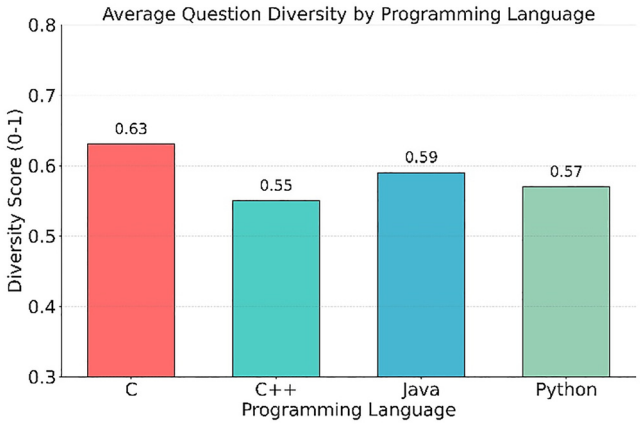

Fig. 7. Average question diversity by programming language

Table 2. Automatic evaluation results by programming language ($N = 456$)

Performance Metric	C	C++	Java	Python	Statistical Significance
Overall Quality Score	0.59	0.57	0.57	0.57	$F(3,452) = 5.01$, $P < 0.010$
Linguistic Complexity	0.35	0.37	0.39	0.44	$F(3,452) = 8.73$, $P < 0.001$
Code Coverage	1.00	1.00	1.00	1.00	No significant difference
Precision	0.36	0.35	0.35	0.39	$F(3,452) = 6.40$, $P < 0.001$
Recall	1.00	1.00	1.00	1.00	Perfect recall across all languages
F1-Score	0.53	0.52	0.52	0.56	$F(3,452) = 5.71$, $P < 0.001$
Novelty Score	0.17	0.14	0.15	0.15	$F(3,452) = 3.35$, $P < 0.050$
Educational Alignment	0.48	0.42	0.42	0.42	$F(3,452) = 7.91$, $P < 0.001$
Cognitive Diversity	0.53	0.50	0.52	0.50	$F(3,452) = 4.61$, $P < 0.010$

enable more varied questioning than C++'s standardized object-oriented patterns.

Table 2 shows automatic evaluation metrics for code-based question generation across four programming languages. C achieved a slightly higher overall quality score of 0.59, while the other languages scored 0.57. C code tends to be less syntactically ambiguous, allowing the system's static analysis and template-matching components to extract structural elements slightly better.

While the study employs well-defined metrics, the absence of human evaluation limits the contextual accuracy of generated questions. As a result, two human evaluators were used to complement the automatic evaluation. The manual metrics used are relevance and educational value of the questions. The human evaluators were allowed to rate based on their teaching experience. Relevance can cover code topic match, code context understanding, difficulty appropriateness, and clarity. Educational value can cover concept coverage, cognitive challenge, feedback potential, and engagement. The two evaluators were given the same 40 questions divided evenly between the four programming languages. Table 3 shows human evaluation metrics for code-based question generation across four programming languages. Table 3 shows C leads slightly. Python, Java, and C++ are tied at 3.45, showing a fairly even performance. Two tests were conducted to understand whether this slight difference has statistical significance. First, a paired t-test comparing C versus each of Python, Java, and C++ average scores as shown in Table 4. Two, one-way ANOVA comparing average scores across all four languages (F-statistic: 48.44, P-value: $1.01 \cdot 10^{-12}$ (very low)). The difference between C and other languages is very slight. Based on the table of paired t-tests and ANOVA results, the differences between C and the other languages are statistically significant, even if they were very slight.

Table 3. Human evaluation results by programming language ($N = 40$)

Metric	Python	Java	C++	C
Relevance	3.80	3.70	3.70	3.80
Educational Value	3.10	3.20	3.20	3.20
Average Score	3.45	3.45	3.45	3.50

Table 4. Paired t-test results for human evaluation differences

Comparison	t-statistic	P-value	Significant? ($\alpha = 0.05$)
C vs. Python	7.22	0.00005000 (very low)	Yes
C vs. Java	9.64	0.00000500 (very low)	Yes
C vs. C++	16.10	0.00000006 (very low)	Yes

The human evaluation complements the automated evaluation by validating key findings while providing educators' perspective on question quality. Both approaches consistently identified C as a better performer, though human evaluation revealed more balanced performance across languages than suggested by automated metrics alone. The convergence between automated educational alignment scores and human-assessed educational value demonstrates the validity of computational metrics for educational applications. However, the human evaluation's emphasis on practical teaching utility provides essential context that purely computational measures cannot capture, highlighting the importance of multi-faceted evaluation approaches in educational technology research.

5. DISCUSSION

5.1. Research contributions

This methodology introduces several key contributions to automated programming question generation. Unlike many existing systems focusing on a single programming language, this approach handles four languages with a unified framework. It combines AST-based parsing (for Python) with regex-based parsing (for other languages) to achieve broad language coverage without sacrificing depth of analysis. It implements a pattern-based approach to identify common algorithms in code, enabling algorithm-specific questions. It systematically categorizes questions into different difficulty levels based on cognitive complexity rather than arbitrary designations. It generates example parameters for function calls based on parameter names, creating more realistic and contextually appropriate questions. Finally, it ensures questions cover multiple aspects of programming knowledge.

5.2. Limitations

While this study's results are promising, it is important to acknowledge certain limitations. The current methodology has several limitations that suggest directions for future research. The regex-based parsing for Java, C++, and C is less precise than AST-based one, which may affect question quality. The current approach relies on static code analysis and does not include dynamic runtime behavior analysis. The system recognizes structural patterns but has limited understanding of the semantic purpose of the code. Finally, the fixed templates may become predictable with extended use.

5.3. Future directions

Future improvements could include using language-specific parsers for each supported language, incorporating machine learning for more adaptive question generation, adding dynamic code execution analysis, implementing more sophisticated algorithm detection, developing context-aware template generation, and investigating the educational effectiveness of automatically generated questions through student performance analysis.

6. CONCLUSION

This research developed and evaluated a template-based approach using static code analysis for automated question generation from source code. By leveraging Abstract Syntax Trees (ASTs) and predefined templates, the system effectively generated contextually relevant questions across multiple programming languages, addressing a core challenge in programming education. Experimental results showed consistent quality across C (0.59), Java (0.57), Python (0.57), and C++ (0.57). Expert evaluations rated the system's utility between 3.45 and 3.50 across languages, with significant statistical support ($F = 48.44$, $P = 1.01 \cdot 10^{-12}$), confirming its practical applicability. The generated questions spanned all six Bloom's taxonomy levels - 16% Remember, 24% Understand, 16% Apply, 22% Analyze, 14% Evaluate, and 8% Create - maintaining an identical distribution across all languages. This balanced cognitive coverage underscores the system's ability to support comprehensive learning assessments. This work offers a multilingual code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes and an approach for automatically evaluating the proposed system based on a set of evaluation criteria complemented by human evaluation metrics. Current diversity and quality scores highlight room for improvement. Future work should expand template libraries, improve the question-generation filtering process to increase precision, incorporate machine learning to enhance quality, and conduct longitudinal studies to assess learning outcomes over time. The proposed system provides a validated foundation for scalable, automated assessment in programming education. With good quantitative support (quality: 0.59–0.57; cognitive diversity: 0.50–0.53; expert rating: 3.45–3.50), it offers a practical, adaptable tool for educators. In summary, this work marks a promising early-stage (baseline) system toward intelligent, scalable multilingual assessment systems - bridging static analysis and educational theory to meet the evolving demands of computer science education.

REFERENCES

[1] J. Alshboul and E. Baksa-Varga, "A review of automatic question generation in teaching programming," *Int. J. Adv. Computer Sci. Appl.*, vol. 13, no. 10, pp. 45–51, 2022.

[2] I. Riouak, N. Fors, J. Öqvist, G. Hedin, and C. Reichenbach, "Efficient demand evaluation of fixed-point attributes using static analysis," in *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, Pasadena, CA USA, October 20–21, 2024, pp. 56–69.

[3] G. Son, H. Ko, H. Lee, Y. Kim, and S. Hong, "Multilingual challenges in automated evaluators: A case study on English and Korean," *OpenReview*, 2025. [Online]. Available: https://openreview.net/forum?id=8NIUi6Ha1f. Accessed: May 16, 2025.

[4] J. Alshboul and E. Baksa-Varga, "A hybrid approach for automatic question generation from program codes," *Int. J. Adv. Computer Sci. Appl.*, vol. 15, no. 1, pp. 10–17, 2024.

[5] M. Hidvégi, G. Mezei, and S. Bácsi, "The challenges of visualizing DMLA models," *Pollack Period.*, vol. 16, no. 3, pp. 13–19, 2021.

[6] S. Breese, A. Milanova, and B. Cutler, "Using static analysis for automated assignment grading in introductory programming classes," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, Seattle, Washington USA, March 8–11, 2017, pp. 704.

[7] S. Rakangor and Y. Ghodasara, "Literature review of automatic question generation systems," *Int. J. Scientific Res. Publications*, vol. 5, no. 1, pages 1–5, 2015.

[8] G. Kurdi, J. Leo, B. Parsia, U. Sattler, and S. Al-Emari, "A systematic review of automatic question generation for educational purposes," *Int. J. Artif. Intelligence Educ.*, vol. 30, no. 1, pp. 121–204, 2020.

[9] N. Mulla and P. Gharpure, "Automatic question generation: A review of methodologies, datasets, evaluation metrics, and applications," *Prog. Artif. Intelligence*, vol. 12, no. 1, pp. 1–32, 2023.

[10] G. Deena, K. Raja, and K. Kannan, "An automatic question generation system using rule-based approach in Bloom's Taxonomy," *Recent Adv. Computer Sci. Commun.*, vol. 14, no. 5, pp. 1477–1487, 2019.

[11] M. Alqaradaghi, G. Morse, and T. Kozsik, "Detecting security vulnerabilities with static analysis - A case study," *Pollack Period.*, vol. 17, no. 2, pp. 1–7, 2021.

[12] K. Sterner, "Automated checking of programming assignments using static analysis," BSc Thesis, Mälardalens University, Sweden, 2021.

[13] L. L. Shwe, S. Matayong, and S. Witosurapot, "Enabling cognitive and unified similarity-based difficulty ranking mechanisms for

AQG on multimedia content," *Expert Syst. Appl.*, vol. 277, 2025, Art no. 127244.

[14] E. H. S. Y. Elim, "Promoting cognitive skills in AI-supported learning environments: The integration of Bloom's Taxonomy," *Education 3–13*, pp. 1–11, 2024.

[15] B. Khoy, "Unlocking cognitive learning objectives: A comprehensive evaluation of how textbooks and syllabi align with revised Bloom's Taxonomy across disciplines," *Curriculum Perspect.*, 2025. [Online]. Available: https://doi.org/10.1007/s41297-024-00295-2. Accessed: May 16, 2025.

[16] P. Jansen, "The TIOBE programming community index," 2025. [Online]. Available: https://www.tiobe.com/tiobe-index/. Accessed: May 16, 2025.

[17] J. Alshboul, MultilingualCodeBasedQG, GitHub, 2025. [Online]. Available: https://github.com/jalshboul/MultilingualCodeBasedQG. Accessed: May 16, 2025.